МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Изучаем программирование на C

Открой для себя секреты гуру-программистов

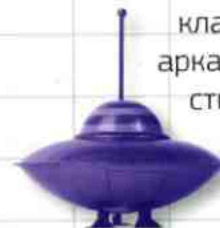
Узнай, как утилита make может изменить твою жизнь

Научись избегать глупых ошибок при работе с указателями

Представь, как функции с переменным параметром помогают Сью быть более гибкой

Порезвись в стандартной библиотеке C

Создай классическую аркадную игру в стиле ретро



Дэвид Гриффитс
Дон Гриффитс

O'REILLY®

МИРОВОЙ
КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Head First C

David Griffiths &
Dawn Griffiths

O'Reilly

МИРОВОЙ
КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Изучаем программирование на C

Дэвид Гриффитс
Дон Гриффитс

Москва

УДК 004.43
ББК 32.973.26-018.1
Г 85

Authorized Russian translation of the English edition of Head First C,
1st Edition © 2012 David Griffiths and Dawn Griffiths (ISBN 9781449399917).
This translation is published and sold by permission of O'Reilly Media, Inc.,
which owns or controls all rights to publish and sell the same.

Перевод с английского ООО «Айдиономикс»

Гриффитс Д.

Г 85 Изучаем программирование на С ; пер. с англ. / Дэвид Гриффитс, Дон Гриффитс. – М. : Эксмо, 2013. – 624 с. : ил. – (Мировой компьютерный бестселлер).

ISBN 978-5-699-60233-9

Вы всегда мечтали о том, чтобы найти более легкий способ изучения программирования на С? «Изучаем программирование на С» предлагает методику, с помощью которой вы научитесь создавать программы на этом языке. В книге используется уникальный подход, который выходит за рамки синтаксиса и пошаговых руководств и поможет вам стать отличным программистом. Вы изучите ключевые моменты, в том числе основы языка, динамическое управление памятью, указатели и арифметические операции с ними. А благодаря более продвинутым темам, таким как многопоточность и сетевое программирование, «Изучаем программирование на С» может рассматриваться в качестве учебника для студентов.

Практические задания помогут усовершенствовать ваши способности, проверить приобретенные вами навыки и сделать вас более уверенным в себе.

УДК 004.43
ББК 32.973.26-018.1

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения ООО «Издательство «Эксмо».

ISBN 978-5-699-60233-9

© ООО «Айдиономикс», перевод на русский язык, 2013
© Оформление. ООО «Издательство «Эксмо», 2013

Посвящается Деннису Ритчи (1941–2011), отцу языка Си

Авторы

Дэвид Гриффитс

Дон Гриффитс

Дэвид Гриффитс (David Griffiths) начал программировать в возрасте 12 лет, посмотрев документальный фильм о работе Сеймура Пейперта (Seymour Papert). В 15 лет он написал реализацию компьютерного языка LOGO, придуманного Пейпертом. После изучения теоретической математики в университете Дэвид начал писать программы для компьютеров и журнальные статьи для людей. Он успел поработать консультантом по Agile, программистом и служащим в автомастерской но не в таком порядке. Дэвид может писать код на более чем десяти языках, хотя разговаривает только на одном. В свободное от написания книг, программирования и консультаций время предпочитает путешествовать вместе со своей любимой женой и соавтором Дон.

До того как Дэвид принялся за написание этой книги, на его счету уже были два других издания из цикла *Head First: Head First Rails* и *Head First Programming*.

Вы можете посетить его страничку в «Твиттере»: *http://twitter.com/dogriffiths*

Дон Гриффитс (Dawn Griffiths) начала свой славный путь в престижном университете Великобритании, где получила диплом по математике с отличием, после чего решила связать карьеру с разработкой программного обеспечения. Сейчас она имеет за плечами 15-летний опыт работы в IT-индустрии.

Прежде чем присоединиться к Дэвиду в работе над этой книгой, Дон успела закончить две другие книги из цикла *Head First: Head First Statistics* и *Head First 2D Geometry*, а также принять участие в создании множества различных изданий этой серии.

Когда у Дон выпадает свободная минута, вы можете застать ее за оттачиванием навыков в тай-цзи, за плетением кружев, готовкой или на пробежке. Она также любит путешествовать и проводить время со своим мужем Дэвидом.

Оглавление (краткое)

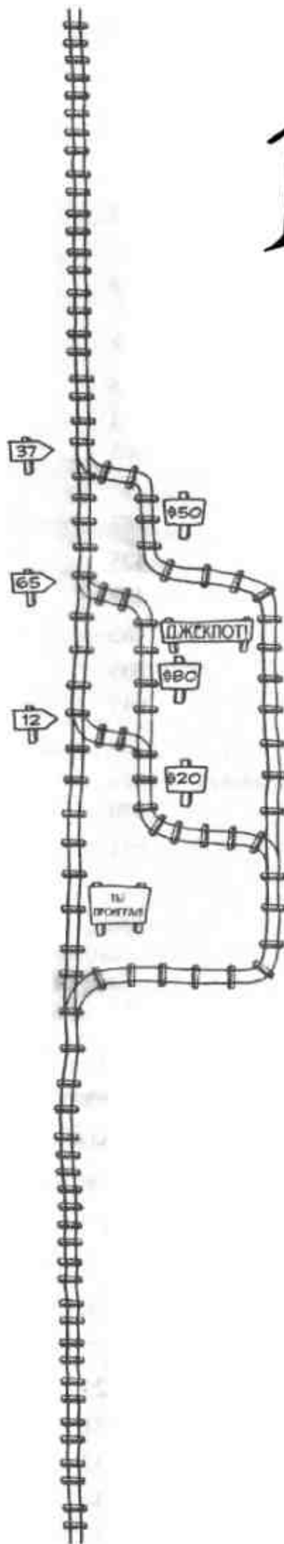
Введение: *Как пользоваться этой книгой* 25
1. Начинаем работать с языком Си: *Погружаемся* 37
2. Память и указатели: *На что ты указываешь?* 77
2.5. Строки: *Теория строк* 119
3. Создание небольших простых инструментов: *Делайте что-то одно, но делайте это хорошо* 139
4. Использование нескольких исходных файлов: *Разбиваем на части, собираем вместе* 193
Лабораторная работа 1: *Arduino* 243
5. Структуры, объединения и битовые поля: *Создавайте собственные структуры* 253
6. Структуры данных и динамическая память: *Наводим мосты* 303
7. Продвинутые функции: *Выжмите из своих функций все соки* 347
8. Статические и динамические библиотеки: *Легко заменяемый код* 387
Лабораторная работа 2: *OpenCV* 425
9. Процессы и системные вызовы: *Разрушая границы* 433
10. Межпроцессное взаимодействие: *Общение — это хорошо* 465
11. Сокеты и работа в Сети: *Нет места лучше, чем 127.0.0.1* 503
12. Потоки: *Это параллельный мир* 537
Лабораторная работа 3: *Бластероиды* 559
Приложение I. На закуску: *Топ-10 фактов (которым мы не уделили внимание)* 575
Приложение II. Все темы: *Вспомнить все* 589
Алфавитный указатель 611

Оглавление (полное)

Введение

Ваш мозг по отношению к Си. Когда *вы* стараетесь что-то изучить, ваш *мозг* пытается оказать вам услугу, убеждая, что все это *не имеет никакого значения*. Он думает: «Лучше сосредоточиться на чем-то более важном, например на том, как избежать встречи со свирепым хищником, или на том, что кататься на сноуборде нагишом — это плохая затея». Как же *убедить* мозг, что ваша жизнь зависит от знания Си?

Для кого эта книга? 26
Мы знаем, о чем подумали вы 27
Метапознание: размышления о мышлении 29
Вот как вы можете подчинить себе свой мозг 31
Прочтите 32
Команда технических рецензентов 34
Благодарности 35



Начинаем работать с языком Си

1 Погружаемся

Хотите узнать, как думает компьютер?

Вам нужно написать **высокопроизводительный код** для новой игры? Запрограммировать контроллер **Arduino**? Или использовать продвинутую **стороннюю библиотеку** в своем приложении для iPhone? Тогда язык Си вам поможет. Си работает на **гораздо более низком уровне**, чем большинство других языков программирования, поэтому понимание Си позволит вам лучше разобраться в том, **что на самом деле происходит внутри компьютера**. Си также может помочь в освоении других языков. Поэтому хватайте свой компилятор и немедленно приступайте к делу.

Си — это язык для компактных быстрых программ	38
Как выглядит программа на Си целиком	41
Но как мы запустим программу?	45
Два вида команд	50
Код, который мы имеем в настоящий момент	51
Подсчет карт? В Си?	53
Кроме равенства есть и другие логические выражения	54
Инструкция switch: вот так поворот	62
Иногда одного раза недостаточно…	65
Циклы часто имеют одинаковую структуру	66
Чтобы прервать цикл, используйте break…	67
Ваш инструментарий языка Си	76

Память и указатели

2 На что ты указываешь?

Чтобы сделать что-то серьезное с помощью Си, необходимо разобраться, как этот язык управляет памятью.

Язык Си позволяет вам лучше *контролировать* использование **компьютерной памяти** в программе. В этой главе вы приоткроете занавес и узнаете, что именно происходит, когда вы **считываете и записываете переменные**. Вы изучите **принцип работы массивов**, научитесь избегать **грубых ошибок при работе с памятью** и, что самое главное, поймете, что умелое обращение с **указателями и адресацией** — это ключ к высококлассному программированию на Си.

Код на языке Си содержит указатели	78
В недрах памяти	79
Отправляемся в плавание с указателями	80
Попробуйте передать указатель переменной	83
Использование указателей в памяти	84
Как передать строку в функцию	89
Переменные массивов похожи на указатели	90
О чем думает компьютер, когда выполняет ваш код	91
Переменные массивов и указатели не одно и то же	95
Почему массивы на самом деле начинаются с нуля	97
Использование указателей для ввода данных	101
Будьте осторожны при использовании scanf()	102
fgets() — альтернатива функции scanf()	103
Строковый литерал никогда не может быть изменен	108
Если вы собираетесь изменить строку — сделайте копию	110
Как устроена память	116
Ваш инструментарий языка Си	117

Строки

2.5 Теория строк

Чтение — это не все, что можно делать со строками.

Вы уже знаете, что строки в Си на самом деле являются *массивами символов*, но что данный язык позволяет с ними *делать*? Вот здесь нам и пригодится библиотека *string.h*. Она входит в состав стандартной библиотеки Си и отвечает за **манипуляции над строками**. Используя *string.h*, вы можете их ***объединять, копировать*** или ***сравнивать***. В этой главе вы узнаете, как создавать **массивы строк**, а затем тщательно рассмотрите процесс ***поиска внутри строк*** с помощью функции **`strstr()`**.

В отчаянных поисках Фрэнка	120
Создание массива массивов	121
Находим строки, содержащие искомый текст	122
Использование функции strstr	125
Пришло время рассмотреть наш код	130
Массив массивов и массив указателей	134
Ваш инструментарий языка Си	137

3 Создание простых инструментов

Делайте что-то одно, но делайте это хорошо

Любая операционная система включает в себя простые инструменты.

Простые инструменты выполняют **специальные небольшие задачи**, такие как чтение и запись файлов или фильтрация данных. Для более сложных задач вы можете даже *соединить несколько инструментов вместе*. Как же создаются эти простые инструменты? В данной главе вы рассмотрите стандартные блоки, на основе которых они строятся, а также узнаете, как работать с **аргументами командной строки**, как управлять **потоками информации** и **перенаправлять** ее. Не теряйте время, вооружайтесь инструментами.

Простые инструменты могут решать сложные задачи	140
Вот как должна работать эта программа	144
Но мы не используем файлы…	145
Мы можем использовать перенаправление	146
Представляем стандартный поток ошибок	156
По умолчанию стандартный поток ошибок выводится на экран	157
Функция fprintf() печатает в файловый дескриптор	158
Давайте обновим наш код с учетом использования fprintf	159
Простые инструменты обладают гибкостью	164
Не изменяйте утилиту geo2json	165
Для разных задач требуются разные инструменты	166
Соедините ваш ввод и вывод с помощью канала	167
Утилита bermuda	168
Но что если мы хотим вывести данные в несколько файлов?	173
Создайте собственный поток данных	174
У функции main() есть кое-что, о чем мы пока не говорили	177
Позвольте библиотеке сделать работу за вас	185
Ваш инструментарий языка Си	192

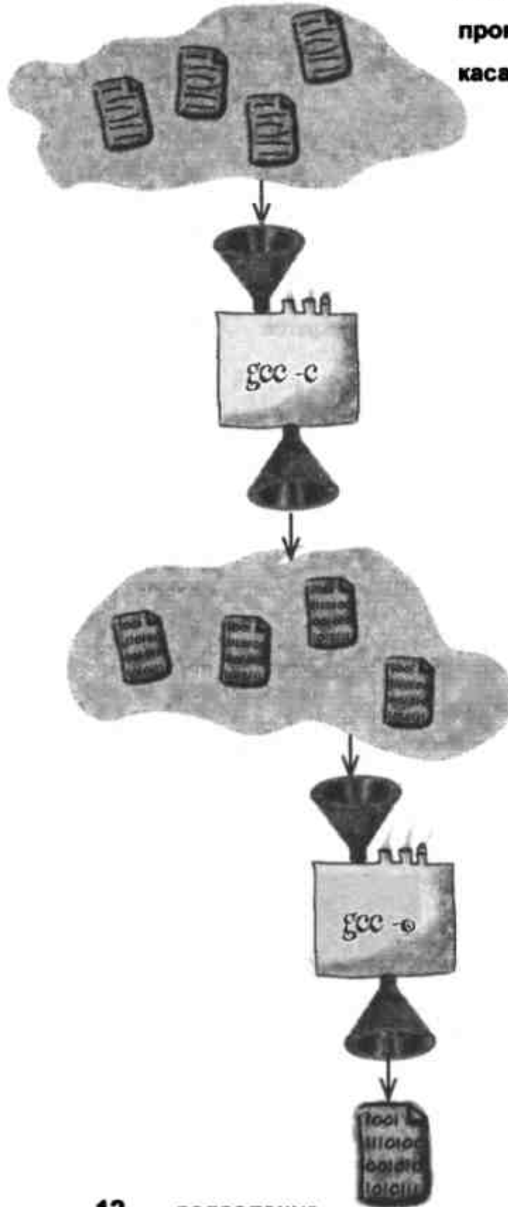
Использование нескольких исходных файлов

Разбиваем на части, собираем вместе

4

Для большой программы лучше не использовать один объемный исходный файл.

Представьте себе, какой сложной и трудоемкой может оказаться поддержка одного-единственного файла программы в масштабах предприятия. В этой главе вы узнаете, как с помощью Си разбить исходный код на **небольшие легко управляемые части**, а потом собрать их в **одну большую программу**. Заодно сможете более подробно изучить некоторые **тонкости, касающиеся типов данных**, и познакомиться с утилитой `make`.

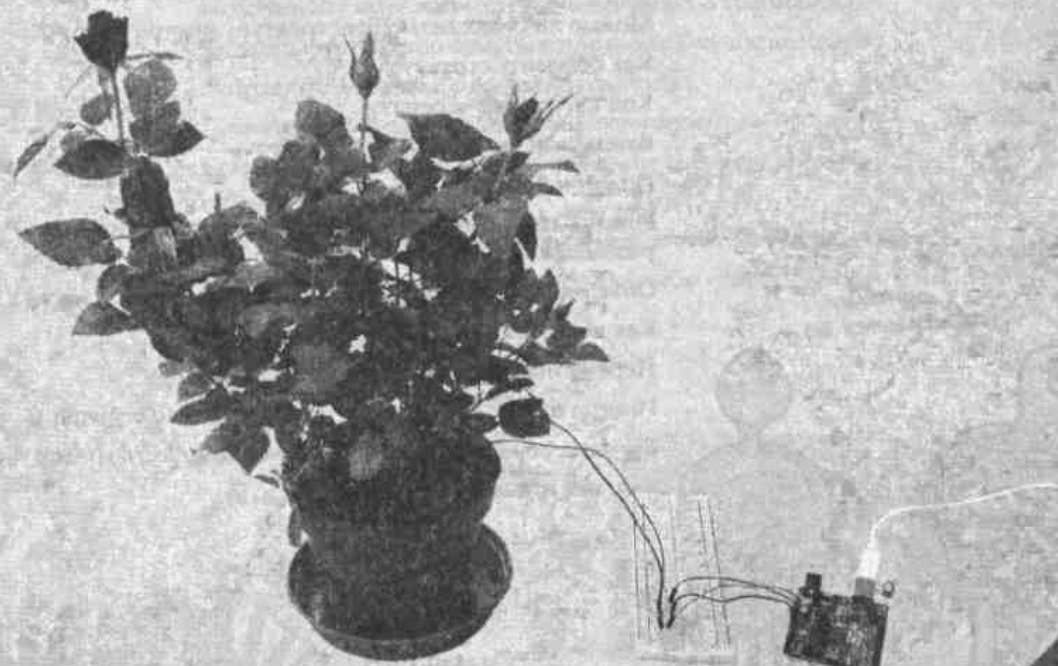


Краткое руководство по типам данных	198
Не помещайте что-то большое во что-то маленькое	199
Приведение типа float к целому числу	200
О нет… Это же безработные актеры…	204
Давайте посмотрим, что случилось с кодом	205
Компиляторы не любят сюрпризы	207
Отделяем объявление от определения	209
Создаем ваш первый заголовочный файл	210
Если у вас есть набор часто используемых возможностей…	218
Вы можете разделить код на отдельные файлы	219
Взгляд на компиляцию изнутри	220
Для разделяемого кода потребуется отдельный заголовочный файл	222
Это ведь не космические технологии… правда?	225
Не нужно перекомпилировать каждый файл	226
Сначала вы компилируете исходный код в объектные файлы	227
Сложно уследить за всеми файлами	232
Автоматизируйте процесс сборки с помощью утилиты make	234
Как работает утилита make	235
Сообщите утилите make о своем коде с помощью файла makefile	236
Пуск!	241
Ваш инструментарий языка Си	242

Лабораторная работа 1

Arduino

Мечтали ли вы когда-нибудь о том, чтобы ваши растения сами говорили, когда их нужно поливать? Что ж, с помощью Arduino они могут это делать! В лабораторной работе 1 вы напишете систему мониторинга растений на основе Arduino на языке Си.



Структуры, объединения и битовые поля

5 Создавайте собственные структуры

В жизни далеко не все можно свести к простым числам. До сих пор мы рассматривали базовые типы данных языка Си, но что если вы хотите выйти за пределы чисел и фрагментов текста и **смоделировать что-то, имеющее отношение к реальному миру**? Это можно сделать с помощью ***структур***, создавая свои собственные типы. Мы покажем, как **базовые типы данных** могут сочетаться в структурах, и даже рассмотрим так называемые ***объединения***, с помощью которых можно **передать неопределенность, встречающуюся в реальной жизни**. А если вам нужны простые ответы «да» или «нет», то ***битовые поля*** могут оказаться именно тем, что вам необходимо.

Иногда приходится иметь дело с большими объемами данных	254
Разговор в офисе	255
Создавайте собственные структурированные типы данных с помощью структур	256
Просто дайте им рыбу	257
Вы можете считывать поля структуры с помощью оператора «точка»	258
Можно ли поместить одну структуру внутрь другой?	263
Как обновить структуру	272
Код клонирует черепаху	274
Вам нужен указатель на структуру	275
(*t).age или *t.age	276
Иногда для одного и того же типа сущностей нужны разные типы данных	282
Объединения позволяют повторно использовать место в памяти	283
Как пользоваться объединениями	284
Переменная типа enum хранит обозначения	291
Иногда нужен контроль на уровне отдельных битов	297
Битовые поля хранят определенное количество битов	298
Ваш инструментарий языка Си	302

6 Структуры данных и динамическая память

Наводим мосты

Иногда одной структуры недостаточно.

Для моделирования сложных требований к данным нужно **соединить несколько структур**. В этой главе вы узнаете, как с помощью **указателей** объединять нестандартные типы данных в **большие сложные структуры**. В процессе создания **связных списков** вы изучите *ключевые принципы* этого подхода, а также научитесь создавать структуры, совместимые с данными разных объемов, **динамически выделяя память в куче** и освобождая ее в момент, когда работа с ней уже закончена. И если организация всего этого процесса слишком усложнится, вам поможет `valgrind`, о котором мы также поговорим.

Нужно ли вам гибкое хранилище?	304
Связные списки похожи на цепочки данных	305
Связные списки поддерживают вставку данных	306
Создаем рекурсивную структуру	307
Создадим острова с помощью Си…	308
Вставляем значения в список	309
Используйте кучу в качестве динамического хранилища	314
Возвращайте обратно память, которая вам больше не нужна	315
Запрашиваем память с помощью malloc()…	316
Давайте исправим наш код, применив функцию strdup()	322
Освободите память после того, как закончите с ней работать	326
Обзор системы ЭСИПЛ	336
Экспертиза программного обеспечения: использование valgrind	338
Используйте valgrind несколько раз, чтобы собрать больше улик	339
Рассмотрим улики	340
Проверяем на практике наше решение	343
Ваш инструментарий языка Си	345

7

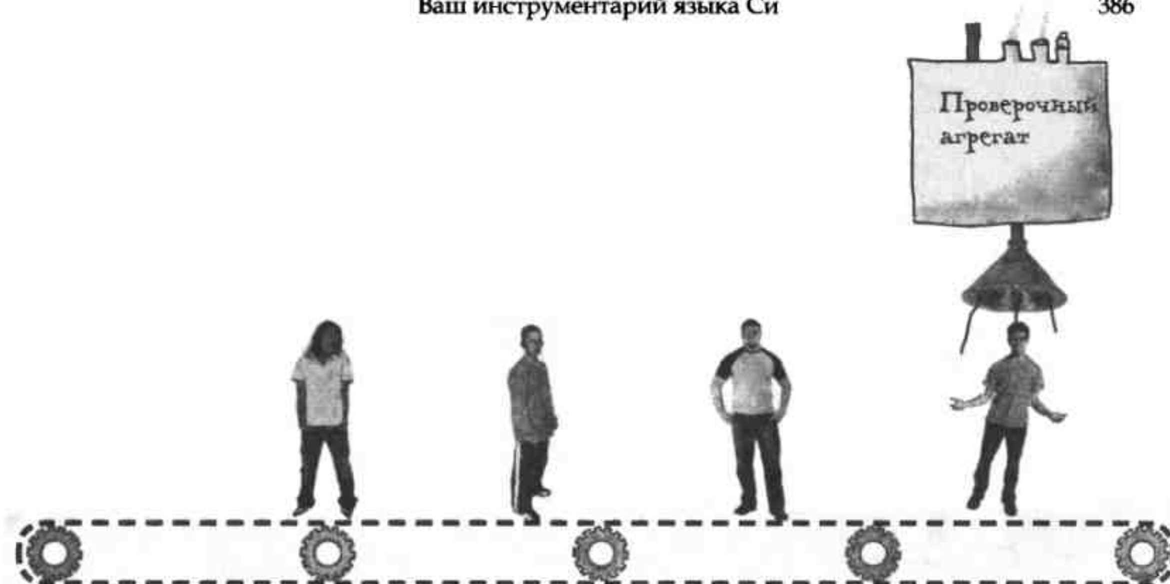
Продвинутые функции

Выжмите из своих функций все соки

Базовые функции — это хорошо, но иногда нужно нечто большее.

До сих пор мы уделяли внимание основам, но что если для достижения результата вам потребуется что-то более *мощное* и *гибкое*? В этой главе вы узнаете, как усовершенствовать код, передавая функции в качестве параметров, научитесь сортировать элементы с помощью функций сравнения и в завершение увидите, как можно сделать код *супергибким*, используя функции с переменным числом аргументов.

В поисках идеала	348
Передача кода в функцию	352
Нам нужно передать в find() имя функции	353
Имя функции является указателем на эту функцию…	354
…но такого типа данных, как функция, не существует	355
Как создать указатель на функцию	356
Выполняем сортировку с помощью стандартной библиотеки Си	361
Наводим порядок с помощью указателей на функции	362
Автоматизируем составление уведомительных писем	370
Создаем массив указателей на функции	374
Сделайте свои функции эластииииичными	379
Ваш инструментарий языка Си	386



8 Легко заменяемый код

Вы уже знаете, какой мощью обладают стандартные библиотеки.

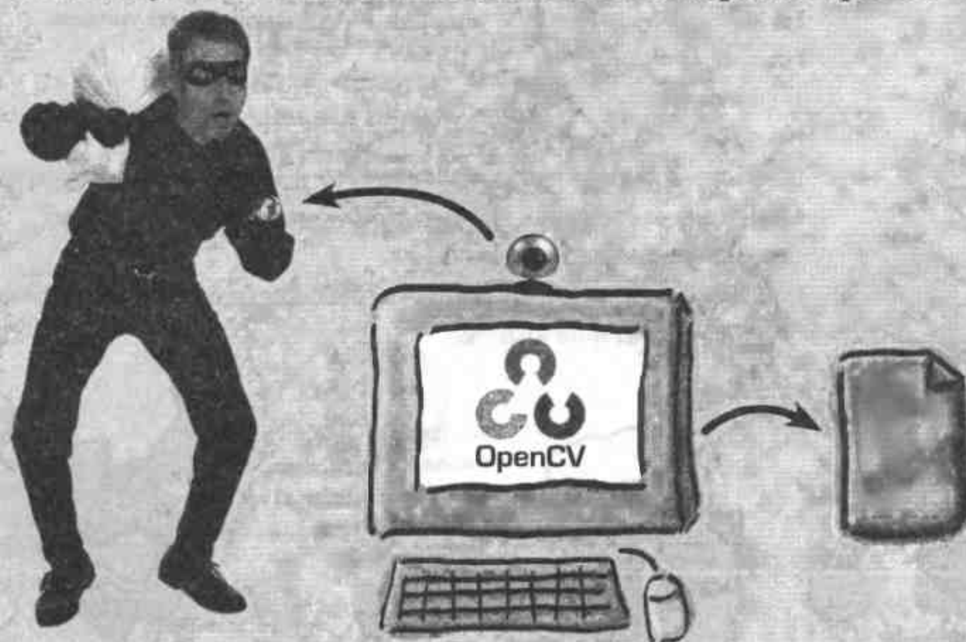
Пришло время применить эту мощь в *собственном* коде. Здесь вы узнаете, как создавать **собственные библиотеки** и **использовать один и тот же код в нескольких приложениях**. Вы научитесь разделять код во время выполнения программы с помощью **динамических библиотек**. И дочитав эту главу, вы сможете писать масштабируемый, простой и эффективный в управлении код.

Код, который вы можете принести с собой в банк	388
Угловые скобки предназначены для стандартных заголовков	390
Но что если вы хотите разделять код?	391
Разделяем заголовочные файлы	392
Разделяем объектные файлы .o с помощью полного пути	393
Архив содержит объектные файлы	394
Создайте архив с помощью команды ar…	395
И наконец, скомпилируйте другие свои программы	396
Тренажерный зал Head First выходит на международный уровень	401
Подсчет калорий	402
Но все немного сложнее…	405
Программы состоят из множества частей…	406
Динамическая компоновка происходит во время выполнения программы	408
Можно ли скомпоновать архив во время выполнения программы?	409
Сначала создадим объектный файл	410
На разных платформах динамические библиотеки называются по-разному	411
Ваш инструментарий языка Си	423

Лабораторная работа 2

OpenCV

Представьте, что компьютер в ваше отсутствие может присматривать за домом и сообщать о подозрительных личностях, которые бродят вокруг. В лабораторной работе 2 вы создадите детектор для обнаружения незваных гостей, используя возможности языка Си и мастерство OpenCV.



Процессы и системные вызовы

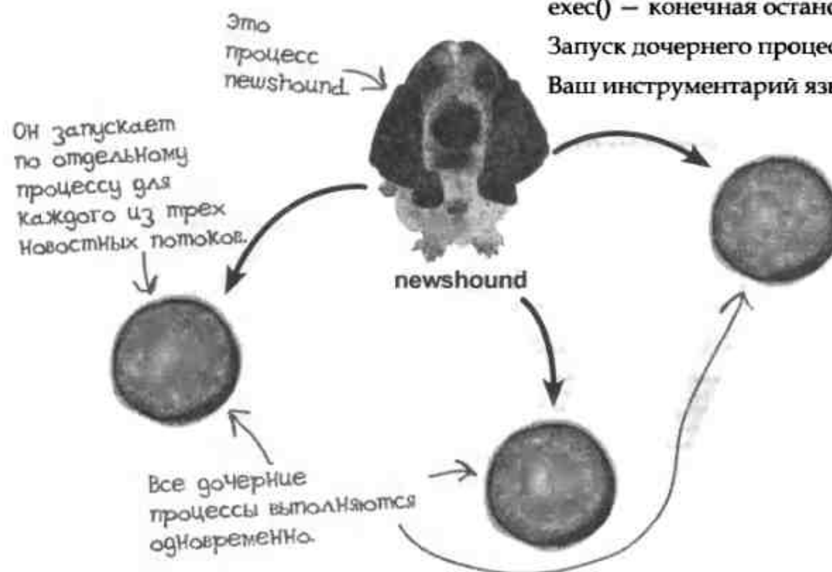
Разрушая границы

9

Пришло время мыслить нестандартно.

Вы уже знаете, что с помощью командной строки можно собирать сложные приложения из небольших утилит. Но что если вы хотите *вызывать другие программы* из собственного кода? В этой главе вы научитесь использовать системные сервисы для создания *процессов* и управления ими. Благодаря этому вы получите доступ к *электронной почте*, *браузеру* и *любому установленному внешнему приложению*. К концу этой главы вы сможете выйти *за рамки языка Си*.

Системные вызовы — это ваша прямая связь с операционной системой	434
Кто-то проник в систему	438
Безопасность не единственная проблема	439
Функция exec() дает вам больший контроль	440
Существует много функций exec()	441
Функции с поддержкой массивов — execv(), execvp(), execve()	442
Передача переменных среды	443
Большинству системных вызовов присущи одни и те же проблемы	444
Читаем новости с помощью RSS	452
exec() — конечная остановка для вашей программы	456
Запуск дочернего процесса с помощью fork() + exec()	457
Ваш инструментарий языка Си	463



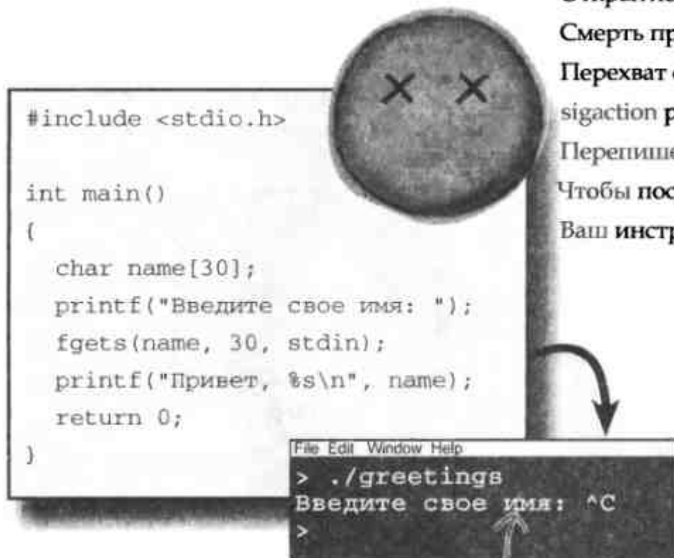
Межпроцессорное взаимодействие

10 Общение — это хорошо

Создание процессов — это только первый шаг.

Что если вам захочется *управлять* запущенным процессом? Что если вам захочется *отправить ему данные*? Или *прочитать его вывод*? Благодаря **межпроцессному взаимодействию** процессы могут *выполнять работу* сообща. Вы увидите, что **мощь** вашего кода может возрасти многократно, если позволить ему ***общаться*** с другими программами системы.

Перенаправление ввода и вывода	466
Типичный процесс изнутри	467
При перенаправлении всего лишь меняются потоки данных	468
С помощью fileno() можно получить дескриптор	469
Иногда нужно подождать	474
Поддерживайте связь со своим детищем	478
Соединяйте свои процессы с помощью каналов	479
Практический пример: открытие новостей в браузере	480
Дочерний процесс	481
Родительский процесс	481
Открытие веб-страницы в браузере	482
Смерть процесса	487
Перехват сигналов для запуска собственного кода	488
sigaction регистрируется с помощью функции sigaction()	489
Перепишем код с использованием обработчика сигнала	490
Чтобы посылать сигналы, используйте команду kill	493
Ваш инструментарий языка Си	502

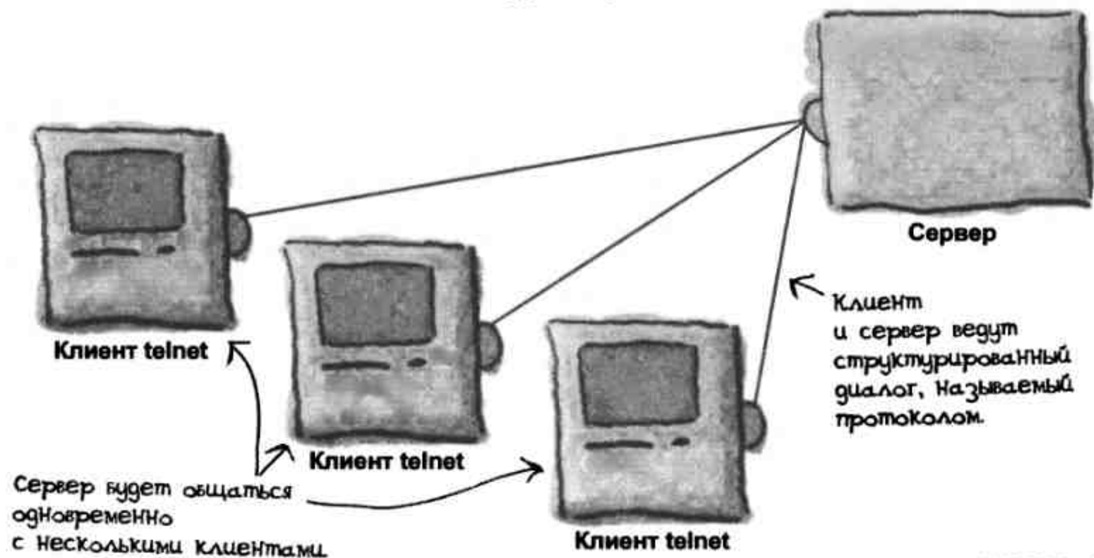


Сокеты и работа в Сети

11 Нет места лучше, чем 127.0.0.1

Программам на разных компьютерах тоже нужно общаться. Вы уже научились использовать ввод/вывод для организации взаимодействия с файлами и узнали, каким образом могут общаться два процесса на одном и том же компьютере. Теперь вы сможете *разговаривать со всем остальным миром*, создавая код на языке Си, способный взаимодействовать с другими программами **по Сети в любой точке *планеты***. К концу этой главы вы научитесь создавать как **серверные**, так и **клиентские** приложения.

Интернет-сервер «Тук-тук»	504
Обзор сервера «Тук-тук»	505
Как сервер разговаривает с Интернетом	506
Сокеты — это не совсем обычные потоки данных	508
Иногда сервер стартует не так, как положено	512
Говорила же мама всегда делать проверку на ошибки	513
Прием данных от клиента	514
Сервер может общаться только с одним человеком в отдельный момент времени	521
Вы можете клонировать процесс для каждого клиента	522
Написание веб-клиента	526
Клиент всегда прав	527
Создаем сокет для IP-адреса	528
getaddrinfo() получает адреса доменов	529
Ваш инструментарий языка Си	536



Потоки

12 Это параллельный мир

Программам часто приходится выполнять несколько заданий одновременно.

С помощью стандартных POSIX-потоков вы можете сделать код более отзывчивым, **разделяя его на несколько параллельно выполняемых частей**. Но будьте осторожны! Потоки являются мощным инструментом, однако вряд ли вы захотите, чтобы они столкнулись друг с другом. В этой главе вы научитесь расставлять светофоры и наносить **разметку** так, чтобы ***в вашем коде не образовывались пробки***. В результате вы узнаете, как создавать **POSIX-потоки** и как с помощью **механизмов синхронизации** ***сохранять целостность хрупких данных***.

Задачи выполняются последовательно… или нет… 538

…и разбиение на процессы не всегда подходит 539

Простые процессы выполняют только одну задачу за один раз 540

Наймите дополнительный персонал — используйте потоки 541

Как создавать потоки 542

Создаем поток с помощью pthread_create 543

Код небезопасен с точки зрения многопоточности 548

Вам нужно добавить светофоры 549

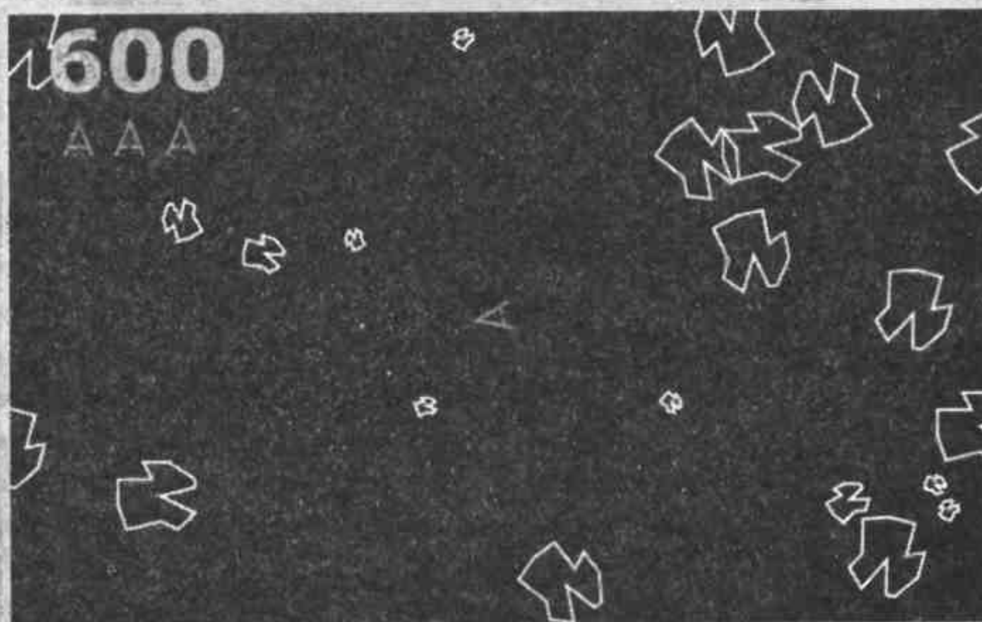
Используйте мьютексы как светофоры 550

Ваш инструментарий языка Си 557

Лабораторная работа 3

Бластероиды

В лабораторной работе 3 вы отдадите дань уважения одной из самых популярных и долговечных видеоигр. Пришло время написать «Бластероиды»!



Приложение I

На закуску

Топ-10 фактов (которым мы не уделили внимание)

Даже после всего сказанного еще кое-что осталось.

Есть несколько фактов, о которых, как нам кажется, вы должны знать. С одной стороны, нам не хотелось делать эту книгу объемной настолько, чтобы ее нельзя было поднять без специальной физической подготовки. Но с другой стороны, мы считаем, что кое-какие моменты заслуживают хотя бы краткого упоминания и проигнорировать их было бы неправильно. Поэтому прежде чем окончательно отложить книгу в сторону, **ознакомьтесь со следующими темами**.

№ 1. Операторы	576
№ 2. Директивы препроцессора	578
№ 3. Ключевое слово static	579
№ 4. Определение размера	580
№ 5. Автоматизированное тестирование	581
№ 6. Еще немного о gcc	582
№ 7. Еще немного о make	584
№ 8. Средства разработки	586
№ 9. Создание пользовательских интерфейсов	587
№ 10. Справочные материалы	588

Приложение II

Все темы

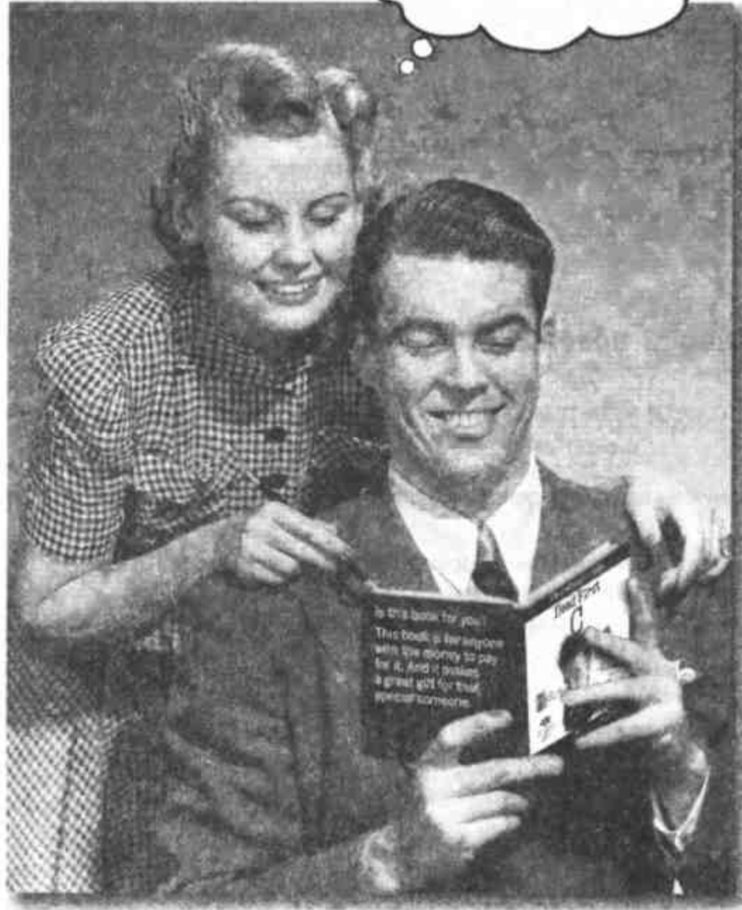
Вспомнить все

Всегда хотелось собрать все замечательные факты о Си в одном месте?

Здесь мы собрали все темы и правила, которые были затронуты в этой книге. Взгляните на них и постарайтесь вспомнить все, о чем мы вам рассказали. Для каждого факта указана глава, где он был рассмотрен, в случае чего вы можете запросто вернуться назад. Возможно, вы даже захотите вырвать эти страницы и приклеить их на стену.

Как пользоваться этой книгой

Введение



В этом разделе дается ответ на животрепещущий вопрос:
«Так почему же они ПОМЕСТИЛИ это в книгу о программировании на Си?»

Для кого эта книга

Если вы можете утвердительно ответить на все эти вопросы:

1. Вы уже знаете, как программировать на других языках?
2. Хотите ли вы овладеть Си, создать новую грандиозную программу, сколотить небольшое состояние и отойти от дел, чтобы коротать время на собственном острове?
3. Предпочитаете ли вы заниматься настоящим делом и применять на практике изученный материал, вместо того чтобы часами выслушивать лекции?

Ладно, это мы, наверное, загнули. Но нужно же с чего-то начинать, правда?

тогда эта книга для вас.

Кому лучше держаться подальше от этой книги

Если вы можете утвердительно ответить на любой из этих вопросов:

1. Вам нужен вводный курс или справочник по Си?
2. Вы предпочтете, чтобы 15 орущих обезьян вырывали вам ногти, лишь бы не изучать что-то новое? Вы считаете, что хороший учебник по языку Си должен быть всеобъемлющим и унылым до слез?

тогда эта книга *не* для вас.

(Примечание отдела по маркетингу: эта книга сгодится для всех, у кого есть кредитная карточка...)

Мы знаем, о чем подумали вы

«Как *это* может быть серьезной книгой о языке Си?»

«Что это за иллюстрации?»

«Могу ли я таким образом что-нибудь *выучить*?»

Мы знаем, о чем подумал ваш *мозг*

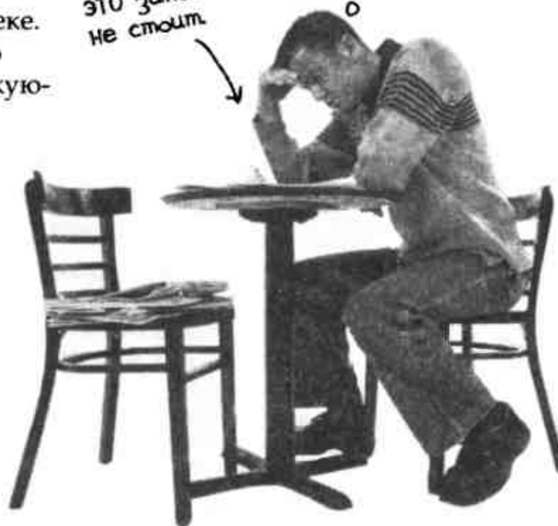
Ваш мозг жаждет чего-то нового. Он постоянно ищет, изучает, *ждет* чего-то необычного. Он так устроен, и это помогает вам в жизни.

Так как же поступает ваш мозг со всеми теми обычными, тривиальными, нормальными вещами, с которыми вы сталкиваетесь? Он делает все, чтобы они *не мешали* ему запоминать действительно *важную* информацию. Он не обращает внимания на скучные и очевидно бесполезные вещи, беспощадно их фильтруя.

Откуда же вашему мозгу *известно*, что является важным? Предположим, вы вышли прогуляться и тут на вас набросился тигр. Что будет твориться в вашей голове и с вашим телом?

Нейроны накалятся. Предельная концентрация. *Химический всплеск.*

Вот откуда ваш мозг знает…



Это важно! Не забудьте это!

Однако представьте, что вы находитесь дома или в библиотеке. Это безопасное и уютное место, совсем без тигров. Вы что-то изучаете. Готовитесь к экзамену. Или пытаетесь выучить какую-то сложную техническую тему за неделю, максимум за десять дней (чего, по мнению вашего начальника, должно быть достаточно).

Но есть одна проблема. Ваш мозг пытается оказать вам большую услугу. Он хочет убедиться, что вся эта *очевидно* ненужная информация не отнимет у вас столь дефицитные ресурсы. Ресурсы, которые лучше потратить на запоминание по-настоящему *важных* вещей. Таких как тигры. Или угроза для жизни. Или то, что вы никогда больше не должны размещать фотографии с той вечеринки на своей страничке в «Фейсбуке». И нет никакого простого способа отговорить ваш мозг от этого, сказав: «Эй, мозг, спасибо большое, но какой бы скучной ни была эта книга и как бы мало эмоций она у меня сейчас ни вызывала, я, *правда*, хочу запомнить то, что в ней написано».

В роли читателя данной книги мы бы хотели видеть человека, который хочет учиться.

Так что же нужно для того, чтобы что-нибудь *выучить*? Прежде всего вам необходимо *понять* прочитанное, а затем убедиться, что вы это не *забудете*. Речь идет не о зубрежке фактов. Согласно последним исследованиям в области когнитивистики, нейробиологии и педагогической психологии, *процесс обучения* — это нечто намного большее, чем просто чтение текста. Мы знаем, что заставляет ваш мозг работать.

Несколько принципов, на которых базируется эта книга.

Визуализация. Изображения запоминаются намного лучше, чем обычные слова, и делают обучение более эффективным (запоминание и восприимчивость улучшаются вплоть до 89 %). Благодаря этому информация становится более понятной. **При размещении слов внутри соответствующих иллюстраций или рядом с ними** вероятность возникновения у читателей проблем, связанных с содержимым книги, снижается почти *вдвое* (по сравнению с классической подачей материала, когда текст находится внизу или вовсе на другой странице).

Разговорный стиль и персонификация. Новейшие исследования показывают, что студенты справляются с итоговыми тестами почти на 40 % лучше, если в учебниках есть прямое обращение к читателю от первого лица с использованием разговорного стиля вместо официального тона. Рассказывайте истории, а не читайте лекции. Используйте неформальный язык. Не будьте слишком серьезными. Что бы *вас* больше заинтересовало — воодушевляющее общение на званом обеде или прослушивание лекции?

Попытка сделать так, чтобы читатель более глубоко вник в содержание. Иными словами, пока вы не начнете активно напрягать свои нейроны, в вашей голове ничего не родится. Читатель должен быть достаточно мотивирован, заинтересован, заинтригован и вдохновлен, чтобы решать проблемы, делать выводы и генерировать новые знания. А для этого ему нужны испытания, упражнения и вопросы, заставляющие задуматься; нужно сделать так, чтобы в процессе обучения участвовали обе доли головного мозга и разные уровни сознания.

Завоевание и удерживание внимания читателя. Все мы сталкивались с ситуацией, когда действительно нужно было что-то выучить, но после первой же страницы нас клонило в сон. Ваш мозг интересуется неординарными, интересными, странными, броскими и неожиданными вещами. Изучение нового сложного технического материала пройдет намного быстрее, если вам при этом не будет скучно.

Эмоции. Сейчас нам уже известно, что способность к запоминанию чего-либо сильно зависит от эмоциональной составляющей. Вы запоминаете то, что для вас важно. Вы запоминаете, когда что-то *чувствуете*. Нет, мы имеем в виду не душещипательные истории о мальчике и его собаке. Под эмоциями мы подразумеваем удивление, любопытство, веселье; когда мы думаем «Что за...» и чувствуем удовлетворение от того, что решили головоломку, выучили то, что остальные считают сложным, или понимаем, что мы знаем нечто такое, чего *не знает* наш заносчивый приятель, который якобы более подкован в технических вопросах.

Метапознание: размышления о мышлении

Если вы действительно хотите ускорить и углубить процесс обучения, обратите внимание на то, как вы обращаете внимание. Подумайте над тем, как вы думаете. Изучите то, каким образом вы учитесь.

В юном возрасте большинство из нас не оканчивали курсы метапознания и не знакомы с теорией обучения. Мы *планировали* учиться, а не «*учить*, как нужно учиться».

Но мы предполагаем, что если вы держите в руках эту книгу, то действительно хотите научиться программировать. И, вероятно, не желаете тратить на это слишком много времени. Если вы хотите применить на практике материал из этой книги, его необходимо *запоминать*. А для этого нужно *понимать* прочитанное. Чтобы получить от этой (или *любой* другой) книги максимальную пользу, возьмите на себя ответственность за свой мозг. За то, как он поступит с *этим* материалом.

Хитрость заключается в том, чтобы заставить ваш мозг думать о новом материале, как о чем-то *действительно важном*, как будто от этого зависит ваше благополучие. Словно перед вами тигр. В ином случае вы обречены на постоянную войну со своим мозгом в попытках заставить его запомнить новый материал.

Как же заставить свой мозг относиться к программированию так же серьезно, как к голодному тигру?

Можно пойти двумя путями: медленным и скучным или более быстрым и эффективным. Медленный путь — чистой воды зубрежка. Вам, конечно, известно, что упорство и труд перетрут даже самый неинтересный материал. При достаточном количестве повторений ваш мозг рано или поздно подумает: «Мне не *кажется*, что эта вещь для него важна, но раз он продолжает смотреть на нее *снова* и *снова*, то, наверное, что-то в ней есть».

Быстрый путь заключается в ***повышении мозговой активности***, в частности разных ее *типов*. На предыдущей странице мы уже описали большую часть подходов, которые призваны помочь мозгу работать в ваших интересах. Например, исследования показывают, что размещение слов *внутри* изображений, которые их иллюстрируют (а не в заголовке или в основном блоке текста), заставляет ваш мозг искать связь между текстом и картинкой, что приводит в действие больше нейронов. Больше нейронов — больше шансов на то, что ваш мозг сочтет потребляемую информацию стоящей и запомнит ее.

Помогает и разговорный стиль изложения. Люди имеют склонность больше сосредотачиваться, когда к ним кто-то обращается; они отслеживают мысль, чтобы поддержать беседу. Сложно поверить, но ваш мозг может *проигнорировать* тот факт, что «беседа» ведется между вами и книгой! С другой стороны, если книга выдержана в официальном и сухом стиле, ваш мозг будет вести себя так, словно вы слушаете лекцию в аудитории, наполненной сонными студентами. Нет нужды оставаться сосредоточенным.

Но иллюстрации и разговорный стиль — это только начало…

Вот что МЫ сделали

Мы использовали ***изображения***, потому что ваш мозг приспособлен к восприятию визуальной информации, а не текста. С его точки зрения, картинка *действительно* стоит тысячи слов. Мы поместили ключевой текст *внутрь иллюстраций*, на которые он ссылается, а не в заголовок или куда-либо еще, потому что так ваш мозг работает более эффективно.

Мы использовали ***повторения***, выражая одни и те же вещи *разными* способами, с помощью разного материала и в *разных смыслах*, увеличивая шансы на то, что информация будет закодирована сразу в нескольких участках вашего мозга.

Мы использовали понятия и изображения так, чтобы вас ***удивить***, потому что ваш мозг приспособлен к восприятию чего-то нового. Мы ввели *некую* ***эмоциональную*** *составляющую*, потому что ваш мозг восприимчив к биохимии эмоций. Благодаря этому вы *ощущаете*, что данную информацию лучше бы запомнить, даже если это ощущение основано всего лишь на ***юморе***, ***неожиданности*** или ***интересе***.

Мы использовали персонифицированный ***разговорный стиль*** изложения, так как ваш мозг больше сосредотачивается при общении, чем во время пассивного просмотра презентации. Такой подход срабатывает даже при *чтении*.

Мы добавили более 80 ***упражнений***, потому что ваш мозг может выучить и запомнить больше, если вы ***делаете*** какие-то вещи, а не просто *читаете* о них. Мы сделали упражнения сложными, но выполнимыми, потому что именно этого хотят большинство людей.

Мы использовали ***несколько стилей подачи материала***, ведь все люди разные. Одни могут отдавать предпочтение пошаговым методикам, другие захотят сначала понять общую картину, третьим нужно увидеть пример. Но, вне зависимости от ваших личных пристрастий в обучении, *каждый* читатель получит пользу от знакомства с материалом, представленным разными способами.

Мы подготовили материал для ***обеих половин вашего мозга***, потому что чем больше его участков вы задействуете, тем лучше пройдет обучение и запоминание и тем дольше вы сможете оставаться сконцентрированным. Поскольку нагрузка на одну половину мозга означает, что у другой есть шанс отдохнуть, ваши занятия станут более продуктивными и продолжительными.

Мы также включили в книгу ***рассказы*** и упражнения, которые предоставляют ***несколько разных точек зрения***, так как ваш мозг более глубоко вникает в проблему, если его заставить давать оценки и делать выводы.

Помимо упражнений мы добавили ***сложные задачи***, где задавали ***вопросы***, на которые не всегда можно ответить однозначно. Это связано с тем, что ваш мозг лучше обучается и запоминает при выполнении какой-то *работы* (то же самое с вашим *телом* — чтобы *поддерживать* его в хорошей форме, *недостаточно просто наблюдать* за посетителями фитнес-клуба). Но мы сделали все от нас зависящее, чтобы направить ваши тяжелые усилия в *правильное* русло и ***вы не потратили ни одного лишнего дендрита*** на работу с тяжелым для понимания примером или на разбор сложного, насыщенного терминами либо чрезмерно лаконичного текста.

В рассказах, примерах, иллюстрациях мы изображали *людей* потому… ну потому, что *вы сами* человек. И ваш мозг уделяет больше внимания именно *людям*, а не *предметам*.

Вот как ВЫ можете подчинить себе свой мозг

Итак, свою работу мы выполнили. Дальше все зависит от вас. Наши советы — всего лишь отправная точка. Прислушивайтесь к своему мозгу и сами определяйте, какие из этих пунктов вам подходят, а какие — нет. Можете попробовать что-нибудь новое.

1. **Не спешите. Чем больше вы поймете, тем меньше вам придется запоминать.**

   Недостаточно просто *читать*. Нужно останавливаться и обдумывать прочитанное. Когда в книге вам задается вопрос, не пропускайте его. Представьте, что кто-то действительно вас спрашивает. Чем более глубоко вы заставите свой мозг вникать в материал, тем больше у вас шансов что-нибудь выучить и запомнить.

2. **Выполняйте упражнения. Делайте заметки.**

   Мы включили их в книгу, но не будем делать их за вас — это как тренироваться за кого-то другого. Кроме того, недостаточно просто *смотреть* на них. **Пользуйтесь карандашом.** Существует множество свидетельств того, что физическая активность *во время* занятий способствует процессу обучения.

3. **Читайте разделы «Не бывает глупых вопросов».**

   Все без исключения. Это не какое-то дополнение — *это часть основного материала!* Не пропускайте их.

4. **Старайтесь не читать ничего после этой книги перед сном. По крайней мере, ничего, что требует умственного напряжения.**

   Процесс изучения (особенно сохранение информации в долговременной памяти) частично происходит *после* того, как вы откладываете книгу в сторону. Вашему мозгу нужно время, чтобы обработать прочитанное. Если в течение этого процесса вы попытаетесь запомнить что-то новое, часть изученного материала будет утрачена.

5. **Разговаривайте о прочитанном. Вслух.**

   При разговоре активируются разные части мозга. Если вы пытаетесь что-то понять или хотите с большей долей вероятности запомнить прочитанное, попробуйте произнести это вслух. А еще лучше, попытайтесь объяснить это на словах кому-нибудь другому. Процесс изучения ускорится, и при этом у вас могут появиться идеи, о которых вы не задумывались при чтении.

6. **Пейте воду. Много воды.**

   Лучше всего вашему мозгу работается во влажной среде. Обезвоживание (которое может наступить даже до того, как вы почувствуете жажду) снижает способность к познанию.

7. **Прислушивайтесь к своему мозгу.**

   Следите за тем, чтобы ваш мозг не был перегружен. Если вы начинаете поверхностно воспринимать или забывать только что прочитанный текст, значит пришло время сделать перерыв. Пройдя определенный рубеж, вы уже не сможете ускорить свое обучение, увеличивая объем материала. Более того, это может даже навредить.

8. **Почувствуйте что-нибудь.**

   Вашему мозгу нужно знать, что это *важно.* Пытайтесь прочувствовать рассказы, которые вы читаете. Делайте для фотографий свои собственные заголовки. Вздохнуть от плохой шутки *все же* лучше, чем совсем ничего не ощутить.

9. **Пишите много кода!**

   Есть только один способ изучить программирование на языке Си — **писать много кода.** И именно этим вы будете заниматься при чтении данной книги. Написание кода — это навык, и улучшить его можно только на практике.. Мы предоставим вам множество практических занятий: в каждой главе есть упражнения с реальными проблемами, которые вам нужно решить. Не пропускайте их — процесс обучения во многом строится на выполнении этих задач. Для каждого упражнения в данной книге есть решения — не бойтесь **ими пользоваться,** если у вас что-то не получается (очень легко застрять на какой-то мелочи)! Но сначала попытайтесь справиться самостоятельно. По крайней мере старайтесь не переходить к следующему разделу книги, пока не заставите пример работать.

Прочтите

Эта книга не просто справочник, вы должны по ней учиться. Мы специально убрали из нее все, что может помешать изучению подготовленного нами материала. Начинать следует с первых страниц, так как дальше информация подается исходя из того, что вы уже видели и выучили.

Мы исходим из того, что вы новичок в Си, но не в программировании.

Мы предполагаем, что вы уже имеете опыт в программировании. Пусть небольшой, но вы должны быть знакомы с такими вещами, как циклы и переменные в каких-нибудь других языках, например JavaScript. В действительности Си — довольно продвинутый язык, и если вам *совсем* не доводилось заниматься программированием, то мы советуем поискать для начала какую-нибудь другую книгу. Хорошим выбором будет *Head First Programming*.

Вам нужно установить на свой компьютер компилятор Си.

На протяжении всей книги мы будем использовать *GNU Compiler Collection* (`gcc`), потому что он бесплатен и чертовски хорош. Вам нужно убедиться, что `gcc` установлен на вашем компьютере. Если вы работаете в *Linux*, то он, скорее всего, у вас уже есть. Если вы пользователь Mac, вам нужно установить инструменты для разработки в составе Xcode. Вы можете загрузить их на сайте Apple или прямо из магазина приложений *App Store*.
У обладателей компьютеров под управлением Windows есть два варианта. *Cygwin (http://www.cygwin.com)*[1] представляет собой имитацию *UNIX*-среды, включая `gcc`. Но если вы хотите, чтобы процесс создания программ был простым и понятным, лучше загрузить *MinGW (http://www.mingw.org)* — минималистическую версию `gcc` для Windows .

Мы начнем со знакомства с базовыми концепциями языка Си, после чего научим вас пользоваться ими на практике.

В главе 1 мы рассмотрим основы языка Си. Таким образом, переходя к главе 2, вы уже будете иметь опыт создания настоящих рабочих, полезных и… хм… веселых приложений. Оставшуюся часть книги мы посвятим совершенствованию ваших навыков, незаметно превращая вас из *новичка* в *гуру программирования на Си*.

---

[1] Все указанные в книге сайты англоязычные. Издательство не несет никакой ответственности за их содержимое и напоминает, что со времени написания книги сайты могли измениться или вовсе исчезнуть. — *Примеч. ред.*

Практические упражнения являются ОБЯЗАТЕЛЬНЫМИ.

Практические задания — это не дополнительный материал, они являются частью основного содержания данной книги. Одни помогут вам запомнить прочитанное, другие важны для понимания, а третьи пригодятся для закрепления пройденного материала. ***Не пропускайте упражнения.***

Избыточность не случайна и важна.

Одна из отличительных особенностей этой книги заключается в том, что мы *действительно* хотели сделать ее понятной. И мы надеемся, что после прочтения вы будете помнить все, что изучили. Содержимое большинства справочников не рассчитано на то, чтобы оставаться в памяти надолго, но данная книга является *учебником*, поэтому одни и те же понятия будут встречаться несколько раз.

Примеры максимально упрощены.

Плохо, когда приходится пересматривать 200 строчек кода ради тех двух, которые нужны для понимания происходящего. Большинство примеров в книге представлены в максимально урезанном виде, чтобы изучаемая часть была простой и понятной. Так что не стоит ждать от кода целостности или даже завершенности. Этими свойствами должен обладать *ваш* код после того, как вы дочитаете книгу.

Упражнения «Сила мозга» не имеют решения.

Для некоторых из них просто нет правильного ответа, другие являются частью процесса обучения, и только вам решать, справились ли вы с ними. Иногда вы будете встречать подсказки, указывающие верное направление.

Команда технических рецензентов

Дейв Китабян

Винс Милнер

Технические рецензенты:

Дейв Китабян (Dave Kitabjian) обладает двумя учеными степенями в областях электротехники и компьютерной инженерии, имеет двадцатилетний опыт консалтинга, интегрирования, проектирования и построения информационных систем для совершенно разных клиентов — от крупных фирм до высокотехнологичных новых компаний. Вне работы Дейв любит играть на гитаре и фортепьяно и проводить время со своей женой и тремя детьми.

Винс Милнер (Vince Milner) более двадцати лет занимается программированием на Си (и на многих других языках) для широкого спектра платформ. В перерывах между получением степени магистра математики он раз за разом терпит поражения от своего шестилетнего сына в настольных играх и безуспешно пытается переехать в новый дом.

Благодарности

Нашему редактору

Огромное спасибо **Брайану Сойеру (Brian Sawyer)**, именно он попросил нас написать эту книгу. Брайан верил в нас от начала и до конца, предоставил нам возможность опробовать новые идеи и не слишком паниковал, когда сроки начали поджимать.

Команде издательства O'Reilly

Выражаем огромную благодарность следующим людям, которые помогали нам все это время: **Карен Шейнер (Karen Shaner)** за ее искусные навыки в поиске иллюстраций и за то, что держала все на плаву; **Лори Петрицки (Laurie Petrycki)** за то, что кормила и подбадривала нас во время нашего пребывания в Бостоне; **Брайану Джепсону (Brian Jepson)** за то, что открыл перед нами чудесный мир Arduino, и **команде по предварительной работе над книгой** за то, что ранние версии нынешнего издания были доступны в Сети. И наконец, спасибо **Рэйчел Монаган (Rachel Monaghan)** и технологическому отделу, которые отвечали за процесс выпуска данной книги и чья тяжелая работа осталась за кулисами. Ребята, вы просто потрясающие!

Семье, друзьям и коллегам

В процессе работы над книгами из цикла *Head First* мы обрели множество друзей. Отдельное спасибо Лу **Барру (Lou Barr)**, **Бретту Маклафлину (Brett McLaughlin)** и **Сандерсу Клейнфельду (Sanders Kleinfeld)** за то, что многому нас научили.

Дэвид: Благодарю **Энди Паркера (Andy Parker)**, **Джо Бротона (Joe Broughton)**, **Карла Жака (Carl Jacques)**, **Саймона Джонса (Simon Jones)** и других своих друзей, которым я уделял так мало внимания, пока писал книги.

Дон: Мне было бы намного труднее работать над этой книгой, не будь поддержки моих родных и друзей. Отдельное спасибо **маме** и **папе**, **Карлу**, **Стиву**, **Джилл**, **Джеки**, **Джойс** и **Полу**. Я искренне признательна вам за то, что вы помогали мне и подбадривали меня.

Тем, без кого ничего бы не получилось

Наша команда технических экспертов проделала воистину замечательную работу, уберегая нас от возможных ошибок и отслеживая правильность всего написанного. Мы также несказанно благодарны всем тем, кто присылал отзывы о ранних версиях этого издания. Мы считаем, что благодаря вам книга стала гораздо лучше.

И наконец, благодарим **Кэти Сьерра (Kathy Sierra)** и **Берта Бейтса (Bert Bates)** за создание этой выдающейся серии книг.

1 Начинаем работать с языком Си

Погружаемся

Хотите узнать, как думает компьютер?

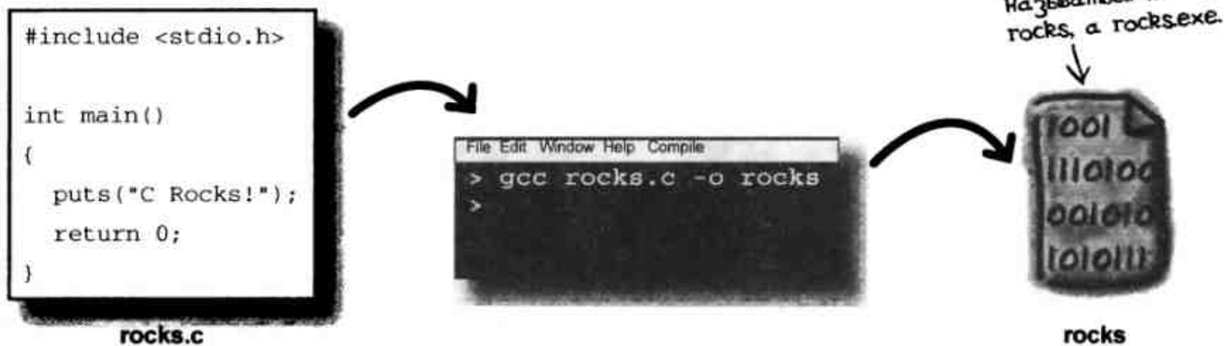
Вам нужно написать **высокопроизводительный код** для новой игры? Запрограммировать контроллер **Arduino**? Или использовать продвинутую **стороннюю библиотеку** в своем приложении для iPhone? Тогда язык Си вам поможет. Си работает на **гораздо более низком уровне**, чем большинство других языков программирования, поэтому понимание Си позволит вам лучше разобраться в том, **что на самом деле происходит внутри компьютера**. Си также может помочь в освоении других языков. Поэтому хватайте свой компилятор и немедленно приступайте к делу.

Си — это язык для компактных быстрых программ

Язык Си предназначен для написания компактных быстрых программ. Он более низкоуровневый, чем большинство других языков программирования. Это означает, что *созданный с его помощью код наиболее понятен для компьютера.*

Как работает Си

В действительности компьютеры понимают только один язык — машинный код — бинарный поток единичек и нулей. Ваш код на Си преобразуется в машинный с помощью **компилятора**.



1 Исходник

Все начинается с создания исходного файла. Исходный файл содержит код на языке Си, пригодный для восприятия человеком.

2 Компиляция

Исходный код пропускается через компилятор. Компилятор ищет ошибки и, если все хорошо, компилирует исходный код.

3 Вывод

Компилятор создает новый файл, который называется *исполняемым*. Этот файл содержит машинный код — поток единиц и нулей, понятный компьютеру. Это и есть программа, которую вы запускаете.

Си используется там, где важны скорость и объем. Большинство операционных систем написано на Си. Многие компьютерные языки также написаны на Си, равно как и большая часть игровых программ.

Вы можете столкнуться с тремя стандартами языка Си. ANSI C был создан в конце 1980-х гг. и используется для наиболее старого кода. В 1999 г. многие аспекты языка были исправлены в стандарте C99. А в 2011 г. появился современный стандарт C11 с некоторыми новыми интересными возможностями. Отличия между разными версиями не очень большие, и мы будем указывать на них время от времени.

Наточите свой карандаш

Попытайтесь угадать, что делает каждый из этих фрагментов.

Опишите действия, которые, по вашему мнению, выполняет этот код

```
int card_count = 11;
if (card_count > 10)
    puts("Карты в колоде удачные. Повышаем ставку.");
```

```
int c = 10;
while (c > 0) {
    puts("Я не должен писать код в виде
классов");
    c = c - 1;
}
```

```
/* Подразумевается, что имя короче 20 латинских (и 10
кириллических) символов. */
char ex[20];
puts("Введите имя вашей девушки:");
scanf("%19s", ex);
printf("Дорогая %s. \n\n\tC тобой покончено.\n", ex);
```

```
char suit = 'H';
switch(suit) {
case 'C':
    puts("Clubs(Трефы)");
    break;
case 'D':
    puts("Diamonds(Бубны)");
    break;
case 'H':
    puts("Hearts(Черви)");
    break;
default:
    puts("Spades(Пики)");
}
```

Наточите свой карандаш
Решение

Не волнуйтесь, если вам что-то непонятно. Далее в этой *книге будут даны более подробные пояснения.*

```
int card_count = 11;
if (card_count > 10)
    puts("Карты в колоде удачные. Повышаем ставку.");
```

int – целое число

Здесь текст выводится в командной строке или на терминал.

Создаем целочисленную переменную и устанавливаем для нее значение 11.

Счетчик больше 10?

Если да, то выводим сообщение в командной строке.

```
int c = 10;
while (c > 0) {
    puts("Я не должен писать код в виде классов");
    c = c - 1;
}
```

Фигурные скобки определяют блочную инструкцию.

Создаем целочисленную переменную и устанавливаем для нее значение 10.

До тех пор пока значение больше 0,...

...выводим сообщение...

...и уменьшаем счетчик.

Конец кода, который должен повторяться.

```
/* Подразумевается, что имя короче 20 латинских (и 10 кириллических) символов. */
char ex[20];
puts("Введите имя вашей девушки: ");
scanf("%19s", ex);
printf("Дорогая %s. \n\n\tС тобой покончено.\n", ex);
```

Это означает «поместить все, что ввел пользователь, внутрь массива ex».

Здесь строка символов вставляется на место %s.

Это комментарий.

Создаем массив из 20 символов.

Выводим сообщение на экран.

Сохраняем в массив то, что ввел пользователь.

Выводим сообщение, включающее введенный текст.

```
char suit = 'H';
switch(suit) {
case 'C':
    puts("Clubs(Трефы)");
    break;
case 'D':
    puts("Diamonds(Бубны)");
    break;
case 'H':
    puts("Hearts(Черви)");
    break;
default:
    puts("Spades(Пики)");
}
```

Инструкция switch проверяет одну и ту же переменную на разные значения.

Создаем символьную переменную, помещаем в нее букву «H».

Проверяем значение переменной.

Оно равно «C»?

Если да, то выводим «Clubs (Трефы)».

Затем пропускаем остальные проверки.

Оно равно «D»?

Если да, то выводим «Diamonds (Бубны)».

Затем пропускаем остальные проверки.

Оно равно «H»?

Если да, то выводим «Hearts (Черви)».

Затем пропускаем остальные проверки.

В противном случае...

...выводим «Spades (Пики)».

На этом проверки заканчиваются.

Как выглядит программа на Си целиком

Чтобы создать полноценную программу, вам необходимо написать свой код на языке Си и сохранить его в *исходный файл*. Исходные файлы можно создавать с помощью любого текстового редактора, имя файла при этом, как правило, заканчивается на *.c*.

← Это всего лишь условность, но вы должны ее соблюдать.

Давайте взглянем на типичный исходный файл на языке Си.

1 Программы на Си обычно начинаются с комментария. Комментарий описывает назначение кода, содержащегося в файле, и, возможно, содержит информацию о лицензии и авторском праве. Острой необходимости добавлять его сюда (или в любую другую часть файла) нет, но так принято, и большинство программистов на Си рассчитывают подобный комментарий там увидеть.

Комментарий начинается с символов /*.

Эти звездочки необязательны и нужны только для красоты.

Комментарий заканчивается символами */.

```
/*
 * Программа для подсчета количества карт в колоде.
 * Этот код выпущен под публичной Лас-Вегасской лицензией.
 * (c)2014, Команда колледжа по блек-джеку.
 */
#include <stdio.h>

int main()
{
    int decks;
    puts("Введите количество колод.");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("Вы ввели недопустимое количество колод.");
        return 1;
    }
    printf("Всего карт %i\n", (decks * 52));
    return 0;
}
```

2 Дальше следует часть с подключением внешних файлов. Си — очень лаконичный язык, и без использования *внешних библиотек* он практически ни на что не способен. Компилятору необходимо сообщить, какой внешний код использовать, подключая заголовочные файлы соответствующих библиотек. Чаще других будет встречаться заголовок *stdio.h*. Это библиотека `stdio`, содержащая код для вывода данных на терминал и считывания их оттуда.

3 Последнее, что вы найдете в исходном файле, — это функции. Весь код на Си выполняется внутри функций. Самая важная функция в любой программе, написанной на Си, называется **главной — `main()`**. Она является отправной точкой для всего кода в вашей программе.

Давайте рассмотрим функцию main() более подробно.

Подробнее о функции main()

Компилятор начнет выполнять вашу программу с функции main(). Если у вас нет функции с таким именем, ваша программа не сможет запуститься.

Возвращаемый тип главной функции — int. Что бы это значило? Когда компьютер работает с вашей программой, ему нужно как-то определить, успешно она выполнилась или нет. Для этого проверяется *значение, возвращаемое главной функцией*. Если функция main() возвращает 0, это означает, что программа завершилась успешно. Любое другое значение говорит о том, что возникла проблема.

```
int main()
{
    int decks;
    puts("Введите количество колод.");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("Вы ввели недопустимое количество колод.");
        return 1;
    }
    printf("Всего карт %i\n ", (decks * 52));
    return 0;
}
```

Это возвращаемый тип. Для главной функции он всегда должен быть int.

Поскольку функция называется main (главная), программа начнется с нее.

Если бы у нас были какие-нибудь параметры, они бы упоминались здесь.

Тело функции всегда заключено в фигурные скобки.

Имя функции идет вслед за возвращаемым типом. После этого указываются параметры функции, если таковые имеются. В конце следует *тело функции*, которое **обязательно** заключается в *фигурные скобки*.

Уголок ботана

Функция printf() используется для отображения **форматированного вывода.** Вместо символов-спецификаторов она подставляет значения переменных, например:

```
printf("%s говорит, что результат равен %i","Вен", 21);
```

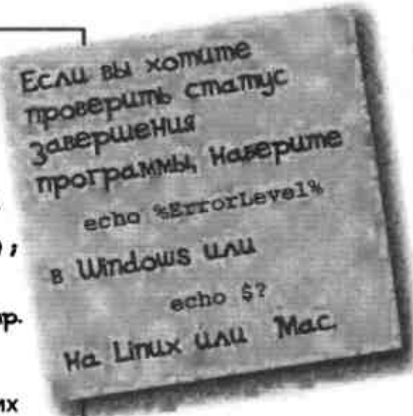
Первый параметр будет вставлен сюда в виде строки.

Первый параметр.

Второй параметр будет вставлен сюда в виде целого числа.

Второй параметр.

При вызове функции printf() вы можете использовать столько параметров, сколько пожелаете, только убедитесь, что для каждого из них есть соответствующий символ-спецификатор %.



Магнитики с кодом

Команда колледжа по блек-джеку с помощью магнитиков составила какой-то код прямо на двери холодильника в общежитии. Однако кто-то перемешал все магнитики! Можете ли вы составить код заново?

```
/*
 * Программа для оценивания важности карт.
 * Выпущено под публичной Лас-Вегасской лицензией.
 * (c)2014 Команда колледжа по блек-джеку.
 */




.................. main()
{
    char card_name[3];
    puts("Введите название карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {



    } else if (card_name[0] ==..................) {
        val = 10;

    }........................(card_name[0] == ..........) {

     ..................................................................
    } else {
        val = atoi(card_name);
    }
    printf("Ценность карты: %i\n", val);

  ......................0;
}
```

В качестве названия карты вводим два символа. → scanf

Преобразовывает текст в число → atoi

Магнитики:

- <stdlib.h>
- ;
- ;
- val = 11
- int
- 'J'
- #include
- 'A'
- return
- else
- #include
- if
- val = 10
- <stdio.h>

Магнитики с кодом. Решение

Команда колледжа по блек-джеку с помощью магнитиков составила какой-то код прямо на двери холодильника в общежитии. Однако кто-то перемешал все магнитики! Вам нужно было составить код заново.

```
/*
 * Программа для оценивания важности карт.
 * Выпущено под публичной Лас-Вегасской лицензией.
 * (c)2014 Команда колледжа по блек-джеку.
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("Введите название карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("Ценность карты: %i\n", val);
    return 0;
}
```

не бывает Глупых Вопросов

В: Что означает `card_name[0]`?

О: Это первый символ, который набрал пользователь. Таким образом, если он ввел 10, `card_name[0]` будет равен 1.

В: Вы всегда пишете комментарии с помощью /* и */?

О: Если ваш компилятор поддерживает стандарт C99, то вы можете начинать комментарий с //. Компилятор воспримет оставшуюся часть такой строчки как комментарий.

В: Как узнать, какой стандарт поддерживает мой компилятор?

О: Проверьте документацию для компилятора. gcc поддерживает все три стандарта: ANSI C, C99 и C11.

Но как мы запустим программу?

Си — *компилируемый язык*. Это значит, что компьютер не будет интерпретировать код напрямую. Вместо этого вам нужно конвертировать (или *компилировать*) исходный код, понятный человеку, в *машинный код*, который может понять компьютер.

Для компиляции кода понадобится программа, которую называют **компилятором**. Одним из самых распространенных компиляторов для языка Си является **gcc** (*GNU Compiler Collection* — набор компиляторов в рамках проекта GNU). gcc доступен на множестве операционных систем и помимо Си способен компилировать большое количество других языков. И главное, он абсолютно бесплатный.

Вот как можно скомпилировать и запустить программу с помощью gcc.

1. **Сохраните код, представленный на противоположной странице в упражнении с магнитиками, в файл под названием cards.c.**

Исходные файлы на языке Си обычно заканчиваются на .c.

2. **Выполните компиляцию, набрав `gcc cards.c -o cards` в командной строке или на терминале.**

```
File Edit Window Help Compile
> gcc cards.c -o cards
>
```

Компилируем cards.c в файл под названием cards

Он будет называться cards.exe, если вы используете Windows

3. **Запустите, набрав `cards`, если вы работаете с Windows, или `./cards`, если используете Mac либо Linux.**

```
File Edit Window Help Compile
> ./cards
Введите название карты:
```

Уголок ботана

Вы можете скомпилировать и запустить свой код за один шаг:

&& здесь означает «и, если все прошло успешно, выполнить это...»

```
gcc zork.c -o zork && ./zork
```

Если вы работаете в Windows, нужно заменить ./zork на zork

Эта команда запустит новую программу только в том случае, если та успешно скомпилировалась. Если при компиляции возникли проблемы, запуск программы будет пропущен и на экран будут выведены ошибки.

Сделайте это!

Сейчас вы должны создать файл *cards.c* и скомпилировать его. С каждой новой главой мы будем работать над этим все больше и больше.

Тест-драйв

Давайте посмотрим, сможет ли программа скомпилироваться и запуститься. Откройте командную строку или терминал на своем компьютере и попробуйте.

Эта строчка компилирует код и запускает карточную программу.

Эта строчка запускает программу. Если вы используете Windows, не набирайте ./.

Повторный запуск программы.

Пользователь вводит название карты...

...и программа выводит соответствующее значение.

```
File Edit Window Help 21
> gcc cards.c -o cards
> ./cards
Введите название карты:
Q
Ценность карты: 10
> ./cards
Введите название карты:
A
Ценность карты: 11
> ./cards
Введите название карты:
7
Ценность карты: 7
```

Помните: вы можете совместить этап компиляции и запуска (вернитесь на предыдущую страницу, чтобы посмотреть, как это сделать).

Программа работает!

Поздравляем! Вы скомпилировали и запустили программу на Си. Компилятор gcc взял понятный для человека исходный код из файла *cards.c* и преобразовал его в *машинный код* карточной программы. Если вы используете Mac или Linux, компилятор поместит исходный код в файл под названием *cards*. Однако для Windows все программы должны иметь расширение *.exe*, поэтому файл получит название *cards.exe*.

не бывает
Глупых Вопросов

В: Почему при запуске программы на Linux и Mac я должен добавлять ./ перед ее названием?

О: В Unix-подобных операционных системах программа запускается только в том случае, если указана директория, в которой она находится, или если эта директория числится в переменной окружения PATH.

Постойте, тут что-то не так. Когда мы спрашивали у пользователя название карты, мы использовали массив символов. **Массив символов**? Зачем? Разве мы не могли использовать **строку** или что-то в этом роде?

Язык Си изначально не имеет поддержки строк.

Но существуют библиотеки расширений для Си, которые предоставляют вам строки.

Си — более низкоуровневый, чем большинство других языков, поэтому вместо строк в Си обычно используют нечто похожее — *массив отдельных символов*. Если вы программировали на других языках, вам, вероятно, уже знакомы массивы. Массив — это просто набор элементов под общим именем. Таким образом, `card_name` — это просто имя переменной, которое мы используем, чтобы обозначить список символов, введенных в командной строке. Мы объявили массив `card_name` так, чтобы он вмещал *два символа*, поэтому мы можем ссылаться на первый и второй символ как на `card_name[0]` и `card_name[1]` соответственно. Чтобы понять, как это работает, давайте подробно рассмотрим компьютерную память и то, как Си оперирует текстом…

Подробнее о строках

Строки — это просто массивы символов. Когда Си видит строку наподобие этой:

```
s = "Shatner"
```

Он считывает ее так, как будто это массив отдельных символов:

Вот как объявляется массив в Си.

```
s = {'S', 'h', 'a', 't', 'n', 'e', 'r'}
```

Каждый символ в строке является просто элементом массива, в связи с чем вы можете ссылаться на конкретные символы в строке с помощью индекса. Например, s[0] и s[1].

S	h	a	...
s[0]	s[1]	s[2]	

Не забывайте о конце строки

Но что происходит, когда Си хочет прочитать содержимое строки, например когда нужно вывести ее на экран? Сейчас во многих языках компьютер сам отслеживает размер массива, но Си работает на более низком уровне, чем большинство других языков, и он не всегда может точно определить, *какова длина* массива. Если Си собирается вывести на экран строку, он должен знать, в каком месте заканчивается массив символов. Это делается с помощью **нуль-символа**.

Си знает, что при виде \0 нужно остановиться.

Нуль-символ добавляется в конце строки и имеет значение \0. Всякий раз, когда компьютеру необходимо прочитать содержимое строки, он по очереди перебирает элементы символьного массива, пока ему не попадется \0. Это означает, что, когда компьютер видит строку:

```
s = "Shatner"
```

он действительно помещает ее в память в таком виде:

S	h	a	t	n	e	r	\0
s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]

\0 — это символ, чей ASCII-код равен 0.

Программисты на языке Си часто называют его нуль-символом.

Вот почему мы должны были объявить card_name в нашем коде таким образом:

```
char card_name[3];
```

В строку card_name может быть записан один или два символа, но строка должна заканчиваться *нуль-символом*, поэтому нам нужно оставить в массиве место для дополнительного элемента.

не бывает Глупых Вопросов

В: Почему нумерация символов начинается с нуля? Почему не с единицы?

О: Индекс — это смещение. Он показывает, насколько далеко находится текущий символ от первого.

В: Зачем?

О: Компьютер разместит символы в памяти в последовательных байтах. Он может использовать индекс, чтобы вычислять местоположение конкретного символа. Зная, что `c[0]` находится в ячейке памяти 1 000 000, он может быстро вычислить, что `c[96]` размещен по адресу 1 000 000 + 96.

В: Зачем ему нужен нуль-символ? Разве он не знает, какая длина у строки?

О: Как правило, не знает. Си не очень хорош в отслеживании размера массивов, а строка — это всего лишь массив.

В: Он не знает, какая длина у массива?

О: Нет. Иногда компьютер может определить длину массива, анализируя код, но в Си, как правило, вы сами должны следить за своими массивами.

В: Есть ли какая-то разница между одинарными и двойными кавычками?

О: Да. Отдельные символы заключаются в одинарные кавычки, тогда как для строк всегда используются двойные.

В: Так как я должен объявлять свои строки: с помощью кавычек (") или явно, в виде массива символов?

О: В большинстве случаев вы будете объявлять строки с помощью кавычек. Это так называемые **строковые литералы**, их легче всего набирать.

В: Есть ли какая-то разница между строковыми литералами и массивами символов?

О: Только одна — строковые литералы являются константами.

В: Что это означает?

О: Это означает, что вы не можете изменять отдельные символы после того, как они были созданы.

В: Что случится, если я попытаюсь?

О: Это зависит от компилятора, но `gcc` обычно выводит ошибку bus error.

В: Что это еще за ошибка такая?

О: Си будет по-разному хранить строки в памяти. Ошибка bus error всего лишь означает, что этот участок памяти не может быть изменен вашей программой.

Безболезненные операции

Не все знаки равенства равны.

В языке Си знак равенства (=) используется для присваивания. Но два таких знака (==) означают проверку равенства.

Присваивает teeth значение 4. →

```
teeth = 4;

teeth == 4;
```

↑ Проверяет, имеет ли teeth значение 4.

Если вы хотите увеличить или уменьшить переменную, то можете сэкономить место, используя присваивания вида += и -=.

Прибавляет 2 к teeth

```
teeth += 2;

teeth -= 2;
```

↑ Вычитает 2 из teeth

Наконец, если вы хотите увеличить или уменьшить переменную на 1, используйте ++ и --.

```
teeth++;

teeth--;
```

← Увеличивает на 1

← Уменьшает на 1

Два вида команд

Пока что все команды, которые нам встречались, попадали в одну из двух категорий.

Делать что-то

Большинство команд в Си являются инструкциями. Простые инструкции — это *действия*; они что-то *выполняют* и что-то *сообщают*. Нам встречались инструкции, которые объявляют переменные, считывают клавиатурный ввод или выводят данные на экран.

```
split_hand();   ← Это простая инструкция.
```

Иногда несколько простых инструкций группируют в *блочные*. Блочные инструкции — это совокупность команд, заключенная в фигурные скобки.

```
{
    deal_first_card();
    deal_second_card();
    cards_in_hand = 2;
}
```

Эти команды формируют блочную инструкцию, так как они заключены в фигурные скобки.

Делать что-то, только если нечто является истинным

Управляющие инструкции, такие как `if`, проверяют условие, прежде чем выполнить код:

```
if (value_of_hand <= 16)   ← Это условие.
    hit();     ← Выполнить эту инструкцию, если условие соблюдается.
else
    stand();   ← Выполнить эту инструкцию, если условие не соблюдается.
```

Если при соблюдении условия нужно выполнить сразу несколько действий, используется блочная инструкция:

```
if (dealer_card == 6) {
    double_down();
    hit();
}
```

Если условие соблюдается, ОБЕ эти команды будут выполнены. Команды сгруппированы внутри одной блочной инструкции.

Нужны ли вам скобки?

Вы можете работать с блочными инструкциями, как с *обычными*, выполняя при этом *целый набор команд*. В Си условие `if` работает следующим образом:

```
if (countdown == 0)
  do_this_thing();
```

В условии `if` выполняется **одна инструкция**. А если вы хотите, чтобы их там было несколько? Заключив список инструкций в фигурные скобки, вы позволите Си обращаться с ними, как с одной единственной командой:

```
if (x == 2) {
  call_whitehouse();
  sell_oil();
  x = 0;
}
```

Программисты на Си предпочитают писать лаконичный и элегантный код, опуская скобки в условиях `if` и циклах `while`. Поэтому вместо

```
if (x == 2) {
    puts("Делать что-то");
}
```

большинство из них напишут:

```
if (x == 2)
    puts("Делать что-то");
```

Код, который мы имеем в настоящий момент

```
/*
 * Программа для оценивания важности карт.
 * Выпущено под публичной Лас-Вегасской
   лицензией.
 * (c)2014 Команда колледжа по блек-джеку.
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("Введите название_карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("Ценность карты: %i\n", val);
    return 0;
}
```

Я тут подумал.
Можно ли сюда добавить проверку значения карты на вхождение в определенный диапазон?
Это может быть полезно...

Сенсационный материал

Я МОГУ СДЕЛАТЬ ВАС ТАКИМ ЖЕ БОГАТЫМ, КАК Я САМ!

Заочные курсы по блек-джеку от Эдди Рича

Эй, как дела? Мне сдается, ты смышленый парень. *Уж я*-то в этом разбираюсь! Послушай, я кое-что тебе расскажу, ведь Эдди — хороший парень, поэтому Эдди введет тебя в курс дела. Видишь ли, я эксперт в подсчете карт. Самый крутой из всех. Ты спрашиваешь, что за подсчет карт? Как по мне, так это самое стоящее дело!

Серьезно, подсчет карт — верный способ повысить шансы на выигрыш при игре в блек-джек. Если в колоде осталось много ценных карт, значит перевес на стороне игрока, то есть на твоей!

Подсчет карт помогает отслеживать оставшееся количество ценных карт. Скажем, ты начинаешь считать с нуля. Затем сдающий карты вытаскивает даму — это ценная карта. Значит, в колоде стало одной ценной картой меньше, поэтому ты уменьшаешь счетчик на единицу:

Это дама ==>count - 1

Но если это дешевая карта, такая как четверка, *счетчик* на единицу увеличивается:

Это четверка ==>count + 1

Ценные карты — это десятки и фигурные карты (валет, дама, король). Дешевые карты — тройки, четверки, пятерки и шестерки.

Поступай так с каждой ценной и дешевой картой, пока значение счетчика не станет действительно высоким. Тогда ты выкладываешь свои деньги на следующей ставке — и вуаля! Скоро у тебя будет больше денег, чем у моей третьей жены!

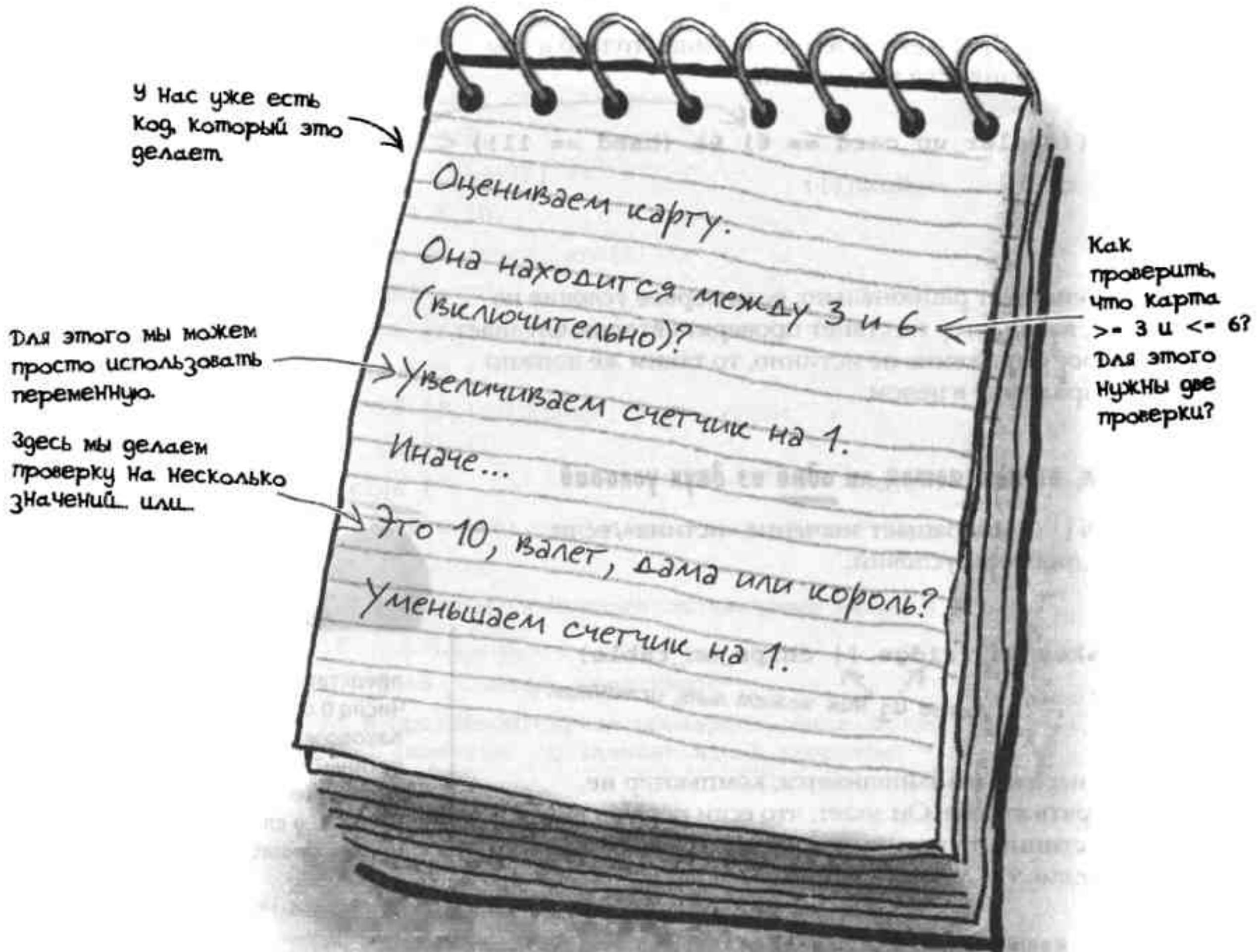
Если хочешь узнать подробности, записывайся на мои заочные курсы по блек-джеку сегодня же. Изучи досконально подсчет карт, а также:

- как использовать критерий *Келли*, чтобы максимально повысить стоимость своей ставки;
- как не получить нагоняй от распорядителя казино;
- как вывести пятна от пирожных на костюме из шелка;
- с чем лучше всего сочетать клетчатый узор.

За подробностями обращайся к кузену Винни через заочные курсы по блек-джеку.

Подсчет карт? В Си?

Подсчет карт — это способ повысить ваши шансы на победу в блек-джеке. Считая карты по мере их сдачи, игрок может выбирать наиболее удачные моменты для высоких и маленьких ставок. Это эффективный, но в то же время довольно простой прием.



Насколько сложно будет написать это на Си? Мы уже видели, как делать одиночную проверку, но алгоритм подсчета карт требует проверять несколько условий: нам нужно понять, что число >= 3 и в то же время <= 6.

Нам нужен набор операций, который объединит условия проверки.

Кроме равенства есть и другие логические выражения

Пока что мы рассмотрели инструкции `if`, которые проверяют истинность одиночного выражения. Но что если нужно проверить сразу несколько условий? Или узнать, что одно из условий *не* выполняется?

&& проверяет выполнение двух условий

Оператор *и* (`&&`) возвращает значение «истина» только в том случае, когда выполняются **оба** условия.

```
if ((dealer_up_card == 6) && (hand == 11))
    double_down();
```

Чтобы данный участок кода выполнился, оба эти выражения должны быть истинными

Оператор *и* действует рационально: если первое условие не выполняется, компьютер не станет проверять второе. Он знает, что если первое выражение не истинно, то таким же должно быть и все выражение в целом.

|| проверяет, выполняется ли одно из двух условий

Оператор *или* (`||`) возвращает значение «истина», если выполняется **любое** из условий.

```
if (cupcakes_in_fridge || chips_on_table)
    eat_food();
```

Любое из них может быть истинным.

Если первое выражение выполняется, компьютер не станет проверять второе. Он знает, что если первое выражение истинно, то истинным должно быть и *все выражение в целом*.

! «переворачивает» значение условия

! — это оператор *не*. Он меняет значение условия на противоположное.

```
if (!brad_on_phone)
    answer_phone();
```

! означает «не»

Уголок ботана

В Си логические выражения представлены в виде чисел. Число 0 обозначает «ложь». Каково же тогда значение для истины? Все, что не равно нулю, воспринимается как истина. Поэтому в следующем коде нет никакой ошибки:

```
int people_moshing = 34;
if (people_moshing)
    take_off_glasses();
```

Фактически программы на Си часто пользуются этим, чтобы быстро проверить, не равно ли какое-то значение нулю.

Упражнение

Вам необходимо изменить программу таким образом, чтобы ее можно было использовать для подсчета карт. Если значение карты находится между 3 и 6, она должна вывести одно сообщение, а если это 10, валет, дама или король — другое.

```
int main()
{
    char card_name[3];
    puts("Введите название карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Проверяем, находится ли значение между 3 и 6 */
    if ...............................................................
        puts("Счетчик увеличился");
    /* В противном случае проверяем, была ли эта карта 10,
       J (валетом), Q (дамой) или K (королем) */
    else if ..........................................................
        puts("Счетчик уменьшился");
    return 0;
}
```

Деликатный путеводитель по стандартам

Стандарт ANSIC для Си не предусматривает значений для «истина» и «ложь». Программы на Си воспринимают 0 как «ложь», а любое другое значение — как «истина». Стандарт C99 позволяет вам использовать в своих программах слова *true* (истина) и *false* (ложь), но компилятор в любом случае будет воспринимать их как 1 и 0.

Упражнение Решение

Вы должны были изменить программу таким образом, чтобы ее можно было использовать для подсчета карт. Программа должна выводить одно сообщение, если значение карты находится между 3 и 6, и другое — если значение карты равно 10, валету, даме или королю.

```
int main()
{
    char card_name[3];
    puts("Введите название карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Проверяем, находится ли значение между 3 и 6 */
    if ((val > 2) && (val < 7))
        puts("Счетчик увеличился");
    /* В противном случае проверяем, была ли эта карта 10,
       J (валетом), Q (дамой) или K (королем) */ (val == 10)
        puts("Счетчик уменьшился");
    return 0;
}
```

Есть несколько способов написания этого условия.

Вы догадались, что для этого достаточно всего одного условия?

не бывает Глупых Вопросов

В: Почему нельзя написать просто | и &?

О: Вы можете использовать | и &, если хотите. Эти операторы всегда будут проверять оба условия, тогда как второе условие в || и && часто может пропускаться.

В: Так для чего же нужны операторы | и &?

О: Эти операторы выполняют нечто большее, чем просто проверку логических условий. Они осуществляют побитовые операции с отдельными битами числа.

В: Что вы имеете в виду?

О: Итак, 6 & 4 == 4, потому что если посмотреть, какие двоичные цифры являются общими для 6 (110 в двоичном коде) и 4 (100 в двоичном коде), то это будет 4 (100).

Тест-драйв

Давайте посмотрим, что произойдет на этот раз, если мы скомпилируем и запустим программу.

Эта строчка компилирует и запускает код.

```
File Edit Window Help FiveOfSpades
> gcc cards.c -o cards && ./cards
Введите название карты:
Q
Счетчик уменьшился

> ./cards
Введите название карты:
8

> ./cards
Введите название карты:
3
Счетчик увеличился

>
```

Мы запускаем программу несколько раз, чтобы убедиться, что она работает с разными диапазонами значений.

Код работает. Совмещая разные условия с помощью логического оператора, мы вместо одного значения проверяем целый диапазон. Теперь у вас есть базовая структура для подсчета карт.

Откровения компилятора

Интервью этой недели:
Чем мы можем быть обязаны gcc?

Head First: Позвольте для начала поблагодарить Вас, gcc, за то, что выкроили для нас время в своем очень плотном расписании.

gcc: Никаких проблем, мой друг. Рад был помочь.

Head First: gcc, это правда, что Вы умеете разговаривать на многих языках?

gcc: Я свободно владею более чем шестью миллионами форм общения…

Head First: Неужели?

gcc: Шучу. Но я действительно разговариваю на многих языках. На Си, естественно, на Си++ и на Objective-C. Я могу справиться с паскалем, фортраном, PL/I и еще много с чем. О, и еще я бегло владею Go…

Head First: А что касается аппаратного обеспечения, Вы способны выдавать машинный код для многих платформ?

gcc: Практически для любой. Обычно, когда инженеры создают новый тип процессора, первым делом они хотят, чтобы на нем работала какая-то из моих разновидностей.

Head First: Как Вам удалось достичь такой невероятной гибкости?

gcc: Секрет, я полагаю, кроется в двойственности моей природы. У меня есть внешний интерфейс (front-end) — это часть меня, которая умеет понимать какой-то конкретный вид исходного кода.

Head First: Написанного на таком языке, как Си?

gcc: Именно. Мой внешний интерфейс может преобразовывать этот язык в промежуточный код. Все мои языковые интерфейсы производят одну и ту же разновидность кода.

Head First: Вы говорите, что Ваша природа двойственна...

gcc: У меня также есть и внутренний интерфейс (back-end). Это система для преобразования промежуточного кода в машинный. Полученный код может работать на многих платформах. Добавьте к этому мою осведомленность о форматах исполняемых файлов на практически любой операционной системе, о которой Вы когда-либо слышали…

Head First: И все же о Вас часто говорят, как об обычном трансляторе. Вы считаете это справедливым? Ведь Ваши обязанности этим не исчерпываются, не так ли?

gcc: Конечно, я занимаюсь чем-то немного большим, чем просто трансляцией. Например, я часто замечаю ошибки в коде.

Head First: Какие именно?

gcc: Я могу найти и самые очевидные вещи, такие как ошибки в именах переменных. Но я также могу подметить и более тонкие моменты, например повторное объявление переменных. Или предупредить программистов о том, что они выбрали для переменной имя, которое уже используется для функции, и т. д.

Head First: Выходит, Вы также следите за качеством кода?

gcc: О да. И не только за качеством, но и за производительностью. Если я обнаруживаю внутри цикла участок кода, который мог бы более эффективно работать за пределами этого цикла, я могу незаметно его переместить.

Head First: Так Вы много на что способны?

gcc: Мне нравится думать, что так оно и есть. Хотя я стараюсь не подавать виду.

Head First: Спасибо Вам, gcc.

Поработайте Компилятором

Каждый из фрагментов кода на этой странице представляет собой полноценный исходный файл. Ваша задача состоит в том, чтобы притвориться компилятором и определить, скомпилируется ли каждый из этих файлов, и если нет, то почему. Для дополнительных призовых баллов опишите, что выведет на экран при запуске каждый из этих скомпилированных файлов и работает ли код так, как было задумано.

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else {
      puts("Туз!");
    }
    return 0;
}
```

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else
      puts("Туз!");
    }
    return 0;
}
```

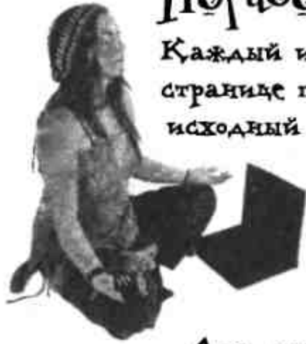
C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    } else
      puts("Туз!");

    return 0;
}
```

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else
      puts("Туз!");

    return 0;
}
```

Поработайте Компилятором. Решение



Каждый из фрагментов кода на этой странице представляет собой полноценный исходный файл. Ваша задача состояла в том, чтобы притвориться компилятором и определить, скомпилируется ли каждый из этих файлов, и если нет, то почему.

Для дополнительных призовых баллов вы должны были описать, что выведет на экран при запуске каждый из этих скомпилированных файлов и сработает ли код так, как было задумано.

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else {
      puts("Туз!");
    }
    return 0;
}
```

Код компилируется. Программа выводит строку «Дешевая карта». Но она не работает так, как ожидалось, потому что оператор else относится не к тому оператору if.

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else
      puts("Туз!");
    }
    return 0;
}
```

Код компилируется. Программа ничего не выводит и работает не так, как ожидалось, потому что оператор else относится не к тому оператору if.

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    } else
      puts("Туз!");

    return 0;
}
```

Код компилируется. Программа написана правильно и выводит «Туз!».

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("Дешевая карта");
    else
      puts("Туз!");

    return 0;
}
```

Код не скомпилируется, потому что количество открывающих и закрывающих фигурных скобок не совпадает.

Как теперь выглядит код?

```
int main()
{
    char card_name[3];
    puts("Введите название карты: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* Проверяем, находится ли значение между 3 и 6 */
    if ((val > 2) && (val < 7))
        puts("Счетчик увеличился");
    /* В противном случае проверяем, была ли эта карта 10,
       J (валетом), Q (дамой) или K (королем) */
    else if (val == 10)
        puts("Счетчик уменьшился");
    return 0;
}
```

Хм... Можем ли мы что-нибудь сделать с этой чередой операторов if? Все они проверяют одно и то же значение — card_name[0], — и большинство из них присваивают переменной значение 10. Интересно, можно ли написать это на Си более эффективно?

Программы на Си часто должны проверять одно и то же значение по несколько раз и затем в каждом случае выполнять очень похожие куски кода.

Сейчас вы можете просто использовать цепочку операторов `if` и, вероятно, этого вполне хватит. Но Си предоставляет вам альтернативный способ написания подобной логики.

Си может выполнять логические проверки с помощью инструкции switch.

Инструкция switch: вот так поворот

Иногда при написании условной логики значение одной и той же переменной необходимо проверять снова и снова. Чтобы избавить вас от необходимости использовать множество операторов `if`, в языке Си предусмотрен другой вариант — инструкция **switch**.

Инструкция `switch` чем-то похожа на `if`, но она может проверять *одну и ту же переменную* на разные значения.

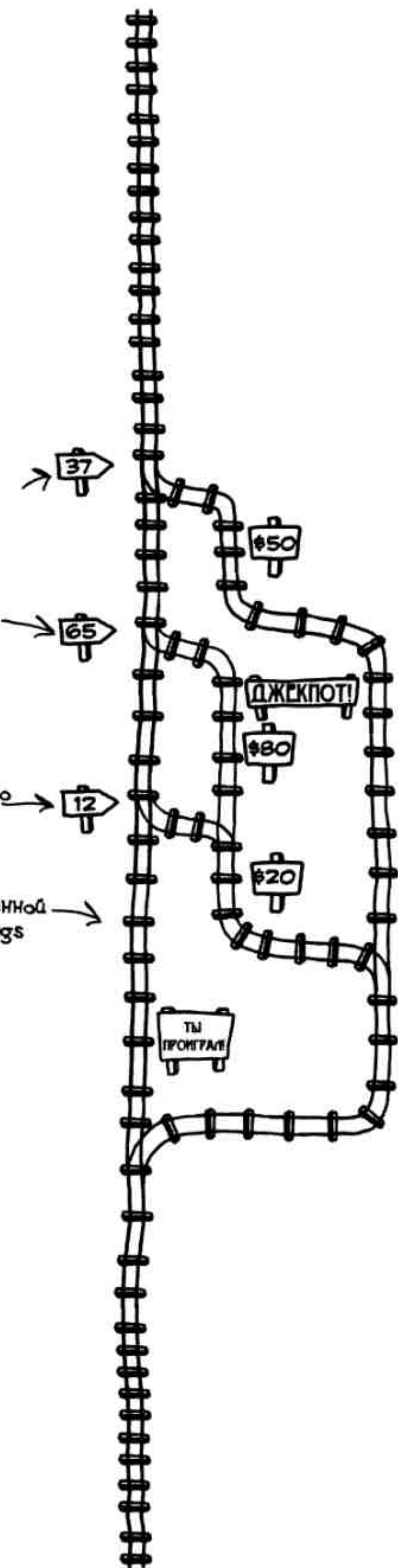
```
switch(train) {
case 37:
  winnings = winnings + 50;
  break;
case 65:
  puts("Джекпот!");
  winnings = winnings + 80;
case 12:
  winnings = winnings + 20;
  break;
default:
  winnings = 0;
}
```

Если train == 37, добавляем 50 к переменной winnings и переходим в конец.

Если train == 65, добавляем 80 к переменной winnings, ЗАТЕМ еще 20 и только потом переходим в конец.

Если train == 12, просто добавляем 20 к переменной winnings.

При любом другом значении переменной train присваиваем переменной winnings значение НОЛЬ.



Когда компьютеру попадается оператор `switch`, он проверяет данное ему значение и затем ищет подходящий оператор `case`. Отыскав совпадение, он выполняет *весь* код, который следует за найденным оператором, пока не достигнет оператора `break`. **Выполнение продолжается до тех пор, пока компьютеру не скажут прервать работу конструкции switch.**

Будьте осторожны!

Пропущенные фигурные скобки могут привести к ошибкам.

В большинстве программ на языке Си в конце каждой секции `case` *указывается инструкция* `break`*. Благодаря этому код становится более понятным, несмотря на некоторое снижение производительности.*

Наточите свой карандаш

Давайте опять взглянем на тот участок кода из вашей программы cards.c.

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else if (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

Как вы думаете, вам удастся переписать этот код с использованием инструкции switch? Свой ответ оставьте ниже:

Наточите свой карандаш Решение

Вы должны были переписать этот фрагмент кода, используя инструкцию switch.

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else if (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

```
int val = 0;
switch(card_name[0]) {
case 'K':
case 'Q':
case 'J':
  val = 10;
  break;
case 'A':
  val = 11;
  break;
default:
  val = atoi(card_name);
}
```

КЛЮЧЕВЫЕ МОМЕНТЫ

- Инструкции switch могут заменить последовательность из нескольких операторов if.
- Инструкции switch проверяют одно значение.
- Компьютер начнет выполнять код, следующий за первым совпавшим оператором case.
- Код будет выполняться до тех пор, пока не встретится оператор break или не закончится инструкция switch.
- Убедитесь в верной расстановке операторов break, иначе ваша инструкция switch будет работать неправильно.

не бывает Глупых Вопросов

В: Зачем мне может понадобиться использовать switch вместо if?

О: Инструкция switch может пригодиться, если вы проверяете одну и ту же переменную несколько раз.

В: Какие преимущества дает использование инструкции switch?

О: Их несколько. Во-первых, прозрачность. Это проще, чем использовать целый блок кода для обработки единственного значения. Набор операторов if выглядит не так очевидно. Во-вторых, вы можете применять разветвленную логику для повторного использования участков кода, предназначенных для обработки разных ситуаций.

В: Обязательно ли инструкция switch должна проверять переменную? Может ли она выполнять проверку значения?

О: Да, может. Инструкция switch просто будет сопоставлять два значения.

В: Могу ли я выполнить проверку строк внутри инструкции switch?

О: Нет, вы не можете использовать инструкцию switch для проверки строки символов или любого другого массива. Этот оператор всегда проверяет единственное значение.

Иногда одного раза недостаточно...

Мы уже многое узнали о языке Си, но осталось еще несколько моментов, которые нам нужно рассмотреть. Мы видели, как пишутся программы для множества различных ситуаций, однако существует фундаментальный вопрос, на который мы пока не обращали внимания. Что если вы хотите, чтобы ваша программа выполняла какое-то действие *снова, снова и снова*?

Использование циклов while в Си

Циклы являются особым видом управляющих инструкций. И если от оператора `if` зависит, будет ли участок кода выполняться в принципе, то цикл решает, *сколько раз* участок кода будет выполняться.

Наиболее простым видом цикла в Си является `while`. Цикл `while` выполняет код снова, снова и снова, до тех пор пока соблюдается какое-то условие.

Проверяет условие, прежде чем выполнить тело цикла.

```
while (<какое-то условие>) {
    ... /* Делаем здесь что-нибудь */
}
```

Тело находится между фигурными скобками.

Если в теле содержится всего одна строчка, вам не нужны фигурные скобки.

Достигая конечной точки тела цикла, компьютер проверяет, по-прежнему ли соблюдается условие. Если да, то код в теле цикла выполняется еще раз.

```
while (more_balls)
    keep_juggling();
```

do и while

Есть еще одна разновидность цикла `while`, которая проверяет условие *после* того, как выполнится тело цикла. Это значит, что цикл всегда выполняется **хотя бы один раз**. Данная инструкция называется `do...while`.

```
do {
  /* Купить лотерейный билет */
} while(have_not_won);
```

Циклы часто имеют одинаковую структуру

Вы можете использовать цикл `while` всякий раз, когда вам нужно повторить какой-нибудь участок кода. Однако в большинстве случаев ваши циклы будут иметь схожую структуру:

- выполнить какое-то простое действие перед началом цикла, например установить значение счетчика;
- провести проверку условия цикла;
- выполнить действие в конце цикла, например обновить счетчик.

Например, это цикл `while`, который считает от 1 до 10:

```
int counter = 1;
while (counter < 11) {
  printf("%i бутылок висело на стене\n", counter);
  counter++;
}
```

Это начальный код цикла.

Это условие цикла.

Здесь в конце тела цикла обновляется счетчик.

Не забывайте: counter++ означает «увеличить переменную counter на 1».

В таких циклах есть код для подготовки переменных непосредственно к использованию в цикле — некое условие, которое проверяется при каждой итерации, а в конце цикла — код для обновления счетчика или чего-то подобного.

Цикл for все это упрощает

Проектировщики языка Си упростили стандартную структуру цикла и создали цикл **for**. Вот тот же фрагмент кода, записанный с его помощью:

```
int counter;
for (counter = 1; counter < 11; counter++) {
  printf("%i бутылок висело на стене\n", counter);
}
```

Здесь инициализируется переменная для цикла.

Это условие, которое проверяется перед каждой итерацией цикла.

Это код, который будет запускаться после каждой итерации.

Поскольку тело цикла состоит всего из одной строчки, вы могли бы обойтись без скобок.

Циклы `for` используются в языке Си столь же *активно*, как и `while`, если не чаще. Благодаря им код становится не только короче, но и проще для понимания — весь код для управления циклом (то есть значением переменной `counter`) содержится в инструкции `for` и находится вне тела цикла.

В теле любого цикла for должно что-то находиться.

Чтобы прервать цикл, используйте break...

Вы можете создавать циклы, которые проверяют условие в начале или в конце своего тела. Но что если вы хотите выйти из цикла во время его выполнения? Вы всегда можете переписать код, но иногда проще мгновенно прервать цикл с помощью инструкции **break**:

```
while(feeling_hungry) {
    eat_cake();
    if (feeling_queasy) {
        /* Выходим из цикла while */
        break;
    }
    drink_coffee();
}
```

break немедленно завершает цикл.

Оператор break используется не только для выхода из цикла, но также внутри инструкций switch.

Осторожно применяйте оператор break, *следите за тем, откуда именно вы выходите.*

Инструкция break сразу же выведет вас из текущего цикла, пропуская оставшуюся часть его тела. Такие инструкции могут быть полезны в качестве наиболее простого способа завершения цикла. Но слишком часто их использовать не стоит — это может затруднить чтение кода.

...и continue, чтобы продолжить

Если вы хотите пропустить оставшуюся часть тела цикла и вернуться в его начало, тогда вам может пригодиться инструкция continue:

```
while(feeling_hungry) {
    if (not_lunch_yet) {
        /* Возвращаемся к условию цикла */
        continue;
    }
    eat_cake();
}
```

continue возвращает вас в начало цикла.

Байки из склепа

С помощью break нельзя завершить инструкцию if.

15 января 1990 г. междугородняя телефонная система компании AT&T вышла из строя, оставив без телефонной связи 60 тыс. человек. В чем причина? Программист, писавший для коммутаторов код на Си, попытался применить оператор break *для завершения инструкции* if. *Однако с помощью* break *инструкцию* if *завершить нельзя. В результате был пропущен целый участок кода, что привело к ошибке, которая на протяжении 9 ч вызвала разрыв 70 млн звонков...*

Подробнее о написании функций

Прежде чем опробовать на практике магию циклов, давайте немного отклонимся от темы и быстро пробежимся по функциям.

До сих пор в каждой программе вы должны были создавать функцию под названием `main()`:

```
int main()
{
  puts("Слишком молод, чтобы умереть, слишком красив, чтобы жить");
  return 0;
}
```

Это имя функции

Эта функция возвращает значение типа int.

В скобках пусто

Тело функции — это та часть, которая что-то выполняет.

Тело функции заключено в фигурные скобки.

По завершении возвращается значение.

Практически все функции в языке Си имеют один и тот же формат. Например, это программа с функцией, которая вызывается из `main()`:

```
#include <stdio.h>

int larger(int a, int b)
{
  if (a > b)
    return a;
  return b;
}

int main()
{
  int greatest = larger(100, 1000);
  printf("%Число %i наибольшее!\n", greatest)
  return 0;
}
```

Возвращает значение типа int.

Эта функция принимает два аргумента — a и b. Они оба имеют тип int.

Здесь вызывается функция.

Функция `larger()` слегка отличается от `main()`, поскольку она принимает ***аргументы*** (или ***параметры***). **Аргумент** — это всего лишь локальная переменная, которая получает свое значение из кода, вызывающего функцию. Функция `larger()` принимает два аргумента (`a` и `b`), а затем возвращает наибольший из них.

Деликатный путеводитель по стандартам

Функция `main()` возвращает тип `int`, поэтому в конце вы должны добавить инструкцию `return`. Код скомпилируется даже в том случае, если вы ее пропустите, но при этом вы получите предупреждение от компилятора. Компилятор, поддерживающий стандарт C99, вставит инструкцию `return`, если вы забудете это сделать. Чтобы компиляция проходила по стандарту C99, используйте параметр -std=99.

Подробнее о функциях типа void

Большинство функций в Си возвращает значение, но иногда вам может понадобиться создать функцию, которой нечего вернуть. Вместо *вычислений* она может просто что-то *делать*. Как правило, функции всегда должны содержать инструкцию `return`, но только не в том случае, если их возвращаемый тип **void**:

Тип void указывает на то, что функция ничего не возвращает

```
void complain()
{
    puts("Я в самом деле не в восторге");
}
```

Нет необходимости указывать инструкцию return, поскольку функция имеет тип void.

В языке Си ключевое слово `void` означает *неважно*. Сообщив компилятору, что вас не интересует значение, которое возвращает функция, вы можете не указывать инструкцию `return`.

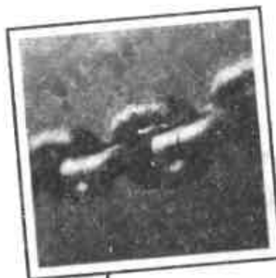
не бывает
Глупых Вопросов

В: Если я создаю функцию типа `void`, то это означает, что она не может содержать в себе инструкцию `return`?

О: Вы можете добавить инструкцию `return`, но компилятор, скорее всего, выдаст предупреждение. Нет никакого смысла указывать ее в функции типа `void`.

В: Серьезно? Почему?

О: Потому что, если вы попытаетесь прочитать значение, возвращаемое функцией типа `void`, компилятор откажется собирать ваш код.



Множественные присваивания

Почти все в языке Си имеет возвращаемый тип, и это касается не только функций. На самом деле даже операции присваивания возвращают значения. Например, взгляните на эту конструкцию:

```
x = 4;
```

Переменной присваивается число 4. Самое интересное заключается в том, что выражение `x = 4` *само* несет то же значение, которое было присвоено (то есть 4). Так почему это важно? Благодаря этому факту вы можете проделывать интересные трюки вроде связывания нескольких операций присваивания:

Теперь y тоже равно 4.

Операция присваивания x = 4 имеет значение 4.

```
y = (x = 4);
```

Эта строчка кода присвоит 4 обеим переменным `x` **и** `y`. Но вы можете немного сократить код, убрав скобки:

```
y = x = 4;
```

Вы будете часто встречать множественное присваивание в коде, где для нескольких переменных устанавливается одно и то же значение.

Перемешанные сообщения

Внизу находится короткая программа на Си. У нее не хватает одного блока. Ваша задача — **сопоставить варианты кода** (слева) **с тем, что выведет программа,** если этот код будет вставлен. Не все строчки вывода будут задействованы, но некоторые из них могут использоваться более одного раза. Соедините линиями варианты кода с соответствующим выводом командной строки.

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [                    ]
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Сюда нужно вписывать варианты кода

Варианты кода:

Сопоставьте каждый из блоков кода с одним из возможных вариантов программного вывода.

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Возможный программный вывод:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

Упражнение

Теперь, когда вы знаете, как создавать циклы `while`, измените программу таким образом, чтобы она хранила текущий счет карточной игры. Выведите текущий счет после каждой карты и завершите программу, если игрок ввел X. Выведите сообщение об ошибке, если игрок ввел неправильное значение карты, например 11 или 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while ( ..................................................................... ){
        puts("Ведите значение карты: ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':
            ........................................................................
        default:
            val = atoi(card_name);
            ........................................................................
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if (val == 10) {
            count--;
        }
        printf("Текущий счет: %i\n", count);
    }
    return 0;
}
```

Вы должны остановиться, если они ввели X.

Что вы будете делать здесь?

Вам нужно вывести ошибку, если значение не входит в диапазон от 1 до 10. Вы также должны пропустить оставшуюся часть тела цикла и повторить попытку.

Увеличиваем счетчик на 1

Уменьшаем счетчик на 1

Перемешанные сообщения Решение

Внизу находится короткая программа на Си. У нее не хватает одного блока. В вашу задачу входило **сопоставить варианты кода** (слева) **с тем, что вывела бы программа**, если бы этот код был вставлен. Не все строчки вывода должны были использоваться, некоторые из них могли употребляться более одного раза. Вам нужно было соединить линиями варианты кода с соответствующим выводом командной строки.

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [          ]
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

Сюда нужно вписывать варианты кода.

Варианты кода:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

Возможный программный вывод:

```
22 46
11 34 59
02 14 26 38
02 14 36 48
00 11 21 32 42
11 21 32 42 53
00 11 23 36 410
02 14 25 36 47
```

Упражнение Решение

Теперь, когда вы знаете, как создавать циклы `while`, вам следовало изменить программу таким образом, чтобы она хранила текущий счет карточной игры. Выведите текущий счет после каждой карты и завершите программу, если игрок ввел X. Выведите сообщение об ошибке, если игрок ввел неправильное значение карты, например 11 или 24.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    while ( card_name[0] != 'X' ) {
        puts("Ведите значение карты: ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':
            continue;
        default:
            val = atoi(card_name);
            if ((val < 1) || (val > 10)) {
            puts("Я не понимаю это значение!");
            continue;
            }
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if (val == 10) {
            count--;
        }
        printf("Текущий счет: %i\n", count);
    }
    return 0;
}
```

Мы должны проверить, равняется ли первый символ X.

Break не выведет нас из цикла, поскольку мы находимся внутри инструкции switch. Мы должны использовать оператор continue, чтобы вернуться в начало и снова проверить условие цикла.

Это всего лишь один из способов написания условий.

Здесь нужен еще один оператор continue, потому что мы хотим продолжить выполнение цикла.

Тест-драйв

На этом программа для подсчета карт завершена. Пришло время испытать ее на деле. Как думаете, будет ли она работать?

Не забывайте, что для Windows не нужно набирать ./.

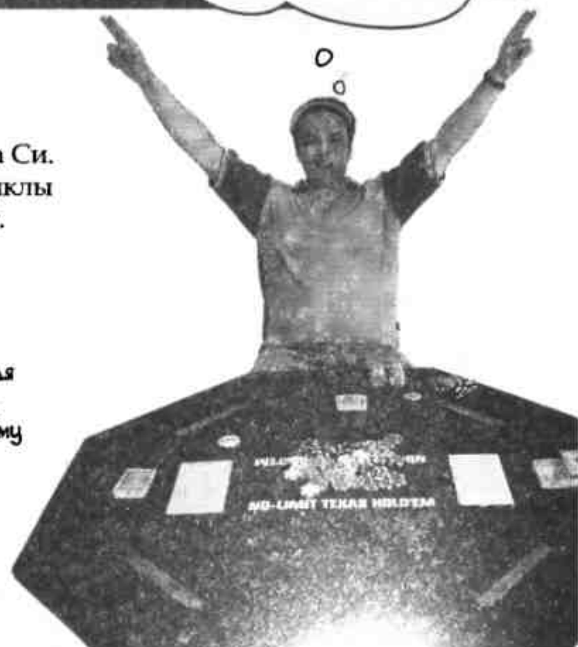
Эта строчка скомпилирует и запустит программу.

```
File Edit Window Help GoneLoopy
> gcc card_counter.c -o card_counter && ./card_counter
Введите значение карты:
4
Текущий счет: 1
Введите значение карты:
K
Текущий счет: 0
Введите значение карты:
3
Текущий счет: 1
Введите значение карты:
5
Текущий счет: 2
Введите значение карты:
23
Я не понимаю это значение!
Введите значение карты:
6
Текущий счет: 3
Введите значение карты:
5
Текущий счет: 4
Введите значение карты:
3
Текущий счет: 5
Введите значение карты:
X
```

Здесь мы проверяем, является ли значение карты корректным.

Счетчик увеличивается!

Делая большие ставки при высоком значении счетчика, я разбогател!



Программа для подсчета карт работает!
Вы завершили написание своей первой программы на Си. Используя мощь инструкций этого языка, включая циклы и условия, вы создали полноценный счетчик для карт.

Отличная работа!

Лирическое отступление: использование компьютера для подсчета карт запрещено законом во многих штатах, и ребятам из казино это может не понравиться. Поэтому не делайте так, договорились?

не бывает Глупых Вопросов

В: Почему я должен компилировать код на Си? В других языках, таких как JavaScript, компиляция не нужна, не так ли?

О: Компиляция нужна, чтобы сделать код быстрым. Но даже в некоторых некомпилируемых языках, таких как JavaScript и Python, для увеличения скорости используется некий вид скрытой компиляции.

В: Является ли Си++ всего лишь другой версией Си?

О: Нет. Си++ изначально был создан как расширение Си, но сейчас это уже нечто большее. И Си++, и Objective-C проектировались, чтобы использовать в Си объектно-ориентированный подход.

В: Что это за объектно-ориентированный подход? Будем ли мы его изучать в этой книге?

О: Эта методика была создана для борьбы со сложностью программ. В данной книге мы не будем специально на ней останавливаться.

В: Си во многом похож на JavaScript, Java, Си# и т. д.

О: Си обладает очень ограниченным синтаксисом. Он оказал влияние на многие другие языки.

В: Как расшифровывается gcc?

О: GNU Compiler Collection (набор компиляторов в составе проекта GNU).

В: Почему набор? Их там больше одного?

О: GNU Compiler Collection можно применять для компиляции многих языков, но, вероятно, Си — тот язык, для которого данный компилятор используется чаще всего.

В: Могу ли я создать бесконечный цикл?

О: Да. Если условием цикла является 1, то такой цикл будет работать вечно.

В: Может ли создание бесконечных циклов быть оправданным?

О: Иногда. Бесконечный цикл (цикл, который никогда не заканчивается) часто используется в таких программах, как сетевые серверы, которые раз за разом выполняют одну и ту же задачу, пока их не остановят. Но большинство программистов проектируют циклы с расчетом на то, что они когда-нибудь завершатся.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Цикл `while` работает до тех пор, пока условие истинно.
- Цикл `do-while` действует аналогично, но выполняет код по меньшей мере один раз.
- Цикл `for` — это более компактный способ написания определенных видов циклов.
- Вы можете выйти из цикла в любой момент с помощью инструкции `break`.
- Вы можете перейти к условию цикла в любое время, используя инструкцию `continue`.
- Инструкция `return` возвращает значение из функции.
- Функции типа `void` не нуждаются в инструкции `return`.
- Большинство выражений в Си имеют значения.
- Операции присваивания обладают значениями, поэтому вы можете объединять их вместе (`x = y = 0`).

Ваш инструментарий языка Си

Вы изучили главу 1, пополнив свои навыки умением работать с базовым инструментарием языка Си. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

Инструкции switch целесообразно использовать для проверки одной переменной на несколько значений.

Простые инструкции являются командами.

Блочные инструкции заключены в { и }.

Любая программа должна иметь функцию main().

Прежде чем запускать свою программу на Си, вам необходимо ее скомпилировать.

#include подключает внешний код, например для ввода или вывода.

Инструкция if выполняет код, если выражение истинно.

Вы можете использовать оператор && в командной строке, чтобы запускать программу только в том случае, если она проходит компиляцию.

Вы можете объединять условия с помощью && и ||.

В ключе -o указывается итоговый файл.

gcc — самый популярный компилятор для языка Си.

Имена ваших исходных файлов должны заканчиваться на .c.

Циклы do-while выполняют код по меньшей мере один раз.

count++ добавляет 1 к count.

count-- вычитает 1 из count.

Цикл for — более компактный способ написания циклов.

Цикл while повторяет код до тех пор, пока условие истинно.

2 Память и указатели

На что ты указываешь?

Чтобы сделать что-то серьезное с помощью Си, необходимо разобраться, как этот язык управляет памятью.

Язык Си позволяет вам лучше *контролировать* использование **компьютерной памяти** в программе. В этой главе вы приоткроете занавес и узнаете, что именно происходит, когда вы **считываете и записываете переменные**. Вы изучите **принцип работы массивов**, научитесь избегать **грубых ошибок при работе с памятью** и, что самое главное, поймете, что умелое обращение с **указателями и адресацией** — это ключ к высококлассному программированию на Си.

Код на языке Си содержит указатели

Чтобы лучше понять указатели, рассмотрим все по порядку.

Указатели являются наиболее фундаментальным понятием в языке программирования Си. Так что же это такое? **Указатель** — это просто адрес фрагмента данных в памяти.

В Си указатели используются по нескольким причинам.

1. **Вместо того чтобы полностью копировать данные, вы можете просто передать указатель.**

Это указатель на местоположение информации

2. **Может возникнуть ситуация, когда два участка кода должны работать с одним и тем же фрагментом данных, а не с его отдельными копиями.**

Указатели помогают решить обе проблемы: избежать копирования данных и сделать их общими. Но если указатели — это всего лишь адреса, чем же они так смущают некоторых? Причина в том, что они **не совсем очевидны**. Если вы невнимательны, можно легко потерять указатель в памяти. Поэтому главное при изучении работы с указателями в Си — *не спешить*.

Расслабьтесь

Не торопитесь при чтении этой главы.

В основе указателей лежит простая идея, но, чтобы разобраться во всем основательно, нужно некоторое время. Не отказывайте себе в частых перерывах, пейте побольше воды. И если вы застряли на каком-то этапе, можете даже понежиться в теплой ванне.

В недрах памяти

Чтобы понять суть указателей, нужно погрузиться в память компьютера.

Каждый раз, когда вы объявляете переменную, компьютер выделяет для нее место где-то в памяти. Если переменная была объявлена внутри функции `main()`, компьютер поместит ее в область памяти под названием **«стек»**. Если объявление произошло вне всяких функций, будет использована **глобальная** область памяти.

```
int y = 1;

int main()
{
  int x = 4;
  return 0;
}
```

Переменная y будет находиться в глобальной области. Адрес в памяти 1 000 000. Значение 1.

Переменная x будет находиться в стеке. Адрес в памяти 4 100 000. Значение 4.

Компьютер может выделить для переменной x участок памяти под номером, скажем, 4 100 000. Если вы присвоите переменной число 4, компьютер сохранит это значение по адресу 4 100 000.

Если вы хотите получить адрес переменной в памяти, можете использовать оператор **&**:

&x – это адрес x.

```
printf("x хранится по адресу %p\n", &x);
```

%p используется для форматирования адресов.

Вот что выведет данный код.

```
x хранится по адресу 0x3E8FA0
```

Это 4 100 000 в шестнадцатеричном представлении.

Скорее всего, на своем компьютере вы получите другой адрес.

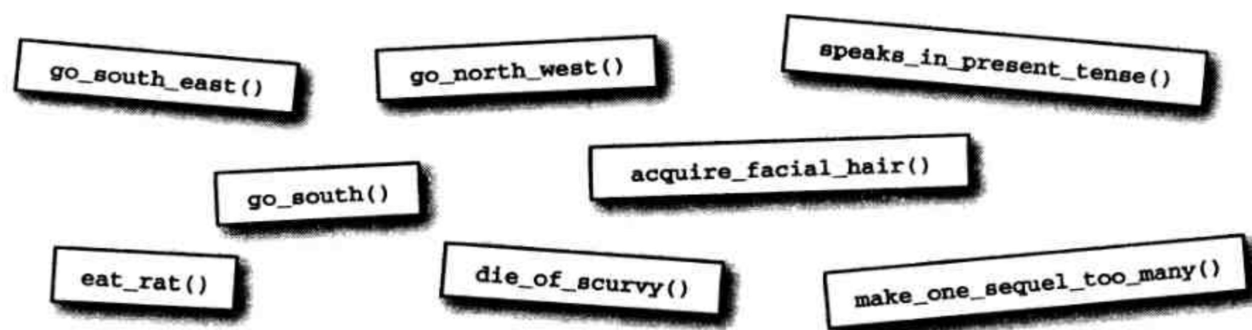
Адрес переменной говорит о том, где именно в памяти ее нужно искать. Вот почему адрес также называют ***указателем***, ведь он ***указывает*** на переменную в памяти.

Переменная, объявленная внутри функции, обычно помещается в стек. Переменная, объявленная вне функции, хранится в глобальной области.

Отправляемся в плавание с указателями

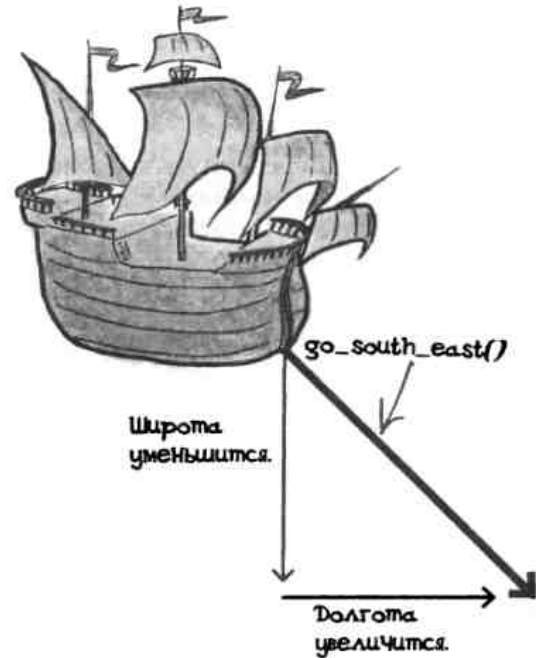
Представьте, что вы пишете игру, участники которой должны прокладывать себе путь в обход...

В игре нужно учесть многие моменты: счет, количество жизней и текущее местоположение игроков. Вряд ли вам захочется писать игру в виде одного цельного куска кода. Вместо этого вы создадите множество небольших функций, каждая из которых делает что-то полезное:



Но какое отношение к этому имеют указатели? Не будем над этим задумываться, а просто начнем писать код. Как и прежде, вы будете использовать обычные переменные. Большая часть игры заключается в навигации корабля в обход Бермудского прямоугольника, поэтому давайте подробнее рассмотрим, что должен делать код в одной из навигационных функций.

Курс на юго-восток, капитан!



Игра будет отслеживать местоположение игрока по широте и долготе. Широта показывает положение относительно юга и севера, а долгота — относительно востока и запада. Если игрок хочет плыть на юго-восток, то значение широты будет *уменьшаться*, а долготы — *увеличиваться*.

Таким образом, вы можете написать функцию `go_south_east()`, которая в качестве аргументов принимает широту и долготу, соответственно уменьшая и увеличивая их:

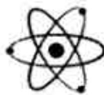
```
#include <stdio.h>

void go_south_east(int lat, int lon)
{
  lat = lat - 1;
  lon = lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;
  go_south_east(latitude, longitude);
  printf("Стоп! Теперь наши координаты: [%i, %i]\n",
latitude, longitude);
  return 0;
}
```

Передаем широту и долготу

Уменьшаем широту.

Увеличиваем долготу.

Программа изначально размещает корабль в точку с координатами [32; -64], следовательно, если он направится на юго-восток, его новое местоположение будет [31; -63]. По крайней мере, *если наш код работает*…



Сила мозга

Взгляните на код внимательно. Вы считаете, он будет работать? Если нет, объясните почему.

Тест-драйв

Код должен передвинуть корабль на юго-восток из точки с координатами [32; -64] в новую точку с координатами [31; -63]. Но если вы скомпилируете и запустите программу, произойдет следующее:

Что за... Корабль остался на месте.

Где же сражение?

```
File Edit Window Help Savvy?
> gcc southeast.c -o southeast
> ./southeast
Стоп! Теперь наши координаты:
[32, -64]
>
```

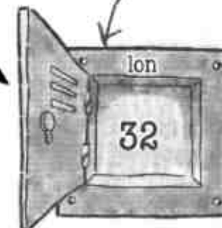
Местоположение корабля *совершенно* не изменилось.

В Си аргументы передаются по значению

Причина неработающего кода кроется в способе вызова функции в Си.

1. Изначально функция `main()` имела локальную переменную под названием `longitude` со значением 32.

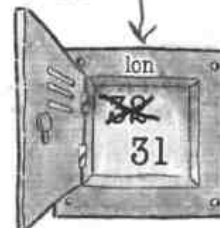
Это новая переменная, которая содержит копию значения longitude.



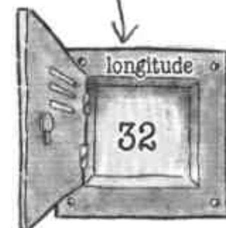
2. Когда компьютер вызывает функцию `go_south_east()`, он **копирует значение** переменной `longitude` в аргумент `lon`. Это просто присваивание значения `longitude` переменной `lon`. При вызове функции вы передаете в виде аргумента не *переменную*, а только ее *значение*.

3. Изменения, которые функция `go_south_east()` вносит в переменную `lon`, касаются только ее локальной копии. Это означает, что, когда компьютер вернется к функции `main()`, переменная `longitude` будет иметь свое начальное значение, то есть 32.

Изменения вносятся только в локальную копию.



Значение первоначальной переменной остается прежним.

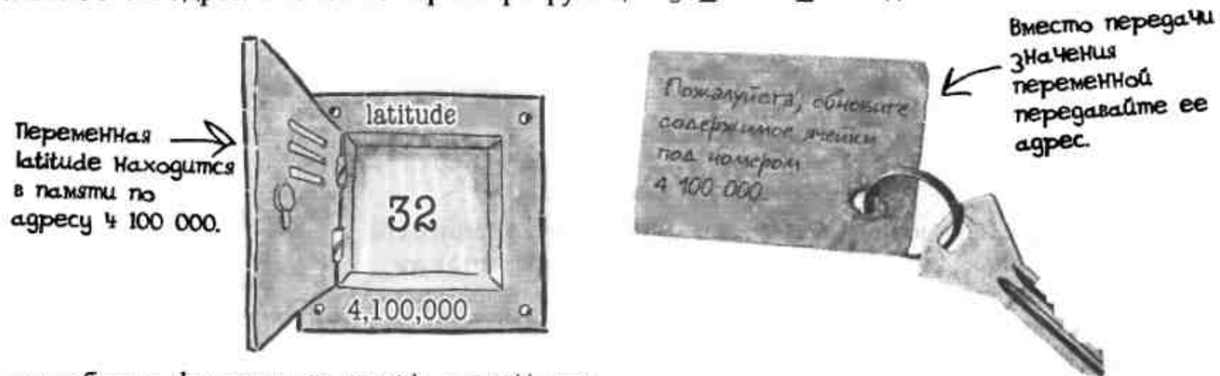


Как в таком случае на языке Си можно написать функцию, которая обновляет переменную?

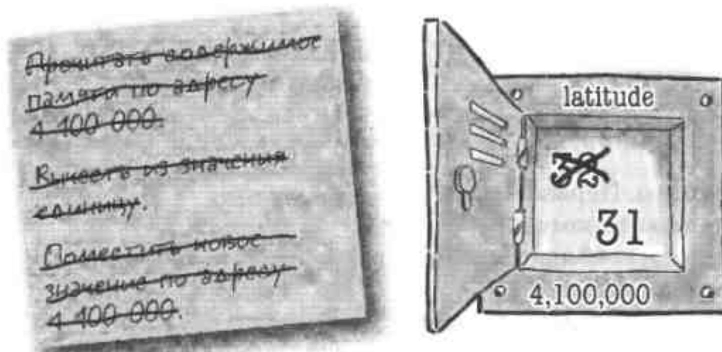
Легко, если использовать указатели...

Попробуйте передать указатель переменной

Что произойдет, если мы передадим не *значения* переменных `latitude` и `longitude`, а их *адреса*? Допустим, переменная `latitude` находится в стековой памяти по адресу 4 100 000. Что будет, если вы передадите значение этого адреса в качестве параметра функции `go_south_east()`?



Если сообщить функции `go_south_east()`, что `latitude` находится по адресу 4 100 000, то она сможет не только найти текущее значение этой переменной, но и изменить его. Функция должна всего лишь обновить содержимое памяти по адресу 4 100 000.



Поскольку функция `go_south_east()` изменяет первоначальную переменную `latitude`, компьютер, вернувшись к функции `main()`, сможет вывести на экран обновленные координаты.

С помощью указателей проще разделять память

Одна из основных причин использовать указатели — разрешить функциям *разделять* память. Данные, созданные в одной функции, могут быть изменены в другой (если ей известен адрес этих данных в памяти).

Теперь вы в общих чертах знаете, как с помощью указателей можно исправить функцию `go_south_east()`. Самое время детально рассмотреть, как это делается.

не бывает Глупых Вопросов

В: Я вывел адрес переменной на своем компьютере, но это было не 4 100 000. Я сделал что-то не так?

О: Вы все сделали правильно. Адреса в памяти, которые ваша программа использует для переменных, на разных компьютерах будут разными.

В: Почему локальные переменные хранятся в стеке, а глобальные — в другом месте?

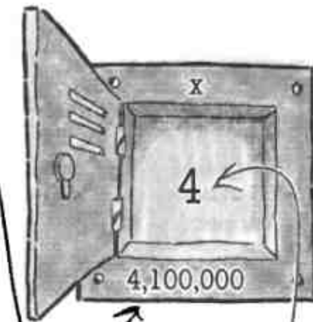
О: Они используются по-разному. У глобальной переменной может быть только одна-единственная копия, но если вы напишете функцию, которая вызывает саму себя, то можете получить очень много экземпляров одной и той же локальной переменной.

В: Для чего используют другие области памяти?

О: Вы узнаете, для чего нужны другие области памяти, когда дочитаете эту книгу.

Использование указателей в памяти

Есть **три** вещи, которые вы должны знать, чтобы считывать и записывать данные с помощью указателей.



1. **Получение адреса переменной.**
Вы уже видели, как можно узнать местоположение переменной в памяти, используя оператор &:

Благодаря формату %p адрес будет выведен в шестнадцатеричном виде.

```
int x = 4;
printf("x находится по адресу %p\n", &x);
```

Оператор & найдет адрес переменной: 4 100 000.

Получив адрес переменной, вы, вероятно, захотите сохранить его где-нибудь. Чтобы это сделать, вам потребуется **указатель, или ссылочная переменная**. Это обычная переменная, которая хранит адрес в памяти. При объявлении необходимо сообщить, данные какого типа хранятся по адресу, на который указывает эта переменная:

Это указатель с адресом, по которому хранится целое число.

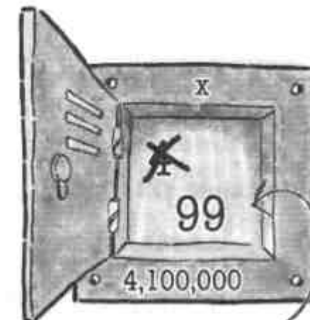
```
int *address_of_x = &x;
```

2. **Считывание содержимого, хранящегося по адресу.**
Если вы знаете адрес в памяти, то с помощью оператора * вы можете прочитать данные, которые хранятся по этому адресу:

```
int value_stored = *address_of_x;
```

Так вы прочитаете данные по адресу, записанному в address_of_x. value_stored получит значение 4, которое изначально хранилось в переменной x.

Операторы & и * прямо противоположны. Первый берет фрагмент данных и сообщает адрес, по которому они хранятся; второй берет заданный адрес и сообщает, какие данные там находятся. Иногда указатели называют ссылками, а оператор * — оператором **разыменовывания**.



3. **Изменение содержимого, хранящегося по адресу.**
Если у вас есть ссылочная переменная и вы хотите изменить данные по адресу, на который она указывает, то вы снова можете воспользоваться оператором *. Но на этот раз его нужно использовать в левой части операции присваивания:

```
*address_of_x = 99;
```

Это заменит содержимое оригинальной переменной x числом 99.

Итак, теперь вы знаете, как считывать содержимое из памяти по определенному адресу и как его записывать. Пришло время исправить функцию go_south_east().

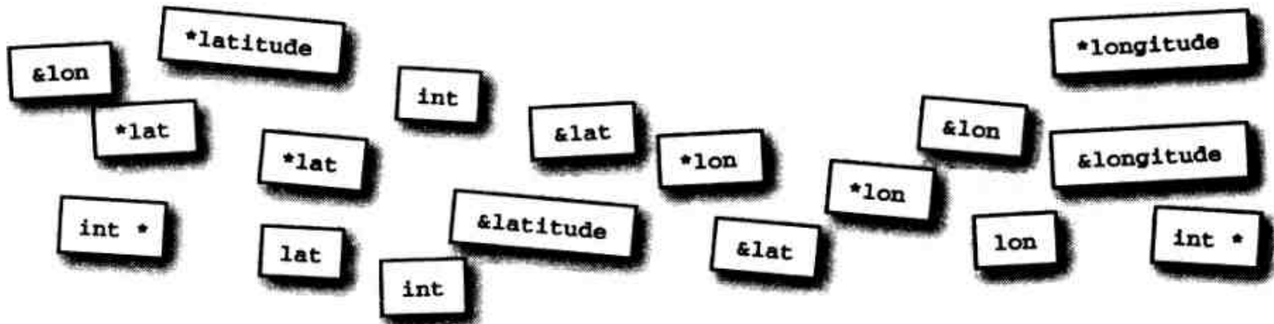
Магнитики в компасе

Вам нужно исправить функцию `go_south_east()` таким образом, чтобы для обновления данных она использовала указатели. Хорошенько подумайте, какой тип данных нужно передавать в функцию и какие операторы необходимо использовать для обновления координат корабля.

Какой тип аргументов будет хранить адреса в памяти для целых чисел?

```
#include <stdio.h>

void go_south_east(.......................... lat, .......................... lon)
{
                  = .......................... - 1;

                  = .......................... + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(.........................., ..........................);
  printf("Стоп! Теперь наши координаты: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

Не забывайте: вы собираетесь передавать адреса переменных.



Магнитики в компасе Решение

Вам нужно было исправить функцию `go_south_east()` таким образом, чтобы для обновления данных она использовала указатели. Необходимо было хорошенько подумать, какой тип данных передавать в функцию и какие операторы использовать для обновления координат корабля.

```
#include <stdio.h>

void go_south_east(int * lat, int * lon)
{
  *lat = *lat - 1;
  *lon = *lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(&latitude, &longitude);
  printf("Стоп! Теперь наши координаты: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

В аргументах будут храниться указатели, поэтому их тип должен быть int *.

*lat может прочитать старое значение и установить новое.

Вам нужно найти адреса переменных latitude и longitude с помощью оператора &.

Тест-драйв

Скомпилировав и запустив *новую* версию функции, вы получите следующее:

К юго-востоку от начального местоположения.

```
> gcc southeast.c -o southeast
> ./southeast
Стоп! Теперь наши координаты: [31, -63]
>
```

Код работает.
Поскольку функция принимает в качестве аргументов указатели, она может обновлять первоначальные переменные `latitude` и `longitude`. Теперь вам известно, как создавать функции, которые не просто возвращают значения, но и могут изменять любые данные по адресу, переданному им.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменные — это выделенные фрагменты памяти.
- Локальные переменные находятся в стеке.
- Глобальные переменные находятся в глобальной области.
- Указатели — это обычные переменные, которые хранят адреса в памяти.
- Оператор `&` находит адрес переменной.
- Оператор `*` может прочитать содержимое по определенному адресу в памяти.
- Оператор `*` также может изменять содержимое по определенному адресу в памяти.

не бывает Глупых Вопросов

В: Указатели представляют собой настоящие адреса? Или это какой-то другой вид ссылок?

О: Это действительные адреса в памяти процесса.

В: Что это значит?

О: Каждому процессу выдается упрощенная версия памяти, которая выглядит как длинная последовательность байтов.

В: А на самом деле память выглядит иначе?

О: В действительности все сложнее, чем кажется. Но от процесса эти подробности скрыты, поэтому операционная система может перемещать его по памяти, выгружать и перезагружать в какое-то другое место.

В: Разве память — это не просто длинный список байтов?

О: Скорее всего, компьютер будет структурировать физическую память более сложным способом. Как правило, адреса группируются в отдельные хранилища по чипам памяти.

В: Должен ли я в этом разбираться?

О: В большинстве случаев вам незачем знать, как именно происходит управление памятью компьютера.

В: Почему при выводе указателей я должен использовать строку форматирования %p?

О: Вам не обязательно указывать строку `%p`. Большинство современных компьютеров допускают использование строки `%li`, хотя при этом компилятор может выдать предупреждение.

В: Почему при использовании формата %p адрес в памяти выводится в шестнадцатеричном виде?

О: Так инженеры обычно указывают адреса в памяти.

В: Чем *указатели* отличаются от *ссылок*?

О: Иногда программисты называют указатели *ссылками*. Однако в языке Си++ это слово обозначает несколько другое понятие.

В: Ах да, Си++. Мы будем его рассматривать?

О: Нет, эта книга только о языке Си.

Как передать строку в функцию

Вы уже знаете, как передавать в функцию простые значения в виде аргументов. Но что если вам нужно отправить что-то более сложное, например строку? Если помните, в предыдущей главе мы говорили, что строки в Си на самом деле являются массивами символов. Следовательно, если вы хотите передать в функцию строку, это можно сделать следующим образом:

```
void fortune_cookie(char msg[])
{
  printf("Сообщение гласит: %s\n", msg);
}

char quote[] = "Печенье вас полнит";
fortune_cookie(quote);
```

В функцию будет передан массив символов.

Аргумент msg определен как массив, но он не содержит информации о длине, поэтому вы не будете знать, сколько символов в этом массиве. *Звучит* логично, но есть в этом кое-что странное…

Дорогая, кто уменьшил строку?

В языке Си есть оператор **sizeof**, с помощью которого можно узнать, сколько байтов что-то занимает в памяти. Вы можете передавать ему тип данных или сами данные:

```
sizeof(int)
sizeof("Черепахи!")
```

На большинстве компьютеров эта строка вернет 4.

При однобайтной кодировке эта строчка вернет 10: 9 печатных символов плюс нуль-символ \0.

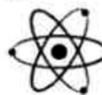
Однако, взглянув на длину строки, которая передается в функцию, вы заметите нечто странное:

```
void fortune_cookie(char msg[])
{
  printf("Сообщение гласит: %s\n", msg);
  printf("msg занимает %i байтов\n", sizeof(msg));
}
```

```
File Edit Window Help TakeAByte
> ./fortune_cookie
Сообщение гласит: Печенье вас полнит
msg занимает 8 байтов
>
```

8? А на некоторых компьютерах мы можем получить даже 4! Как же так?

Вместо того чтобы вывести полную длину строки, код возвращает всего 4 или 8 байтов. Что случилось? Почему компьютер считает, что мы передали более короткую строку?



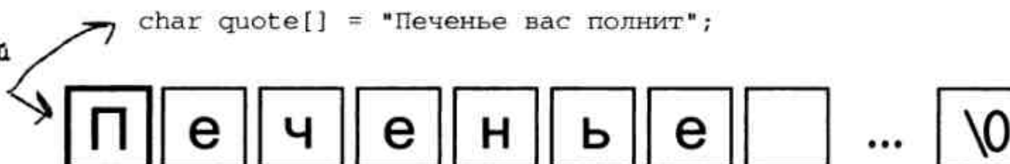
Сила мозга

Как вы думаете, почему число, которое возвращает `sizeof(msg)`, меньше длины целой строки? Что такое `msg`? Почему на разных компьютерах возвращаются разные величины?

Переменные массивов похожи на указатели

Когда вы создаете массив, его переменную можно использовать в качестве **указателя** на начало массива в памяти. Например, компьютер видит строчку кода наподобие

```
char quote[] = "Печенье вас полнит";
```

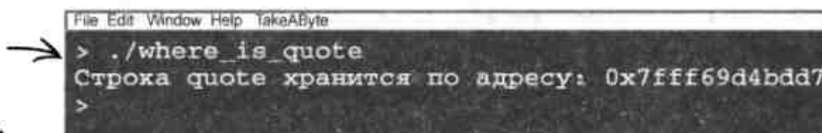


Он выделяет в стеке место для каждого символа строки и еще для нуль-символа `\0`. Кроме того, он связывает **адрес первого символа** с переменной `quote`. Каждый раз, когда переменная `quote` используется в коде, компьютер заменяет ее адресом первого символа в строке. Фактически переменная массива — это *то же самое, что указатель*:

Вы можете использовать quote как ссылочную переменную даже несмотря на то, что это массив.

```
printf("Строка quote хранится по адресу: %p\n", quote);
```

Если вы напишете тестовую программу для вывода адреса, то получите результат, похожий на этот



Значит, нашей функции был передан указатель

Вот что послужило причиной странного поведения функции `fortune_cookie()`, хотя все выглядело так, будто вы передавали строку, на самом деле это был всего лишь указатель на нее:

```
void fortune_cookie(char msg[])
{
  printf("Сообщение гласит: %s\n", msg);
  printf("msg занимает %i байтов\n", sizeof(msg));
}
```

На самом деле msg является указателем.

msg указывает на сообщение.

sizeof(msg) — это размер указателя.

Вот почему оператор `sizeof` вернул такой странный результат. Это был размер **указателя на строку**. В 32-битных операционных системах указатель занимает в памяти 4 байта, а в 64-битных — 8 байтов.

О чем думает компьютер, когда выполняет ваш код

1. **Компьютер видит функцию.**

```
void fortune_cookie(char msg[])
{
 ...
}
```

2. **Затем он видит содержимое функции.**

```
printf("Сообщение гласит: %s\n", msg);
printf("msg занимает %i байтов\n", sizeof(msg));
```

3. **Компьютер вызывает функцию.**

```
char quote[] = "Печенье вас полнит";
fortune_cookie(quote);
```

КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменную массива можно использовать как указатель.
- Переменная массива указывает на первый элемент массива.
- Аргумент функции, объявленный в качестве массива, будет рассматриваться как указатель.
- Оператор `sizeof` возвращает объем, который занимает фрагмент данных.
- Вы также можете использовать `sizeof` для типов данных, например `sizeof(int)`.
- `sizeof(указатель)` возвращает 4 в 32-битных системах, и 8 в 64-битных.

не бывает Глупых Вопросов

В: `sizeof` — это функция?

О: Нет, это оператор.

В: А в чем разница?

О: Оператор преобразуется компилятором в последовательность инструкций. Но, если компьютер вызывает функцию, он должен перейти к отдельному участку кода.

В: Выходит, `sizeof` вычисляется во время компиляции программы?

О: Да. Компилятор может определить размер хранилища в процессе компиляции.

В: Почему указатели имеют разные размеры на разных компьютерах?

О: *В 32-битных системах адреса в памяти хранятся* в виде 32-битных чисел. Поэтому они и называются 32-битными системами. 32 бита == 4 байта. Таким образом, 64-битные системы используют для хранения адресов 8 байтов.

В: Находится ли в памяти созданный мною указатель?

О: Да. Указатель — это всего лишь переменная, хранящая число.

В: А могу ли я найти адрес ссылочной переменной?

О: Да, используя оператор `&`.

В: Могу я преобразовать указатель в обычное число?

О: В большинстве систем можете. Компиляторы языка Си обычно используют для типа `long` тот же размер, что и для адреса в памяти. Таким образом, если `p` — это указатель и вы хотите сохранить его в переменную типа `long`, то вы можете набрать код `a = (long)p`. Чуть позже в этой главе мы вернемся к данному вопросу.

В: В *большинстве* систем? Значит, это не гарантировано?

О: Это не гарантировано.

Любовь с первого взгляда

Сегодня у нас классическое трио холостяков, готовых сыграть в игру «Любовь с первого взгляда».

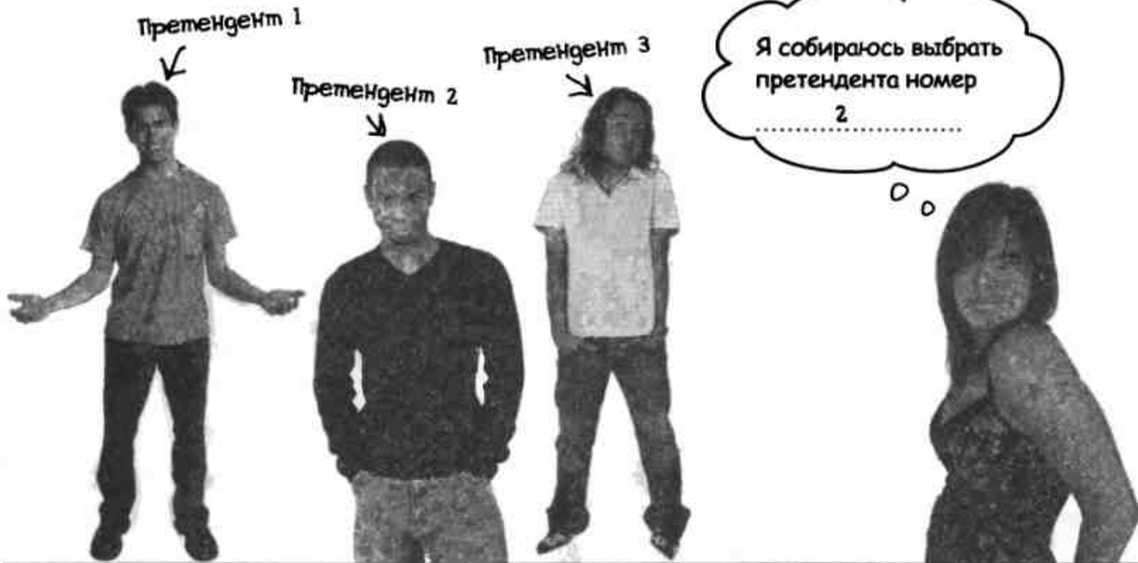
Девушка попытает свое счастье, выбрав одного из претендентов. Кто же станет счастливчиком?

```
#include <stdio.h>

int main()
{
  int contestants[] = {1, 2, 3};
  int *choice = contestants;
  contestants[0] = 2;
  contestants[1] = contestants[2];
  contestants[2] = *choice;
  printf("Я собираюсь выбрать претендента номер %i\n", contestants[2]);
  return 0;
}
```

Любовь с первого взгляда решение

У нас было классическое трио холостяков, готовых сыграть в игру «Любовь с первого взгляда».

Девушка выбрала одного из претендентов. Кто же этот счастливчик?




```
#include <stdio.h>

int main()
{
  int contestants[] = {1, 2, 3};
  int *choice = contestants;
  contestants[0] = 2;
  contestants[1] = contestants[2];
  contestants[2] = *choice;
  printf("Я собираюсь выбрать претендента номер %i\n", contestants[2]);
  return 0;
}
```

choice теперь содержит адрес массива contestants

contestants[2]
-- *choice
-- contestants[0]
-- 2

Переменные массивов и указатели не одно и то же

Даже несмотря на то, что вы можете использовать переменную массива в качестве указателя, между ними существуют некоторые различия. Чтобы их увидеть, рассмотрите этот фрагмент кода.

```
char s[] = "Насколько он большой?";
char *t = s;
```

1 **`sizeof(массив)` — это... размер массива.**
Вы уже видели, как `sizeof(указатель)` возвращает 4 или 8 — размеры указателей в 32- и 64-битных системах соответственно. Но если вы вызовете `sizeof` для переменной массива, Си сообразит, что вы хотите узнать, **сколько места массив занимает в памяти.**

2 **Адрес массива — это... адрес массива.**
Указатель — это просто переменная, которая хранит адрес в памяти. Но что насчет переменной массива? Если вы примените к ней оператор &, результат не будет отличаться от значения самой переменной.

&s == s &t != t

Если программист напишет `&s`, это будет означать: «Каков адрес массива `s`?» Адрес массива `s` — это просто... `s`. Но если кто-то напишет `&t`, это будет означать: «Каков адрес переменной `t`?»

3 **Переменная массива не может указывать ни на что другое.**
Когда вы создаете ссылочную переменную, компьютер выделяет 4 или 8 байтов для ее хранения. Но при создании массива компьютер выделяет память только для него самого, но *не* для его переменной. Вместо нее компилятор вставляет адрес начала массива.

Поскольку переменные массивов не имеют выделенного хранилища, они не могут указывать на что-либо другое.

Это вызовет ошибку компиляции. → `s = t;`

Распад массива

Поскольку ссылочные переменные немного отличаются от переменных массивов, вам нужно быть внимательным при присваивании массивов указателям. Присваивая массив ссылочной переменной, вы передаете ей только **адрес** массива, без всяких сведений о его размере. Таким образом, теряется некоторая информация. Эта потеря называется **распадом**.

Каждый раз, когда массив передается в функцию, он распадается до указателя — это неизбежно. Нужно следить за распадом массивов в своем коде, иначе это может привести к очень коварным ошибкам.

Дело о смертельном списке

В этом особняке было все, о чем только можно мечтать: ландшафтный садик, люстры, собственная ванная. Его 94-летний владелец, Амори Мамфорд III, был найден мертвым в саду — предположительно, сердечный приступ. Была ли смерть естественной? Первоначально врач посчитал, что всему виной была передозировка сердечных препаратов. Но, похоже, что дело с душком и речь не только о мертвом теле в беседке. Проходя в зал мимо полицейских, он приблизился к новоиспеченной 27-летней вдове Мамфорда Баблз.

Загадка на пять минут

«Ничего не понимаю. Он всегда так внимательно обращался с лекарствами. Вот список дозировок». Она показала ему код для дозатора медикаментов.

```
int doses[] = {1, 3, 2, 1000};
```

«Полиция утверждает, будто это я перепрограммировала дозатор. Но я не разбираюсь в технике. Они говорят, что я написала этот код, но я думаю, что он даже не скомпилируется».

Пальцы с безупречно выполненным маникюром скользнули в сумочку и достали оттуда копию программы, которую полиция нашла рядом с кроватью покойного. Было совсем не похоже на то, что она скомпилируется…

```
printf("Отпущенная доза %i", 3[doses]);
```

Что означало выражение `3[doses]`? 3 — это не массив.

Баблз высморкалась. «Я бы такое никогда не написала. И в любом случае тройная доза — это не так уж плохо, разве нет?»

Тройная доза не убила бы старика. Но, возможно, этот код не так прост, как кажется на первый взгляд…

Почему массивы на самом деле начинаются с нуля

Переменную массива можно использовать в качестве указателя на его первый элемент. Иначе говоря, вы можете прочитать первый элемент массива, используя либо квадратные скобки, *либо* оператор *:

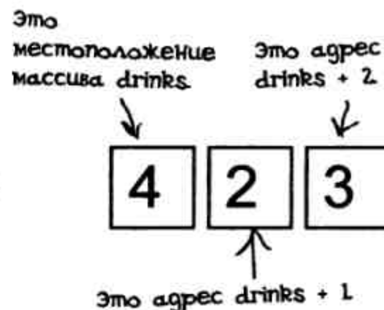
Эти строки кода равнозначны

```
int drinks[] = {4, 2, 3};
printf("1-й заказ: %i напитка\n", drinks[0]);
printf("1-й заказ: %i напитка\n", *drinks);
```

drinks[0] == *drinks

Поскольку адрес — это обычное число, вы можете производить с указателем **арифметические действия** и фактически ***прибавить*** к нему значение, чтобы получить следующий адрес. Таким образом, получить элемент с индексом 2 можно двумя способами: использовать квадратные скобки или просто добавить к адресу первого элемента число 2:

```
printf("3-й заказ: %i напитка\n", drinks[2]);
printf("3-й заказ: %i напитка\n", *(drinks + 2));
```



В целом выражения `drinks[i]` и `*(drinks + i)` равнозначны. Вот почему массивы начинаются с нуля. Индекс — это всего лишь число, которое добавляется к указателю, чтобы найти местоположение элемента.

Наточите свой карандаш

Воспользуйтесь силой арифметических операций при работе с указателями, чтобы утешить разбитое сердце. Эта функция пропустит первые три символа в текстовом сообщении.

```
void skip(char *msg)
{
  puts(.........................);
}

char *msg_from_amy = "Не звони мне";
skip(msg_from_amy);
```

Какое выражение нужно сюда подставить, чтобы начать вывод с четвертого символа?

Функция должна напечатать это сообщение, начиная с буквы «з».

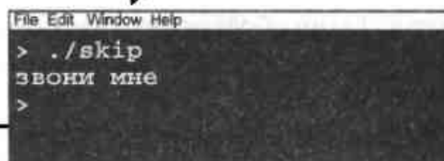
Наточите свой карандаш
Решение

Вы должны были воспользоваться силой арифметических операций при работе с указателями, чтобы утешить разбитое сердце. Эта функция пропускает первые три символа в текстовом сообщении.

```
void skip(char *msg)
{
  puts(msg + 3);
}

char *msg_from_amy = "Не звони мне";
skip(msg_from_amy);
```

Добавив к указателю msg число 3, вы начнете печать с четвертого символа.

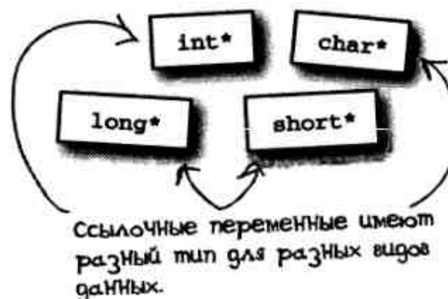
Вот, что выведет код.



Зачем указателям тип

Если указатель — это просто адрес, зачем тогда ему нужен тип? Почему бы просто не хранить все указатели в виде какой-то обобщенной ссылочной переменной?

Дело в том, что арифметические операции с указателями довольно *коварны*. Если добавить **1** к указателю типа char, он будет указывать на следующий адрес в памяти. Но только потому, что char занимает **в памяти 1 байт**.

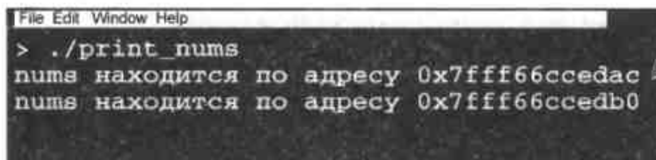


А что если мы имеем дело с указателем типа int? Этот тип обычно занимает 4 байта, поэтому, если вы добавите 1 к указателю типа int, скомпилированный код на самом деле к адресу в памяти добавит 4.

```
int nums[] = {1, 2, 3};
printf("nums находится по адресу %p\n", nums);
printf("nums + 1 находится по адресу %p\n", nums + 1);
```

После запуска этого кода два адреса будут разделены в памяти *более* чем 1 байтом. Тип нужен указателю для того, чтобы **компилятор знал, насколько нужно подкорректировать арифметические операции.**

(nums + 1) находится в 4 байтах от nums.



Не забывайте: эти адреса выводятся в шестнадцатеричном формате.

Дело о смертельном списке

В прошлый раз мы оставили нашего героя во время разговора с Баблз Мамфорд, чей муж получил передозировку из-за подозрительного кода. Кто же на самом деле эта Баблз — хакер-злодей или всего лишь несчастная вдова? Чтобы это выяснить, читайте дальше...

Врач положил код в карман. «Было приятно с Вами побеседовать, миссис Мамфорд. Думаю, больше нет необходимости Вас беспокоить». Он пожал ей руку. «Спасибо Вам, — ответила она, вытирая по-детски невинные голубые глаза, полные слез. — Вы были так добры».

«Не так быстро! — Баблз едва успела удивленно приоткрыть рот, прежде чем на ее руках защелкнулись наручники. — По Вашим рукам видно, что Вы знаете об этом преступлении больше, чем говорите». Чтобы такие мозоли появились на кончиках пальцев, нужно провести за клавиатурой немало времени.

«Баблз, Вы знаете о Си гораздо больше, чем пытаетесь показать. Еще раз взгляните на этот код».

```
int doses[] = {1, 3, 2, 1000};
printf("Отпущенная доза %i", 3[doses]);
```

«Я почуял неладное, когда увидел выражение `3[doses]`. Вы знали, что переменную массива `doses` можно использовать как указатель. Смертельную дозу можно было выдать вот так...» Он нацарапал на чистой стороне салфетки несколько вариантов кода:

```
doses[3] == *(doses + 3) == *(3 + doses) == 3[doses]
```

«Ваш код стал последней каплей. Он отпустил старику тысячекратную дозу. А сейчас Вы отправляетесь туда, где больше никогда не сможете незаконно использовать синтаксис языка Си...»

КЛЮЧЕВЫЕ МОМЕНТЫ

- Переменные массивов могут быть использованы в качестве указателей...
- ...но переменные массивов и указатели — это не совсем одно и то же.
- `sizeof` возвращает разные значения для указателей и переменных массивов.
- Переменные массивов не могут указывать на что-либо другое.
- Если присвоить указателю переменную массива, она теряет информацию о количестве элементов.
- Массивы начинаются с нуля вследствие арифметических операций над указателями.
- Ссылочные переменные должны иметь тип, чтобы корректировать работу арифметических операций над указателями.

не бывает
Глупых Вопросов

В: Нужно ли мне на самом деле разбираться в арифметических операциях над указателями?

О: Некоторые программисты стараются не использовать арифметические операции над указателями, так как при этом легко допустить ошибку. Но с их помощью можно эффективно обрабатывать массивы данных.

В: Можно ли вычитать числа из указателей?

О: Да. Но следите за тем, чтобы не выйти за пределы выделенного для массива места.

В: На каком этапе Си корректирует вычисления, связанные с арифметическими операциями над указателями?

О: Это происходит, когда компилятор генерирует исполняемый файл. Он определяет тип переменной, а затем умножает ее размер на добавляемый или вычитаемый индекс элемента.

В: Продолжайте...

О: Если компилятор видит, что вы работаете с массивом типа `int` и прибавляете к нему число 2, он умножает это число на 4 (размер `int`) и в итоге добавляет 8.

В: Использует ли Си оператор `sizeof` при корректировании арифметических операций с указателями?

О: В сущности, да. Оператор `sizeof` тоже вычисляется на этапе компиляции, поэтому для разных типов данных в нем будут использоваться те же размеры.

В: Могу ли я умножать указатели?

О: Нет.

Использование указателей для ввода данных

Вы уже знаете, как получить строку, введенную пользователем на клавиатуре. Это можно сделать с помощью функции `scanf()`:

Вы сохраните имя в этот массив.

```
char name[40];
printf("Введите ваше имя: ");
scanf("%39s", name);
```

scanf прочитает не более 39 символов и еще символ завершения строки \0.

Как работает функция `scanf()`? Она принимает указатель типа `char`, хотя в данном случае вы передаете ей переменную массива. Но теперь вы должны понимать, **почему** функция `scanf()` принимает указатель. Она должна *обновить* содержимое массива. Функциям, которые обновляют переменные, нужно не само значение, а его **адрес**.

Ввод чисел с помощью scanf()

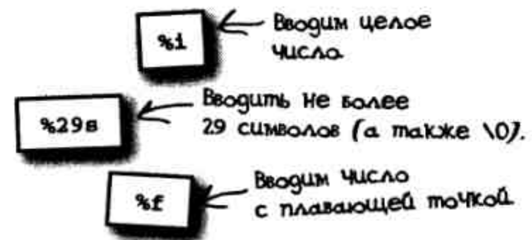
Так как же ввести данные в **числовое поле**? Это можно сделать, передав *указатель* в числовую переменную.

%i означает, что пользователь будет вводить значение типа int.

```
int age;
printf("Введите ваше имя: ");
scanf("%i", &age);
```

Чтобы получить адрес переменной int, используйте оператор &.

Получив адрес числовой переменной, функция `scanf()` может обновить ее содержимое. Чтобы упростить себе задачу, вы можете указать строку форматирования, содержащую те же коды, что вы передавали функции `printf()`. Вы даже можете использовать `scanf()` для считывания сразу нескольких блоков информации:



Этот код считывает имя, затем пробел, а затем фамилию.

```
char first_name[20];
char last_name[20];
printf("Введите имя и фамилию: ");
scanf("%19s %19s", first_name, last_name);
printf("Имя: %s Фамилия:%s\n", first_name, last_name);
```

Имя и фамилия хранятся в отдельных массивах.

```
File Edit Window Help Meerkats
> ./name_test
Введите имя и фамилию: Сандерс Кляйнфельд
Имя: Сандерс Фамилия: Кляйнфельд
>
```

Будьте осторожны при использовании scanf()

У функции `scanf()` есть одна небольшая... проблема. До сих пор при написании кода мы очень внимательно относились к ограничениям на количество символов, которые может прочитать функция `scanf()`:

```
scanf("%39s", name);

scanf("%2s", card_name);
```

Почему так? В конце концов `scanf()` использует тот же вид строк форматирования, что и `printf()`. При выводе строки с помощью `printf()` мы использовали простой спецификатор `%s`. Но при использовании этого же спецификатора со `scanf()` могут возникнуть проблемы, если пользователь перестарается с количеством символов:

```
char food[5];
printf("Введите любимую еду: ");
scanf("%s", food);
printf("Любимая еда: %s\n", food);
```

```
File Edit Window Help TakeAByte
> ./food
Введите любимую еду: печенка-мандарин-енот-ириски
Любимая еда: печенка-мандарин-енот-ириски
Segmentation fault: 11
>
```

Программа аварийно завершилась. Все из-за того, что `scanf()` записывает данные за пределами того участка, который был выделен массиву `food`.

scanf() может приводить к переполнениям буфера

Если вы забудете ограничить длину строки, считываемой вами с помощью `scanf()`, то у программы может не хватить места для вводимых пользователем данных. Лишние данные будут записаны в память, которая не была должным образом выделена компьютером. Если вам повезет, данные просто сохранятся, не вызвав никаких проблем.

Но существует *очень* большая вероятность того, что переполнения буфера приведут к ошибкам. На экране могут появиться сообщения вроде *segmentation fault* или *abort trap*, но результат будет один — программа ***аварийно завершится***.

fgets() — альтернатива функции scanf()

Есть и другая функция, с помощью которой можно вводить текстовые данные, — **fgets()**. Так же как и scanf(), она принимает указатель типа char. Отличие заключается в том, что функции fgets() должна быть передана максимальная длина:

Эта программа делает то же, что и предыдущая.

```
char food[5];
printf("Введите любимую еду: ");
fgets(food, sizeof(food), stdin);
```

Сначала она принимает указатель на буфер.

Затем она принимает максимальный размер строки (включая /0).

stdin просто означает, что данные будут приходить с клавиатуры

Позже вы узнаете больше о stdin

Это означает, что при вызове fgets() вы не можете случайно забыть указать длину — она находится прямо в сигнатуре функции в качестве обязательного аргумента. Также обратите внимание, что размер буфера в fgets() **включает** завершающий символ \0. Поэтому вам не нужно вычитать 1 из длины, как это делается в scanf().

БАЙКИ ИЗ СКЛЕПА

На самом деле fgets() происходит от другой, более старой функции gets().

Функция fgets() показала себя как более безопасная по сравнению со scanf(). Более старая функция gets() значительно проигрывает им обеим в этом плане. Дело в том, что fgets() не предусматривает **никаких ограничений в принципе:**

Серьезно, не используйте ее.

```
char dangerous[10];
gets(dangerous);
```

Функция gets() существует довольно давно. Единственное, что вам стоит о ней знать, — **использовать ее на самом деле не нужно.**

Что еще вы должны знать о fgets()?

Использование sizeof в сочетании с fgets()

В приведенном выше коде максимальная длина задается с помощью оператора sizeof. Но будьте осторожны. Не забывайте, что sizeof возвращает количество памяти, занимаемой переменной. В данном коде sizeof вернет размер массива, представленного переменной food. Если бы food была ссылочной переменной, мы бы получили всего лишь размер указателя.

Если вы уверены в том, что передаете в функцию fgets() переменную массива, то можете смело использовать sizeof. Если же передается обычный указатель, вам необходимо ввести нужный размер самостоятельно:

Если бы переменная food была обычным указателем, вместо использования sizeof нужно было бы явно указать длину.

```
printf("Введите любимую еду: ");
fgets(food, 5, stdin);
```

Бой за чемпионский титул

Отложите все ваши дела! Пришло время для долгожданного боя за чемпионский титул. В красном углу — проворный, как молния, гибкий, но слегка опасный, главный хулиган в области ввода данных — `scanf()`. В синем углу — простой, безопасный, этого парня вы могли бы показать своей маме — `fgets()`!

	scanf():	**fgets():**
Раунд первый: Ограничения		
Ограничиваете ли вы количество символов, которое пользователь может ввести?	`scanf()` может ограничить количество вводимых данных, если вы не забыли указать размер в строке форматирования.	У `fgets()` есть обязательное ограничение. Его нельзя обойти.

Результат: fgets() выигрывает этот раунд по очкам.

	scanf():	**fgets():**
Раунд второй: Несколько полей		
Можно ли вас использовать для ввода более чем одного поля?	Да! `scanf()` не только позволяет вводить несколько полей, но и поддерживает структурированные данные, предоставляя возможность указывать символы, которые будут вставлены между полями.	Ох! `fgets()` пропускает удар в подбородок. `fgets()` позволяет вводить в буфер только одну строку. Никаких других типов данных. Только строки. И только один буфер.

Результат: scanf() уверенно побеждает в этом раунде.

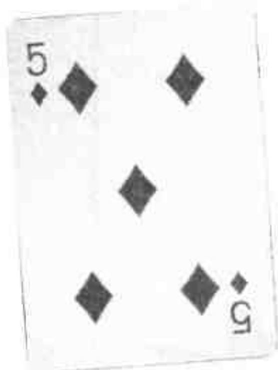
	scanf():	**fgets():**
Раунд третий: Пробелы в строках		
Может ли введенная строка содержать пробелы?	Ух! Тут уж `scanf()` получил по полной программе. Считывая строку со спецификатором `%s`, он останавливается, как только встречает пробел. Поэтому, если вы хотите ввести более одного слова, вам придется вызывать `scanf()` несколько раз или использовать хитрый фокус с регулярными выражениями.	Никаких проблем с пробелами. `fgets()` может каждый раз считывать целую строку.

Результат: Ответный удар! Этот раунд за fgets().

Хороший честный бой между двумя задиристыми функциями. Очевидно, что если вы хотите вводить структурированные данные с *несколькими* полями, то лучше использовать `scanf()`. При вводе единственной неструктурированной строки лучше подойдет `fgets()`.

Кто-нибудь хочет сыграть в наперстки?

В подсобном помещении бара Head First играют в азартную карточную игру. Тасуют три карты, ваша задача — внимательно следить и решить, где, как вам кажется, находится дама. Естественно, что в баре Head First настоящие карты заменяет *код*. Вот используемая программа:



```
#include <stdio.h>

int main()
{
  char *cards = "JQK";
  char a_card = cards[2];
  cards[2] = cards[1];
  cards[1] = cards[0];
  cards[0] = cards[2];
  cards[2] = cards[1];
  cards[1] = a_card;
  puts(cards);
  return 0;
}
```

Найдите даму

Код предназначен для перемешивания букв внутри строки «JQK», состоящей их трех символов (по первым буквам карт валет, дама, король в переводе на английский). Не забывайте, что в Си строка является всего лишь массивом символов. Программа меняет символы местами, после чего выводит полученную строку.

Игроки делают ставки на то, где, по их мнению, окажется буква «Q», затем программа компилируется и запускается.

Ой... у нас проблемы с памятью...

Похоже, с кодом, который написал карточный шулер, возникли проблемы. После компиляции и запуска на экране ноутбука в баре Head First появится следующее:

```
> gcc monte.c -o monte && ./monte
bus error
```

Более того, когда парни пытаются запустить этот же код на других компьютерах и операционных системах, они получают целый вагон различных ошибок:

```
File Edit Window Help HolyCrap
> gcc monte.c -o monte && ./monte
monte.exe has stopped working
```

БАБАХ!

SEGPHAULT!

БУМ!

BUS ERROR!

SEGMENTATION ERROR!

Что не так с этим кодом?

Самое время использовать **интуицию**. Не переусердствуйте. **Просто попытайтесь угадать**. *Просмотрите* все возможные ответы и выберите среди них *один*, который вы считаете правильным.

В чем, **по-вашему**, заключается проблема?

Строка не может быть изменена.	
Символы выходят за пределы строки.	
Строка не находится в памяти.	
Что-то другое.	

Попробуйте догадаться решение

Вам нужно было просмотреть все возможные ответы и, используя **интуицию**, выбрать среди них *один* правильный.

В чем же, **по-вашему**, заключалась проблема?

Строка не может быть изменена.	✓
Символы выходят за пределы строки.	
Строка не находится в памяти.	
Что-то другое.	

Строковый литерал никогда не может быть изменен

Переменная, которая указывает на строковый литерал, не может использоваться для изменения его содержимого:

```
char *cards = "JQK";
```

← Эта переменная не может изменить данную строку.

Но если вы создадите массив из строкового литерала, то **сможете** его изменять:

```
char cards[] = "JQK";
```

Все сводится к тому, как Си работает с памятью...

Внутри памяти: char * cards = "JQK";

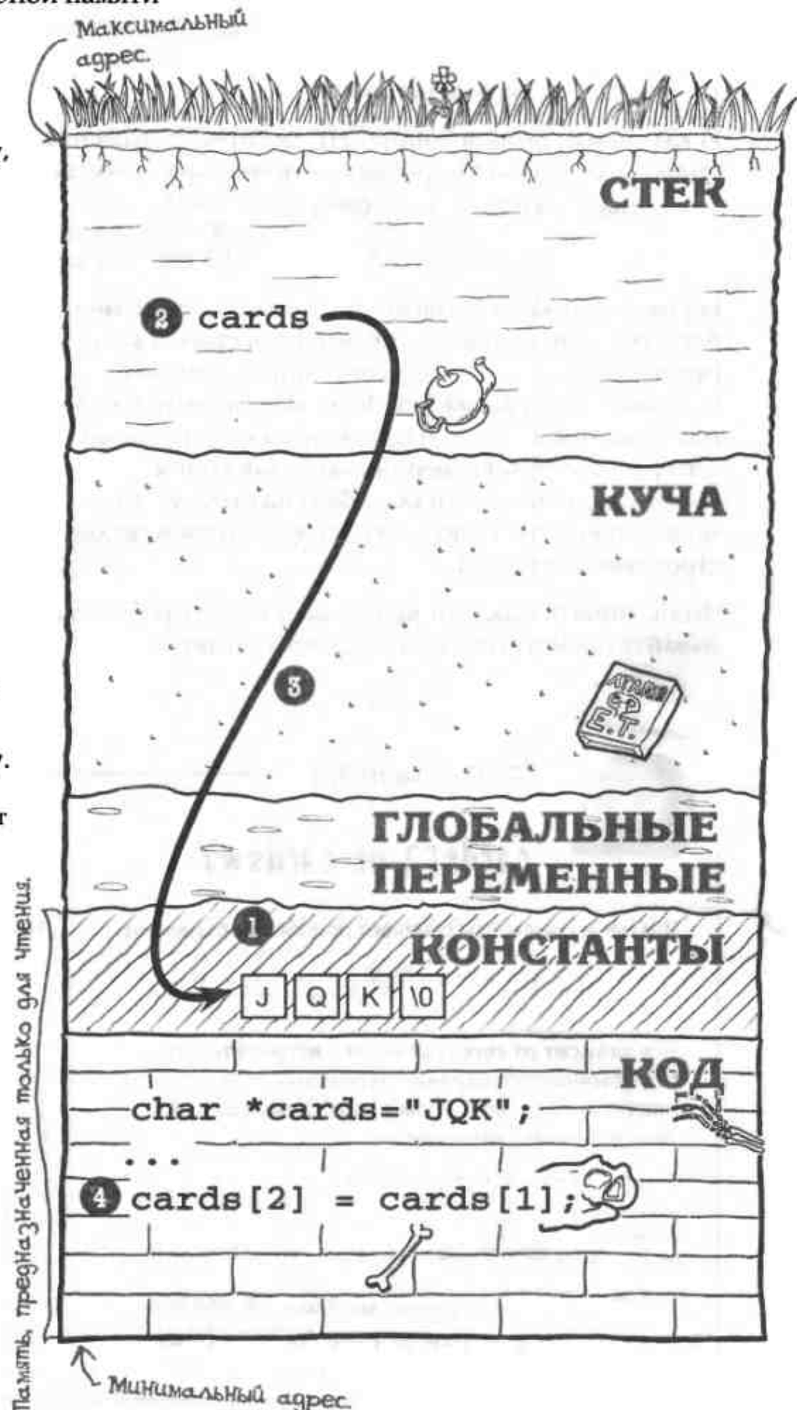
Чтобы понять, почему эта строка кода приводит к ошибке, нам нужно погрузиться вглубь компьютерной памяти и увидеть, что именно там происходит.

1. **Компьютер загружает строковый литерал.**
   Когда компьютер загружает программу, он помещает все постоянные (константные) значения наподобие строкового литерала «JQK» в область памяти, предназначенную **только для чтения.**

2. **Программа создает переменную cards в стеке.**
   Стек — это область памяти, которую компьютер использует для локальных переменных, размещенных внутри функций. Там и будет находиться переменная `cards`.

3. **Переменной cards присваивается адрес «JQK».**
   Переменная `cards` будет содержать адрес **строкового литерала «JQK»**. Помните, эта строка находится в памяти, предназначенной только для чтения.

4. **Компьютер пытается изменить строку.**
   Когда программа пытается изменить содержимое строки, на которую указывает переменная `cards`, оказывается, что она не может этого сделать, — строку можно только прочитать.



Дружище, я не могу ее изменить — она находится в константной области памяти. Ее можно только прочитать.

Таким образом, проблема заключалась в том, что строковые литералы, такие как «JQK», хранятся в памяти, предназначенной только для чтения.

Но раз мы определили проблему, то как мы ее решим?

Если вы собираетесь изменить строку — сделайте копию

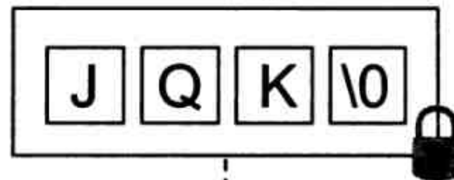
Дело в том, что если мы хотим изменить содержимое строки, нам нужно убедиться, что у нас есть ее **копия**. Если мы создадим копию строки в области памяти, предназначенной *не* только для чтения, у нас не возникнет никаких проблем.

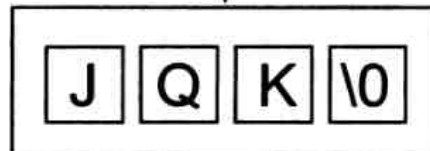
И как же мы сделаем копию? Ну, мы просто создадим строку, как делали это раньше, — в виде *нового массива*.

```
char cards[] = "JQK";
```

cards — это не просто указатель. Теперь это массив.

Эта строка находится в области памяти, предназначенной для чтения...





...поэтому скопируйте строку в область памяти, которая может быть изменена.

Вероятно, не вполне очевидно, почему это все меняет. *Все* строки являются массивами. Но в старом коде переменная `cards` была всего лишь *указателем*, в новом — это уже ***массив***. Если вы объявите массив под названием `cards` и присвоите ему строковый литерал, это будет совершенно новая копия. Переменная не просто *указывает* на строку. Это полностью новый массив, содержащий свежую ***копию*** строкового литерала.

Чтобы понять, как этот код решает нашу проблему, давайте посмотрим, что он делает с памятью.

Уголок ботана

cards[] or cards*?

Что же *на самом деле* означает подобное объявление?

```
char cards[]
```

Все зависит от того, где вы его встретите. Если это обычное объявление переменной, то `cards` является массивом и вам необходимо сразу же присвоить ему значение:

```
int my_function()
{
  char cards[] = "JQK";
  ...
}
```

Cards — это массив.

Размер массива не указан, поэтому вам нужно сразу же присвоить ему что-нибудь.

Но если переменная `cards` была объявлена в качестве *аргумента функции*, значит она является указателем.

```
void stack_deck(char cards[])
{
  ...
}

void stack_deck(char *cards)
{
  ...
}
```

Cards — указатель типа char.

Эти две функции равнозначны.

Внутри памяти: char cards[ ]="JQK";

Мы уже видели, что происходит со *сломанным кодом*, но что насчет изменений, которые мы внесли? Давайте посмотрим.

1. **Компьютер загружает строковый литерал.**
   Как и до этого, когда компьютер загружает программу, он помещает все постоянные (константные) значения, такие как строковый литерал «JQK», в область памяти, предназначенную только для чтения.

2. **Программа создает переменную cards в стеке.**
   Программа должна создать массив, который мы объявили. У него должна быть достаточная длина, чтобы хранить строку «JQK», — 4 символа.

3. **Программа инициализирует массив.**
   Поскольку мы выделили место, программа скопирует содержимое строкового литерала «JQK» в стек памяти.

Итак, разница заключается в том, что в оригинальном коде ссылочная переменная указывала на строковый литерал, предназначенный только для чтения. Но при инициализации массива с помощью строкового литерала вы получите *копию* символов, каждый из которых можно изменять как угодно.

Тест-драйв

Посмотрим, что произойдет, если мы создадим в коде **новый массив**:

```
#include <stdio.h>

int main()
{
  char cards[] = "JQK";
  char a_card = cards[2];
  cards[2] = cards[1];
  cards[1] = cards[0];
  cards[0] = cards[2];
  cards[2] = cards[1];
  cards[1] = a_card;
  puts(cards);
  return 0;
}
```

```
File Edit Window Help Where'sTheLady?
> gcc monte.c -o monte && ./monte
QKJ
```

Код работает! Теперь наша переменная cards указывает на незащищенную область памяти, поэтому мы можем свободно изменять ее содержимое.

Уголок ботана

Один из способов избавиться от этой проблемы в будущем — никогда не писать код, в котором простой указатель типа char ссылается на значение строкового литерала. Например:

```
char *s = "Какая-то строка";
```

В указателе на строковый литерал нет ничего плохого. Проблема появляется только в том случае, когда вы пытаетесь *модифицировать* строку. Если вам все же нужно создать указатель на строковый литерал, не забудьте использовать ключевое слово const:

```
const char *s = "Какая-то строка";
```

Таким образом, если компилятор увидит, что какой-то код пытается изменить строку, он выдаст вам ошибку компиляции:

```
s[0] = 'S';
monte.c:7: error: assignment of read-only location
```

Дело о магической пуле

Он просматривал на компьютере свой личный каталог оружия и боеприпасов, когда раздался стук в дверь и вошла она. Высокая блондинка с приличной сумкой для ноутбука и в дешевых туфлях. Он сразу понял, что она программистка. «Вы должны мне помочь… Вы должны стереть его имя! Джимми ни в чем не виноват, я точно знаю. Не виноват!» Он протянул ей платок, чтобы вытереть слезы, и предложил присесть.

История стара как мир. Она встретила парня, который знал другого парня. Джимми Бломштейн протирал столики в местной кофейне, по выходным разъезжал на велосипеде и пополнял свою коллекцию чучел. Он надеялся, что однажды сумеет накопить достаточно денег для покупки слона. Но связался с дурной компанией. Грабитель в маске (по кличке masked_raider) и Джимми встретились в кафе за утренней чашкой кофе и оба на тот момент были живы.

Загадка на пять минут

```
char masked_raider[] = "живой!";
char *jimmy = masked_raider;
    printf("Грабитель в маске сейчас %s Джимми сейчас %s\n",
     masked_raider, jimmy);
```

```
File Edit Window Help
Грабитель в маске сейчас живой! Джимми сейчас живой!
```

Позже в то же утро грабитель в маске вынужден был удалиться, чтобы совершить ограбление. Таких ограблений на его счету была целая сотня. Но на этот раз он не учел, что в подсобном помещении баре Head First компания правительственных агентов наслаждается еженедельной карточной игрой. Итак, представьте себе картину: звуки выстрела, крик — и через миг злодей уже валяется на тротуаре, истекая кровью:

```
masked_raider[0] = 'M';
masked_raider[1] = 'E';
masked_raider[2] = 'P';
masked_raider[3] = 'T';
masked_raider[4] = 'B';
masked_raider[5] = '!';
```

Проблема в том, что когда Тутс (так звали блондинку) пришла проведать своего парня в кофейню, ей сообщили, что мокаччино, который он сегодня подал посетителям, был последним в его жизни.

```
printf("Грабитель в маске сейчас %s Джимми сейчас %s\n", masked_
raider, jimmy);
```

```
File Edit Window Help
Грабитель в маске сейчас МЕРТВ! Джимми сейчас МЕРТВ!
```

Что, по-вашему, произошло? Как одна магическая пуля смогла убить и Джимми, и грабителя?

Дело о магической пуле

Как одна магическая пуля смогла убить и Джимми, и грабителя?

Джимми, застенчивый бариста, загадочным образом был застрелен одновременно со злодеем в маске.

```
#include <stdio.h>
int main()
{
  char masked_raider[] = "Живой!";
  char *jimmy = masked_raider;
  printf("Грабитель в маске сейчас %s Джимми сейчас %s\n", masked_raider,
  jimmy);
  masked_raider[0] = 'M';
  masked_raider[1] = 'E';
  masked_raider[2] = 'P';
  masked_raider[3] = 'T';
  masked_raider[4] = 'B';
  masked_raider[5] = '!';
  printf("Грабитель в маске сейчас %s Джимми сейчас %s\n",
  masked_raider, jimmy);
  return 0;
}
```

Примечание отдела по маркетингу: уберите скрытую рекламу напитка «Фруктовый заряд для мозгов» — сделка сорвалась

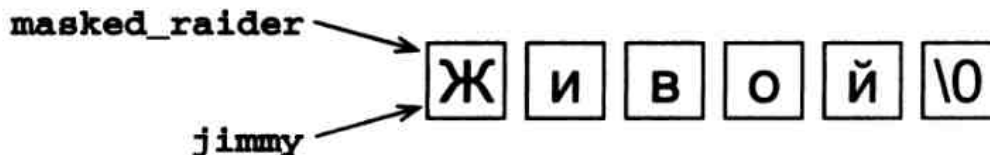
Загадка на пять минут разгадана

Прежде чем детектив сумел докопаться до сути, прошло некоторое время. Выпив большой стакан освежающего напитка «Фруктовый заряд для мозгов», он откинулся на спинку своего кресла и посмотрел через стол прямо в ее голубые глаза. Она выглядела, словно кролик под светом фар приближающегося грузовика, и было совершенно понятно, что за рулем находится именно он.

«Боюсь, у меня для Вас плохие новости. Джимми и грабитель в маске… одно и то же лицо!»

«Нет!»

Она сделала глубокий вдох, поднеся руки к лицу. «Простите, но я говорю то, что думаю. Взгляните на использование памяти», — он нарисовал схему.



«jimmy и masked_raider — это всего лишь псевдонимы для одного и того же адреса в памяти. Они оба указывают на одно и то же место. Когда masked_raider словил пулю, то же самое сделал и jimmy. Добавьте к этому еще счет из приюта для слонов в Сан-Франциско и вот этот заказ на 15 тонн упаковочного материала — и дело закрыто».

КЛЮЧЕВЫЕ МОМЕНТЫ

- Если вы видите в объявлении переменной символ `*`, это значит, что переменная будет указателем.
- Строковые литералы хранятся в памяти, предназначенной только для чтения.
- Если вы хотите изменить строку, ее нужно скопировать в новый массив.
- Вы можете объявить указатель как `const char *`, чтобы запретить модификацию строки в коде.

не бывает Глупых Вопросов

В: Почему компилятор просто не сообщил мне, что я не могу изменять строку?

О: Потому что мы объявили `cards` как `char *`. Компилятор не знал, что эта переменная всегда будет указывать на строковый литерал.

В: Почему строковые литералы размещаются в памяти, которую можно только читать?

О: Потому что они задумывались как константы. Если вы пишете функцию для вывода на экран «Здравствуй, мир», вы не захотите, чтобы другие части программы смогли изменять этот строковый литерал.

В: Во всех ли операционных системах действует правило о неизменяемых строках?

О: В большинстве да. Некоторые версии `gcc` среды Cygwin фактически позволяют вам без проблем изменять строковые литералы. Но этого *никогда* не нужно делать.

В: Что на самом деле означает `const`? Делает ли эта инструкция строку доступной только для чтения?

О: Строковые литералы в любом случае доступны только для чтения. Модификатор `const` означает, что компилятору не понравится, если вы попытаетесь изменить массив с этой конкретной переменной.

В: Всегда ли области памяти размещаются в одном и том же порядке?

О: Они всегда следуют в одном и том же порядке для каждой отдельной операционной системы. Но в разных операционных системах этот порядок может слегка различаться. К примеру, Windows не размещает код в участке памяти с минимальными адресами.

В: Я все еще не могу понять, почему переменная массива не хранится в памяти. Если она существует, значит она где-то находится?

О: Когда программа компилируется, все переменные, ссылающиеся на массив, заменяются адресами самого массива. Получается, что в исполняемом файле переменных массивов уже нет. Это нормально, потому как переменная массива никогда не должна указывать на что-то другое.

В: Копирует ли программа строковый литерал каждый раз, когда я присваиваю его массиву?

О: Это зависит от компилятора. Итоговый машинный код будет либо копировать байты строкового литерала в массив, либо просто устанавливать значения каждого символа в момент объявления.

В: Вы продолжаете использовать слово «объявление». Что оно означает?

О: Объявление — это часть кода, которая объявляет о существовании чего-либо (переменной, функции). Определение — это часть кода, которая определяет, чем является та или иная конструкция. Если вы объявите переменную и присвоите ей значение (например, `int x = 4;`), то это будет и объявление, и определение.

В: Почему у `scanf()` такое название?

О: `scanf()` расшифровывается как scan formatted, так как эту функцию используют для сканирования форматированного ввода.

Как устроена память

Стек
Это область памяти, которая используется для **хранения локальных переменных**. Каждый раз, когда вы вызываете функцию, все ее локальные переменные создаются в стеке. В переводе с английского *stack* — «стопка». Здесь работает тот же принцип, что и со стопкой тарелок, — переменные добавляются при входе в функцию, а при выходе убираются. Странность заключается в том, что стек по сути перевернут вверх дном. Он начинается в верхней части памяти и **«растет» вниз**.

Куча
Эту область памяти мы пока что-то не использовали на практике. Куча предназначена для **динамической памяти** — наборов данных, которые создаются во время работы программы и существуют продолжительное время. Позже мы с ней познакомимся поближе.

Глобальная память
Помните, как мы поместили список песен из музыкального автомата за *пределы* всех функций? В этом и заключается суть глобальной переменной — она существует вне любых функций и доступна для каждой из них. Глобальные переменные создаются при запуске программы, и вы можете свободно их изменять. Но это нежелательно…

Константы
Константы *также* создаются вместе с запуском программы, но они помещаются в **память, предназначенную только для чтения**. Константами могут быть, например, *строковые литералы* — они нужны для работы программы, но никогда не должны изменяться.

Код
И наконец, область кода. Многие операционные системы размещают код в самой нижней части адресного пространства. Эта область также не предназначена для перезаписи. Это часть памяти, куда, в сущности, загружается скомпилированный код.

Максимальный адрес.

СТЕК

КУЧА

ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ

КОНСТАНТЫ

LINK A6, #VARSIZE
MOVE.L SP, SAVESTK(A6)

КОД

Память, предназначенная только для чтения.

Минимальный адрес.

Ваш инструментарий языка Си

Вы изучили главу 2, пополнив свои знания информацией об указателях и памяти. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

С помощью scanf("%i", &x) пользователь может вводить числа напрямую.

Тип int имеет разный размер на разных компьютерах.

&x возвращает адрес x.

&x называется указателем на x.

Переменная-указатель x типа char объявляется как char *x.

Локальные переменные хранятся в стеке.

Строковые литералы хранятся в памяти, предназначенной только для чтения.

Присваивая строку новому массиву, вы копируете ее.

Переменные массивов можно использовать как указатели.

Считывайте содержимое адреса с помощью *a.

fgets(buf, size, stdin) – более простой способ ввода текста.

2.5 Строки

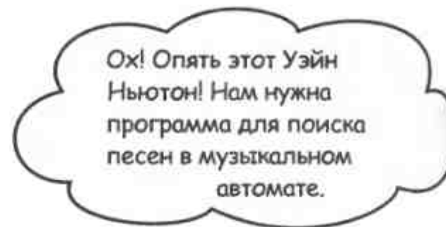
Теория строк

Чтение — это не все, что можно делать со строками.

Вы уже знаете, что строки в Си на самом деле являются *массивами символов*, но что данный язык позволяет с ними *делать*? Вот здесь нам и пригодится библиотека *string.h*. Она входит в состав стандартной библиотеки Си и отвечает за **манипуляции над строками**. Используя *string.h*, вы можете их ***объединять***, ***копировать*** или ***сравнивать***. В этой главе вы узнаете, как создавать **массивы строк**, а затем тщательно рассмотрите процесс ***поиска внутри строк*** с помощью функции `strstr()`.

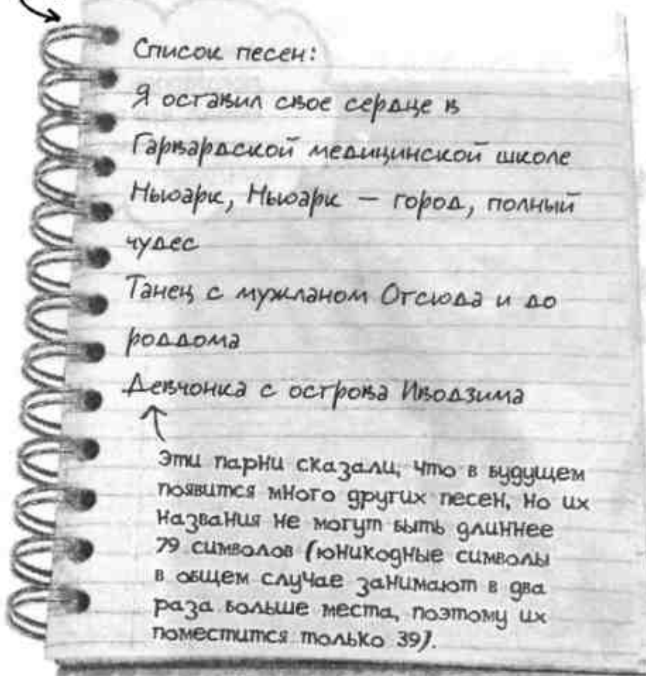
В отчаянных поисках Фрэнка

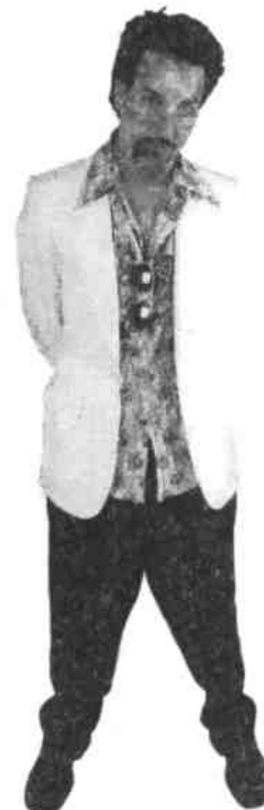
В музыкальных автоматах собрано столько старых песен, что найти какую-то конкретную композицию очень сложно. Чтобы помочь посетителям бара Head First, его работники просят вас написать еще одну программу.



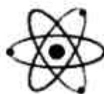
Вот список песен:

Песни из нового альбома под названием «Малоизвестный Синатра».





Сейчас в этом списке всего несколько песен, но он, скорее всего, будет дополнен. От вас требуется написать программу на Си, которая будет спрашивать пользователя, какую песню он ищет, затем выполнять поиск по всему списку и выводить все совпадения.



Сила мозга

В этой программе будет множество строк. Как, по-вашему, можно записать эту информацию на Си?

Создание массива массивов

Вам требуется записать названия некоторого количества песен. В массиве можно создавать сразу несколько записей. Но помните: *каждая строка сама по себе тоже является массивом*. Это значит, что вам нужно создать массив массивов, как этот, например:

Эта первая пара скобок относится к массиву всех строк.

Вторая пара скобок используется для каждой отдельной строки.

Компилятор понимает, что у нас есть пять строк, поэтому писать число между этими скобками не нужно.

```
char tracks[][80] = {
    "Я оставил свое сердце в Гарвардской
    медицинской школе",
    "Ньюарк, Ньюарк — город, полный чудес",
    "Танец с мужланом",
    "Отсюда и до роддома",
    "Девчонка с острова Иводзима",
};
```

Вы знаете, что названия песен никогда не будут длиннее 79 символов (39 символов в случае использования юникода), поэтому укажите здесь значение 80 (39 символов в случае использования юникода).

Каждая строка является массивом, поэтому мы получим массив массивов.

В памяти массив массивов выглядит примерно так:

Для каждого названия песни будет выделено 80 символов (40 символов в случае использования юникода).

Символы внутри строки.

Песни

Я		о	с	т	а	в	и	л		с	в	о	е		с	е	р	д	ц	е
Н	ь	ю	а	р	к	,		Н	ь	ю	а	р	к		-		г	о	р	о
Т	а	н	е	ц		с		м	у	ж	л	а	н	о	м		\0	\0	\0	\0
О	т	с	ю	д	а		и		д	о		р	о	д	д	о	м	а		\0
Д	е	в	ч	о	н	к	а		с		о	с	т	р	о	в	а		И	в ...

tracks[4]

tracks[4][6]

Это значит, что мы сможем получать отдельные названия песен следующим образом:

У этого элемента такое значение.

Это пятая строка. ← Помните, массивы начинаются с нуля.

```
tracks[4]  ──→  "Девчонка с острова Иводзима"
```

Но мы также можем получать отдельные символы каждой строки, если захотим:

```
tracks[4][6]  ──→  'к'
```

← Это седьмой символ в пятой строке.

Теперь мы знаем, как записывать данные в Си. Но что мы с ними будем делать?

Находим строки, содержащие искомый текст

Ребята любезно предоставили вам алгоритм.

Просим пользователя ввести искомый текст.

Перебираем в цикле все названия песен.

Если название песни содержит данный текст, выводим это название.

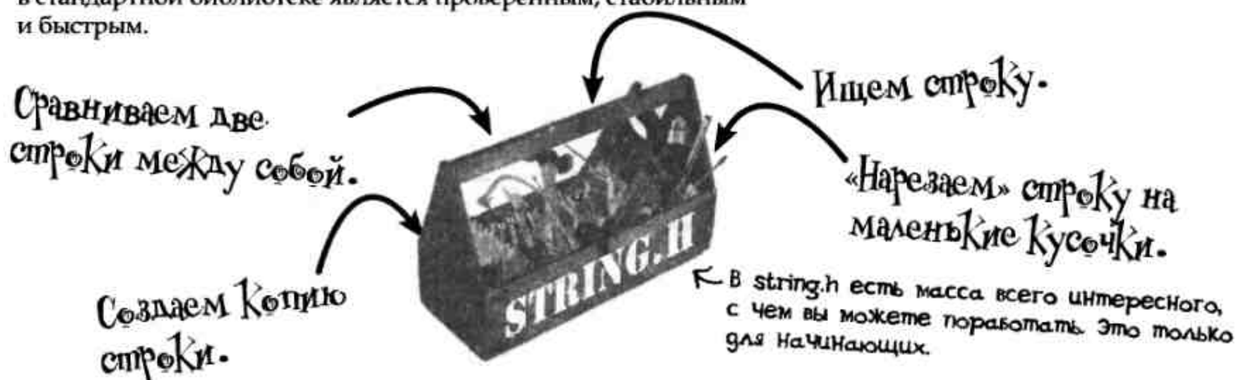
Итак, мы знаем, как записывать песни. Нам также известно, как получить название отдельной песни, поэтому перебрать их в цикле несложно. Мы даже умеем запрашивать у пользователя текст, который нужно искать. Но как узнать, содержит ли название песни нужный нам фрагмент текста?

Использование string.h

Стандартная библиотека языка Си содержит множество кода, который вы получаете совершенно бесплатно при установке компилятора. С помощью библиотечного кода можно открывать файлы, выполнять математические расчеты или управлять памятью. Скорее всего, вы не будете использовать *всю* стандартную библиотеку сразу, поэтому она поделена на несколько разделов, каждый из которых имеет **заголовочный** файл. В заголовочном файле перечисляются все функции конкретного раздела библиотеки.

До сих пор мы по-настоящему использовали только один заголовочный файл — *stdio.h*. Он позволяет работать со стандартными функциями *ввода/вывода*, такими как `printf` и `scanf`.

Стандартная библиотека содержит также код для *обработки строк*. Такая обработка пригодится во многих программах, а код в стандартной библиотеке является проверенным, стабильным и быстрым.



Код из этой библиотеки можно подключать к своей программе с помощью заголовочного файла *string.h*. Добавьте его в верхнюю часть своего кода, как мы это делали с *stdio.h*.

```
#include <stdio.h>
#include <string.h>
```

В своей программе для музыкального автомата мы будем использовать как stdio.h, так и string.h

Попробуйте догадаться

Попробуйте соединить каждую *строковую* функцию с описанием того, что она делает.

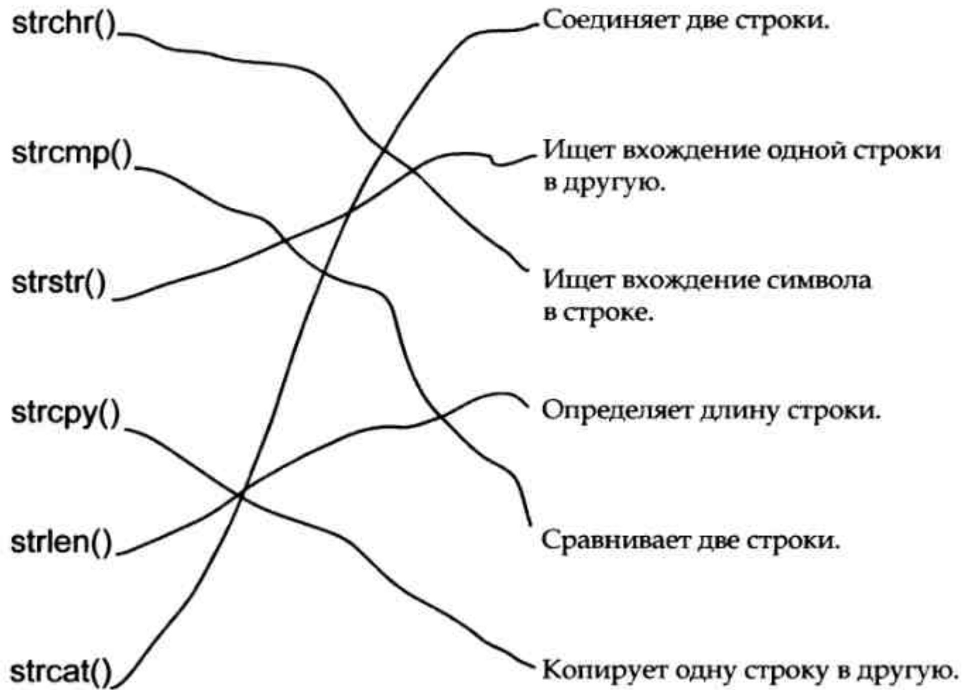
strchr()	Соединяет две строки.
strcmp()	Ищет вхождение одной строки в другую.
strstr()	Ищет вхождение символа в строке.
strcpy()	Определяет длину строки.
strlen()	Сравнивает две строки.
strcat()	Копирует одну строку в другую.

Наточите свой карандаш

Какие из вышеприведенных функций следует использовать в программе для музыкального автомата? Ниже напишите свой ответ.

Попробуйте догадаться решение

От вас требовалось соединить каждую *строковую* функцию с описанием того, что она делает.



Наточите свой карандаш

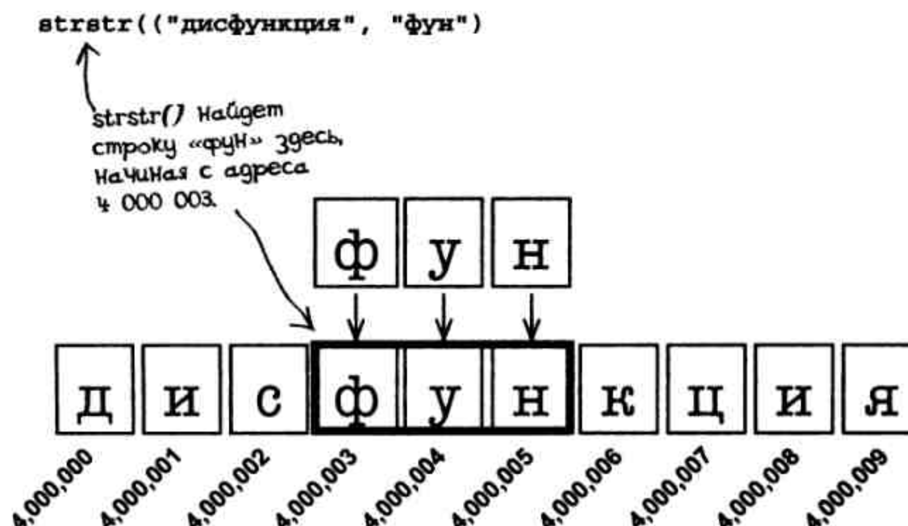
Решение

Вам нужно было написать, какие из вышеприведенных функций следует использовать в программе для музыкального автомата.

strstr

Использование функции strstr

Каким же образом работает функция `strstr()`? Давайте рассмотрим пример. Представим, что нам нужно найти строку «фун» внутри более длинной строки «дисфункция». Мы сделаем вызов наподобие такого:



Функция `strstr()` будет искать *вторую строку в первой*. В случае успеха она вернет адрес, по которому строка находится в памяти. В данном примере функция найдет подстроку «фун» в памяти, начиная с адреса 4 000 003.

Но что если `strstr()` не сможет найти подстроку? Что тогда? В таком случае функция возвращает 0. Как вы думаете, почему так происходит? Если помните, Си воспринимает 0 как *ложь (false)*. Это значит, что вы можете использовать `strstr` для проверки *наличия* одной строки внутри другой. Например:

```
char s0[] = "дисфункция";
char s1[] = "фун";
if (strstr(s0, s1))
  puts("Я нашел фун в дисфункции!");
```

Давайте посмотрим, каким образом можно задействовать strstr() в программе для музыкального автомата.

Головоломка у бассейна

Наши ребята из бара уже начали писать программу для поиска песен в списке. Но несколько страниц с кодом упало в бассейн. Можете ли вы выловить подходящие фрагменты, чтобы завершить функцию поиска? Бассейн давно не чистили, поэтому в нем может плавать код, который совсем не подходит для этой программы.

Примечание: парни использовали несколько новых фрагментов кода, которые они нашли где-то в этой книге.

```
void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if (.............. (..............., ..............))
      printf("Песня номер %i: '%s'\n", .............., ..............);
  }
}
```

Эй, посмотрите, мы создаем отдельную функцию. По-видимому, она будет вызываться внутри функции main(), когда та будет написана.

void всего лишь означает, что функция не вернет значение.

Это цикл for. Чуть позже мы рассмотрим его подробно. На этом этапе вам нужно знать только то, что он выполнит данный фрагмент кода пять раз.

Здесь мы проверяем, содержится ли искомая строка в названии песни.

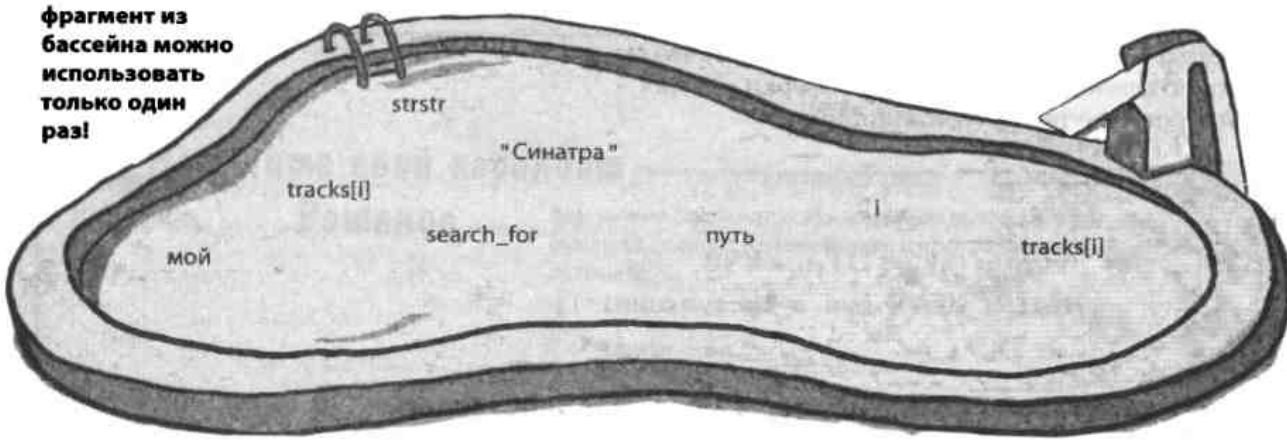
Если в названии песни нашлось совпадение, мы выведем эту строку.

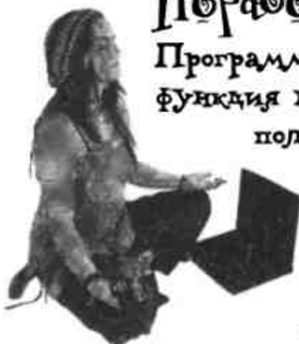
Здесь мы напечатаем два значения.

Одно значение должно быть целым числом.

Другое будет строкой.

Примечание: каждый фрагмент из бассейна можно использовать только один раз!





Поработайте Компилятором

Программе для музыкального автомата нужна функция main(), которая будет считывать пользовательский ввод и вызывать функцию find_track(), рассмотренную на предыдущей странице. Ваша задача — взять на себя роль компилятора и определить, какая из следующих главных функций необходима для нашей программы.

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for) - 1] =
   '\0';
  find_track();
  return 0;
}
```

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 79, stdin);
  search_for[strlen(search_for)
   - 1] = '\0';
  find_track(search_for);
  return 0;
}
```

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for)
   - 1] = '\0';
  find_track(search_for);
  return 0;
}
```

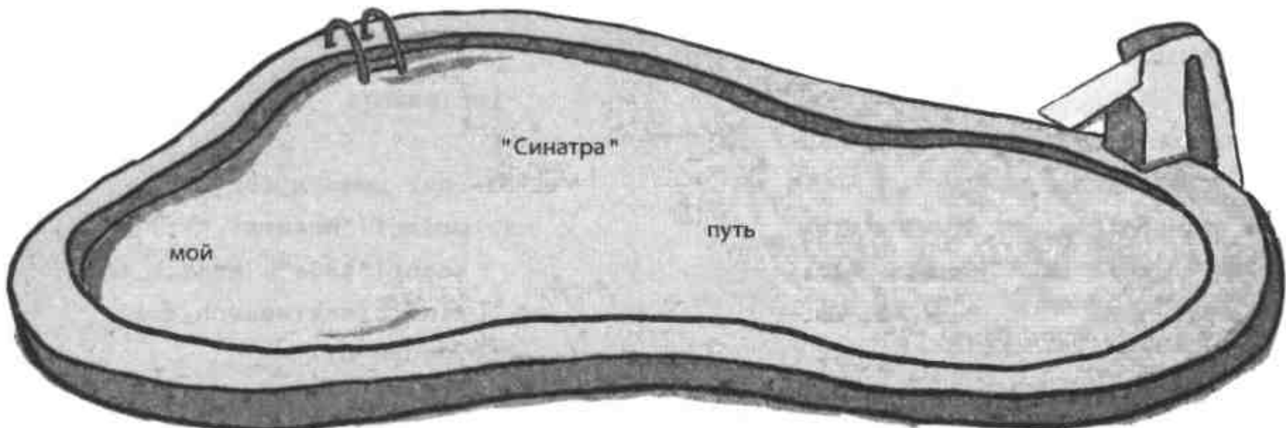
```
int main()
{
  char search_for[80];
  printf("Искать: ");
  scanf("%80s", search_for);
  find_track(search_for);
  return 0;
}
```

Головоломка у бассейна. Решение

Наши ребята из бара уже начали писать программу для поиска песен в списке. Но несколько страниц с кодом упало в бассейн. Вы должны были выловить подходящие фрагменты, чтобы завершить функцию поиска.

Примечание: парни использовали несколько новых фрагментов кода, которые они нашли где-то в этой книге.

```
void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if ( strstr ( tracks[i] , search_for ))
      printf("Песня номер %i: '%s'\n", i , tracks[i] );
  }
}
```



Поработайте Компилятором. Решение

Программе для музыкального автомата нужна функция main(), которая будет считывать пользовательский ввод и вызывать функцию find_track(), рассмотренную на предыдущей странице. Ваша задача состояла в том, чтобы взять на себя роль компилятора и определить, какая из следующих главных функций необходима для нашей программы.

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for) - 1] =
   '\0';
  find_track();
  return 0;
}
```

Функции find_track() при вызове не был передан поисковый запрос.

В этой версии используется не весь массив. Программист уменьшил его длину на единицу, как в случае со scanf().

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 79, stdin);
  search_for[strlen(search_for)
   - 1] = '\0';
  find_track(search_for);
  return 0;
}
```

Это правильная функция main()

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for)
   - 1] = '\0';
  find_track(search_for);
  return 0;
}
```

Эта версия использует scanf() и разрешает пользователю вводить в массив 81 символ (41 символ в случае использования юникода).

```
int main()
{
  char search_for[80];
  printf("Искать: ");
  scanf("%80s", search_for);
  find_track(search_for);
  return 0;
}
```

Пришло время рассмотреть наш код

Давайте скомпонуем код и посмотрим, что у нас получилось.

Вам все еще нужен заголовок stdio.h для функций printf() и scanf().

Вынесем массив tracks за пределы функций main() и find_track(). Таким образом, он будет доступен в любом участке программы.

Это наша новая функция find_track(). Ее необходимо объявить перед тем, как вызывать из функции main().

Этот код выведет все песни, в которых нашлись совпадения.

А это наша функция main() – отправная точка программы.

```
#include <stdio.h>
#include <string.h>

char tracks[][80] = {
  "Я оставил свое сердце в Гарвардской
  медицинской школе",
  "Ньюарк, Ньюарк — город, полный чудес",
  "Танец с мужланом",
  "Отсюда и до роддома",
  "Девчонка с острова Иводзима",
};

void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if (strstr(tracks[i], search_for))
      printf("Песня номер %i: '%s'\n", i, tracks[i]);
  }
}

int main()
{
  char search_for[80];
  printf("Искать: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for) - 1] = '\0';
  find_track(search_for);
  return 0;
}
```

Нам также понадобится заголовок string.h, чтобы выполнять поиск с помощью функции strstr().

i++ означает «увеличить значение i на 1».

Здесь мы запрашиваем текст для поиска.

Теперь мы вызываем нашу новую функцию find_track() и выводим песни, в которых нашлись совпадения.

Порядок, в котором мы собрали код, очень важен. Заголовки подключены вверху, поэтому компилятор будет иметь все нужные функции до того, как скомпилирует наш код. Затем мы объявили массив `tracks` *до* написания функций. Это значит, что мы поместили его в **глобальной области видимости**. Глобальные переменные, такие как `tracks`, существуют за пределами какой-либо конкретной функции, они доступны из любой функции программы.

Наконец, у нас есть еще две функции: `find_track()` и следующая за ней `main()`. Функция `find_track()` должна идти первой, *до* того, как мы вызовем ее из `main()`.

Тест-драйв

Пришло время открыть терминал и проверить, работает ли наш код.

```
File Edit Window Help string.h
> gcc text_search.c -o text_search && ./text_search
Искать: город
Песня номер 1: 'Ньюарк, Ньюарк — город, полный чудес'
>
```

Отличная новость — программа работает!

Несмотря на то что эта программа немного длиннее, чем все написанные нами раньше, в действительности она способна на многое. Она создает массив строк, а затем с помощью библиотеки *string.h* ищет в нем песню, которую запрашивал пользователь.

Уголок ботана

Чтобы получить более подробную информацию о функциях, доступных в *string.h*, посетите страницу *http://tinyurl.com/82acwue*.

Если вы используете операционные системы Mac или Linux, чтобы узнать больше, например, о функции `strstr()` из библиотеки *string.h*, наберите:

```
man strstr
```

не бывает Глупых Вопросов

В: Почему список песен объявлен как `tracks[][80]`? Почему не `tracks[5][80]`?

О: Вы *могли бы* объявить его таким образом, но компилятор способен определить, что в массиве находится пять элементов, поэтому вы можете заменить `[5]` на `[]`.

В: В таком случае почему мы просто не написали `tracks[][]`?

О: Все названия у песен имеют разную длину, поэтому компьютеру нужно сообщить, насколько длинным является каждый элемент массива.

В: Означает ли это, что каждая строка в массиве `tracks` состоит из 80 символов?

О: Программа *выделит* 80 однобайтных (или 40 юникодных) символов для каждой строки, даже если все они значительно короче.

В: Значит, массив `tracks` занимает 80 × 5 = 400 символов в памяти?

О: Да.

В: Что случится, если я забуду подключить заголовочный файл, такой как *string.h*?

О: Некоторые заголовочные файлы компилятор подключит в любом случае, выдав вам предупреждение. В других случаях он просто выведет ошибку компиляции.

В: Зачем мы вынесли объявление массива `tracks` за пределы функций?

О: Мы поместили его в глобальную область видимости. Глобальные переменные могут использоваться любой функцией программы.

В: Сейчас у нас есть две функции. Каким образом компьютер определит, какую из них нужно запускать первой?

О: Программа всегда начинает работу с функции `main()`.

В: Почему я должен размещать функцию `find_track()` перед `main()`?

О: Перед тем как вызвать функцию, язык Си должен знать, какие она принимает параметры и какой тип возвращает.

В: Что случится, если я изменю порядок следования функций?

О: Вы просто получите несколько предупреждений.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Вы можете создать массив символьных массивов `strings[...][...]`.
- Первая пара скобок используется для доступа к внешнему массиву.
- Вторая пара скобок нужна для доступа к данным любого из внутренних массивов.
- Заголовочный файл *string.h* предоставляет вам доступ к функциям из стандартной библиотеки Си для работы со строками.
- В своей программе на Си вы можете создать несколько функций, но первой всегда будет запускаться `main()`.

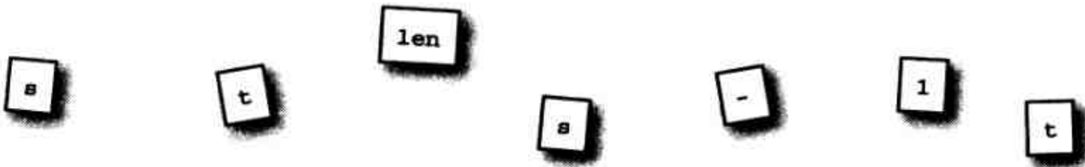
Магнитики с кодом

Ребята работают над новым куском кода для игры. Они создали функцию, которая будет выводить на экран строки в обратном порядке. К сожалению, некоторые магнитики на дверце холодильника сместились. Вы сможете собрать все как было?

```
void print_reverse(char *s)
{
  size_t len = strlen(s);

  char *t = ........... + ........... - 1;

  while ( ........... >= ...........) {
    printf("%c", *t);

    t = ...........  ...........  ...........;
  }
  puts("");
}
```

size_t — это просто целое число, которое хранит размер чего-либо.

Здесь вычисляется длина строки. Например, strlen("ABC")==3.



Магнитики с кодом

Решение

Ребята работают над новым куском кода для игры. Они создали функцию, которая будет выводить на экран строки в обратном порядке. К сожалению, некоторые магнитики на двецрце холодильника сместились. Вам следовало помочь им вернуть все на свои места.

```
void print_reverse(char *s)
{
  size_t len = strlen(s);

  char *t = s + len - 1;

  while ( t >= s ) {
    printf("%c", *t);

    t = t - 1;
  }
  puts("");
}
```

Вычисление адреса, выполняемое в этой строчке, называется адресной арифметикой.

Массив массивов и массив указателей

Вы уже знаете, как хранить наборы строк с помощью массива массивов, но для этого можно использовать и **массив указателей**. Это список адресов в памяти, организованный в виде массива. Массив указателей очень хорошо подходит для быстрого создания списка строковых литералов:

```
char *names_for_dog[] = {"Вобик", "Рекс", "Джим"};
```

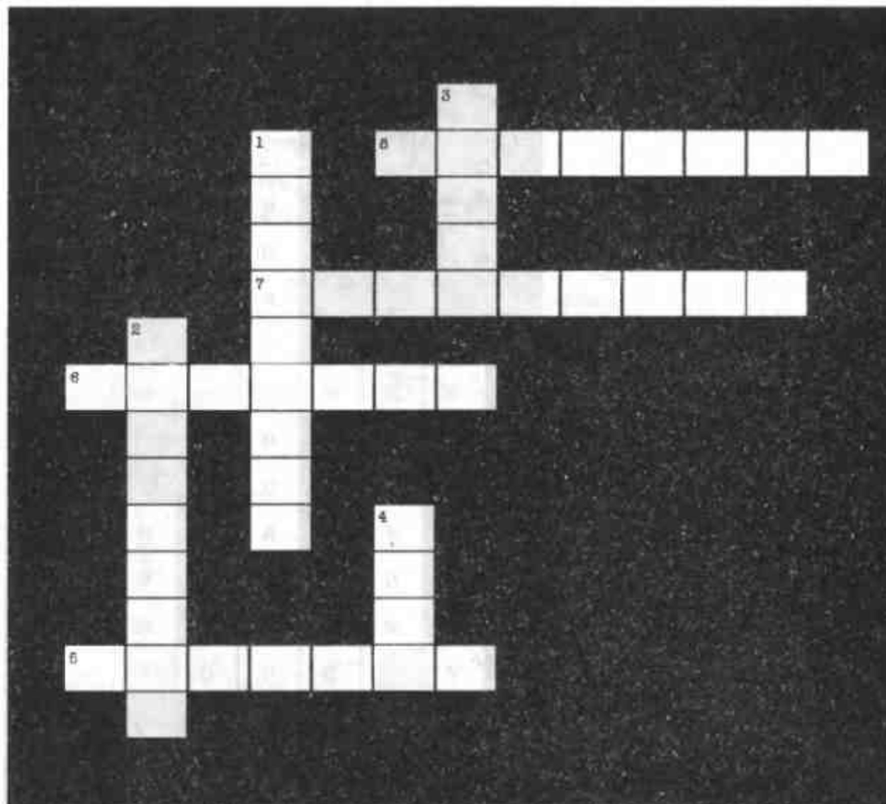
Это массив, который хранит указатели.

На каждый строковый литерал будет ссылаться отдельный указатель.

Доступ к массиву указателей осуществляется точно так же, как в случае с массивом массивов.

Си-Кроссворд

Теперь у ребят появилась работающая функция `print_reverse`, они использовали ее для создания кроссворда. Ответы отображаются в коде в виде выходных строк.



По вертикали

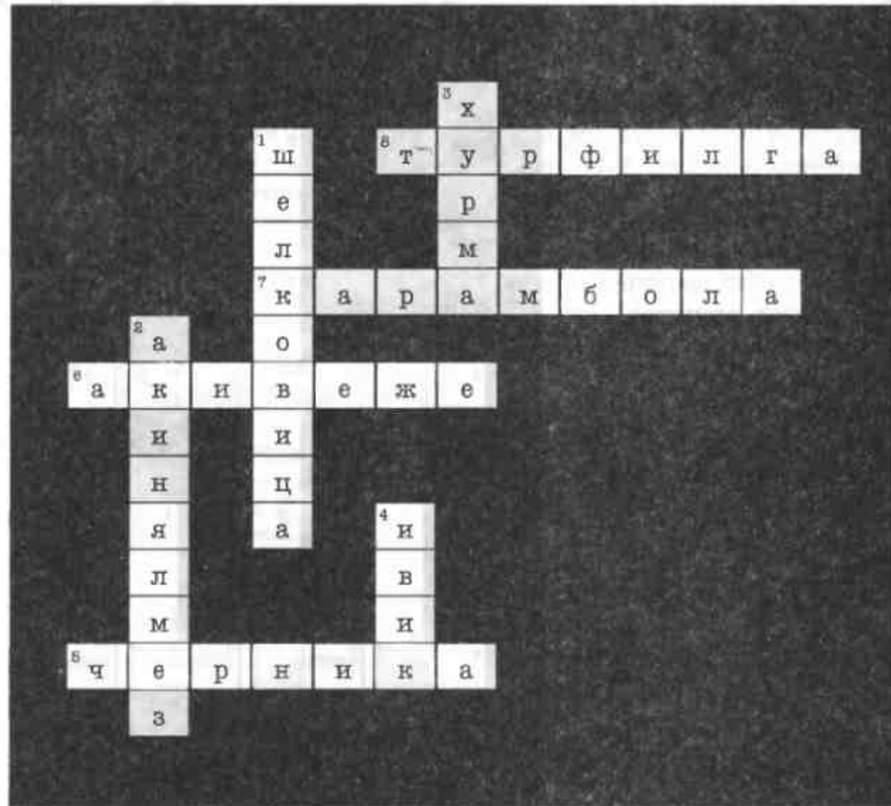
```
int main()
{
  char *juices[] = {
    "питайя", "арбуз", "хурма", "аглифрут",
    "ромовая ягода", "киви", "шелковица", "земляника",
    "черника", "ежевика", "карамбола"
  };
  char *a;
❶ puts(juices[6]);
❷ print_reverse(juices[7]);
  a = juices[2];
  juices[2] = juices[8];
  juices[8] = a;
❸ puts(juices[8]);
❹ print_reverse(juices[(18 + 7) / 5]);
```

По горизонтали

```
❺ puts(juices[2]);
❻ print_reverse(juices[9]);
  juices[1] = juices[3];
❼ puts(juices[10]);
❽ print_reverse(juices[1]);
  return 0;
}
```

Си-Кроссворд Решение

Теперь у ребят появилась работающая функция `print_reverse()`, они использовали ее для создания кроссворда. Ответы отображаются в коде в виде выходных строк.



По вертикали

```
int main()
{
  char *juices[] = {
    "питайя", "арбуз", "хурма", "аглифрут",
    "ромовая ягода", "киви", "шелковица", "земляника",
    "черника", "ежевика", "карамбола"
  };
  char *a;
❶ puts(juices[6]);
❷ print_reverse(juices[7]);
  a = juices[2];
  juices[2] = juices[8];
  juices[8] = a;
❸ puts(juices[8]);
❹ print_reverse(juices[(18 + 7) / 5]);
```

По горизонтали

```
❺ puts(juices[2]);
❻ print_reverse(juices[9]);
  juices[1] = juices[3];
❼ puts(juices[10]);
❽ print_reverse(juices[1]);
  return 0;
}
```

Ваш инструментарий языка Си

Вы изучили главу 2.5, пополнив свои знания информацией о строках. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

Массив строк — это массив массивов.

Заголовок string.h содержит полезные функции для работы со строками.

Массив массивов создается с помощью char strings [...][...].

strcmp() сравнивает две строки.

strstr(a, b) вернет адрес строки b внутри строки a.

strchr() Находит местоположение символа внутри строки.

strcat() объединяет две строки.

strcpy() копирует одну строку в другую.

strlen() Находит длину строки.

3 Создание простых инструментов

Делайте что-то одно, но делайте это хорошо

Любая операционная система включает в себя простые инструменты.

Простые инструменты выполняют **специальные небольшие задачи**, такие как чтение и запись файлов или фильтрация данных. Для более сложных задач вы можете даже *соединить несколько инструментов вместе*. Как же создаются эти простые инструменты? В данной главе вы рассмотрите стандартные блоки, на основе которых они строятся, а также узнаете, как работать с **аргументами командной строки**, как управлять **потоками информации** и **перенаправлять** ее. Не теряйте время, вооружайтесь инструментами.

Простые инструменты могут решать сложные задачи

Простой инструмент — это программа на Си, которая выполняет *одну* задачу и *хорошо с этим справляется*. Она может выводить на экран содержимое файла или список процессов, запущенных на компьютере. Может отображать первые десять строчек файла или отправлять их на принтер. Большинство операционных систем поставляются вместе с целым набором простых инструментов, так называемых утилит, которые можно запустить с помощью командной строки или терминала. Иногда, чтобы выполнить *сложную* задачу, ее можно разбить на ряд *простых*, для решения каждой из которых необходимо написать простой инструмент.

Такие операционные системы, как Linux, состоят преимущественно из сотен и сотен простых инструментов.

Простой инструмент выполняет одну задачу, но делает это хорошо.

Это данные из велосипедного GPS-приемника, представленные в формате CSV.

Это широта. Это долгота.

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
```

Это формат данных, который требуется для карты. Он называется JSON (JavaScript Object Notation).

```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
...
]
```

Данные одни и те же, но формат немного отличается.

Если в какой-то небольшой части программы вам нужно преобразовать данные из одного формата в другой, с этим идеально справится простой инструмент.

Карманный код

Эй, кто из нас не брал с собой распечатанный код в дальнюю поездку и вскоре не обнаруживал... что он совершенно нечитаемый? Конечно, в подобной ситуации были все. Но если немного поразмыслить, то вы вполне способны воссоздать оригинальную версию кода.

Эта программа может считывать данные из командной строки и отображать их в формате JSON. Сможете ли вы заполнить недостающие фрагменты кода?

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = ................;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", ................ , ................ , ................) == 3) {
    if (started)
      printf(",\n");
    else
      started = ................;
    printf("{latitude: %f, longitude: %f, info: '%s'}",............, ..............., ........);
  }
  puts("\n]");
  return 0;
}
```

Мы можем использовать scanf для ввода сразу нескольких фрагментов данных.

Какие значения будут здесь находиться? Не забывайте – scanf() всегда использует указатели.

Функция scanf() возвращает количество значений, которые она смогла прочесть.

Это означает буквально следующее: «Давай мне все символы до тех пор, пока не закончится строка».

Будьте внимательны, присваивая значение переменной started.

Какие значения нужно отобразить?

Карманный код Решение

Эй, кто из нас не брал с собой распечатанный код в дальнюю поездку и вскоре не обнаруживал... что он совершенно нечитаемый? Конечно, в подобной ситуации были все. Но если немного поразмыслить, то вы вполне способны воссоздать оригинальную версию кода.

Эта программа может считывать данные из командной строки и отображать их в формате JSON. Вам нужно было заполнить недостающие фрагменты кода.

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

Начальное значение для переменной started должно быть 0, что означает false.

Не забыли поставить & перед числовыми переменными? Функции scanf() нужны указатели.

Мы будем отображать запятую только в том случае, если уже отобразили предыдущую строчку.

Как только цикл стартовал, мы можем присвоить переменной started значение 1, что означает true.

Здесь нам не нужен оператор &, потому что функция printf() использует значения, а не их адреса.

Тест-драйв

Так что же произойдет, когда мы скомпилируем и запустим этот код? Что он будет делать?

Это данные, которые были выведены на экран.

Это данные, которые ввели вы.

Ввод и вывод перемешались.

```
File Edit Window Help JSON
> ./geo2json
data=[
42.363400,-71.098465,Speed = 21
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'}42.363327,-71.097588,Speed = 23
,
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'}42.363255,-71.096710,Speed = 17
,
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'}42.363182,-71.095833,Speed = 22
,
...
...
...
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'}42.362385,-71.086182,Speed = 21
,
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}^D
]
>
```

Еще несколько часов набора на клавиатуре...

В конце вам нужно нажать CTRL+D, чтобы остановить программу.

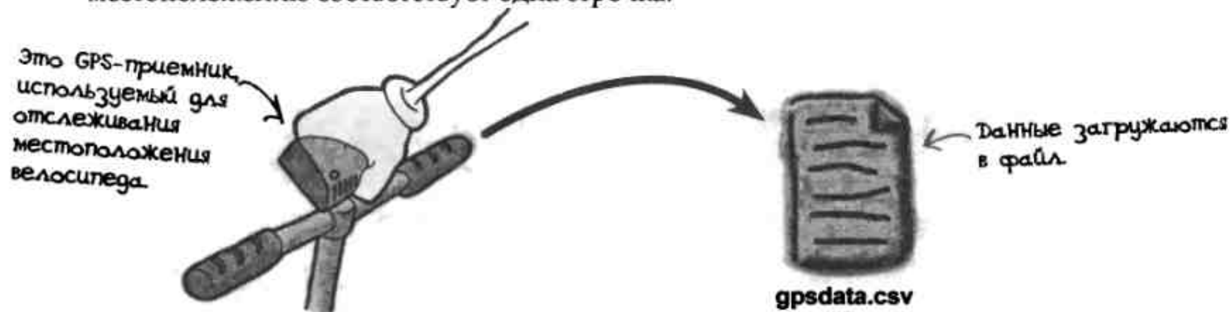
Программа позволяет вводить GPS-данные с помощью клавиатуры, после чего выводит их на экран в формате JSON. Проблема в том, что *ввод и вывод перемешались*. Кроме того, **данных очень много**. При написании простого инструмента вам вряд ли захочется вводить данные вручную — лучше, чтобы весь объем информации считывался из **файла**.

Также непонятно, каким образом будут использоваться данные в формате JSON. Очевидно, что на *экране* от них мало толку.

Итак, хорошо ли работает наша программа? Все ли она делает правильно? **Нужно ли нам изменять код?**

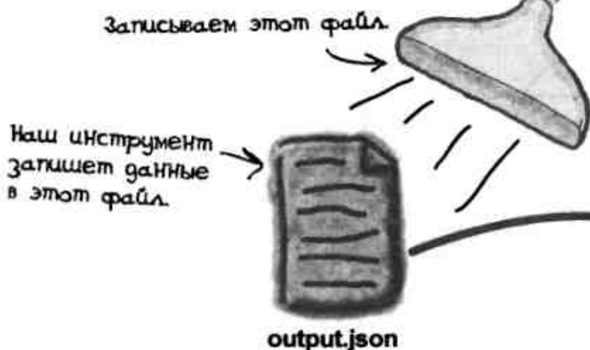
Вот как должна работать эта программа

1 **Берем данные из велосипедного GPS-приемника и загружаем их.** В результате получится файл *gpsdata.csv*, в котором каждому местоположению соответствует одна строчка.



2 **Утилита geo2json нужна для построчного считывания содержимого из файла gpsdata.csv...**

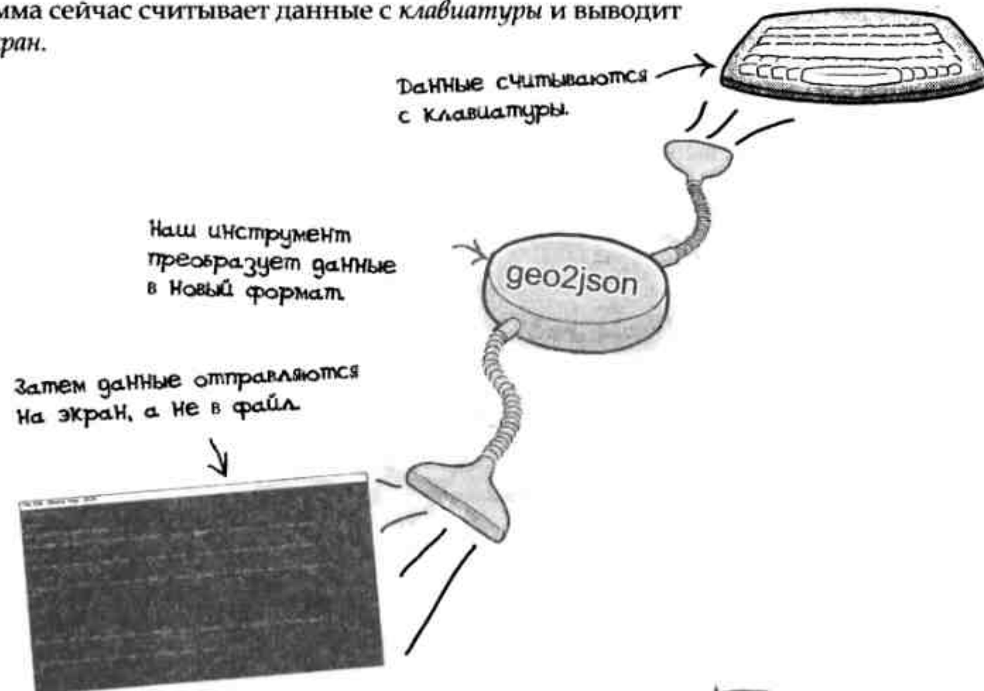
3 **...и последующей записи полученных данных в формате JSON в файл с названием output.json.**



4 **Веб-страница, на которой размещено картографическое приложение, считывает файл output.json.** Таким образом все местоположения наносятся на карту.

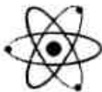
Но мы не используем файлы...

Дело в том, что вместо чтения и записи файлов наша программа сейчас считывает данные с *клавиатуры* и выводит их на *экран*.



Но это не совсем хорошо. Пользователю не захочется вводить все данные самому, ведь они уже хранятся где-то в файле. К тому же если данные в формате JSON выводятся только на экран, то нет никакой возможности нанести их на карту на веб-странице.

Необходимо, чтобы программа работала с **файлами**. Но как этого добиться? Какой код мы должны поменять, если хотим использовать *файлы* вместо клавиатуры и экрана? Да и нужно ли вообще изменять код?



Сила мозга

Можно ли заставить нашу программу использовать файлы, не внося изменения в код? Даже без *перекомпиляции*?

Уголок ботана

Инструменты, которые построчно считывают данные, обрабатывают их и снова записывают, называются **фильтрами**. Если вы работаете с Unix-подобной операционной системой или со средой Cygwin для Windows, то у вас уже есть несколько установленных фильтров:

head — утилита, отображающая несколько первых строк файла;

tail — фильтр для вывода концевых строк файла;

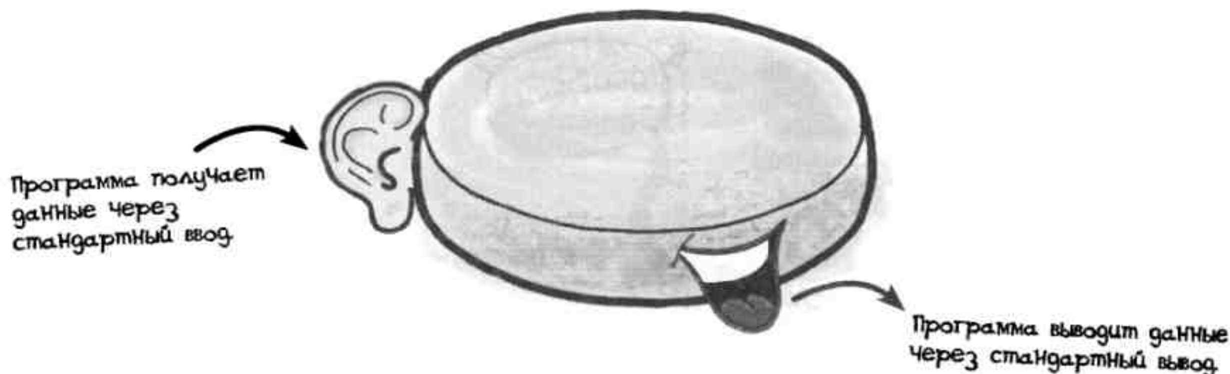
sed — потоковый редактор для поиска и замены текста.

Чуть позже мы узнаем, как можно соединить несколько фильтров в **цепочку**.

Мы можем использовать перенаправление

Функции `scanf()` и `printf()` служат для чтения с клавиатуры и вывода на экран. Но дело в том, что с экраном и клавиатурой обе функции *напрямую* не общаются, а вместо этого используют **стандартные входные и выходные потоки данных**.

Суть терминов кроется в их названии: это потоки данных, которые «текут» в программу или «вытекают» из нее. *Стандартные ввод и вывод* создаются операционной системой во время выполнения программы.



Операционная система управляет тем, как данные попадают в стандартный ввод и куда они следуют через стандартный вывод. Если вы запускаете программу из командной строки или терминала, операционная система будет посылать все набранные на клавиатуре символы в стандартный входной поток. Если операционная система считывает какие-либо данные из стандартного выходного потока, по умолчанию она выводит их на экран.

Функции `scanf()` и `printf()` не знают (и им все равно), откуда приходят данные и куда они идут дальше. Они просто читают стандартный ввод и производят запись в стандартный вывод.

На первый взгляд, все кажется запутанным. В конце концов, почему бы просто не позволить программе работать с клавиатурой и экраном напрямую? Разве это не проще?

Что ж, существует очень веская причина, по которой операционные системы общаются с программами с помощью стандартных ввода и вывода:

вы можете перенаправлять стандартные потоки данных таким образом, чтобы они использовали для чтения и записи нечто другое, например файлы.

Вы можете перенаправлять стандартный ввод с помощью оператора <...

Вместо ввода текста с клавиатуры можно использовать оператор <, чтобы считывать данные из файла.

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
42.363182,-71.095833,Speed = 22
42.363110,-71.094955,Speed = 14
42.363037,-71.094078,Speed = 16
42.362965,-71.093201,Speed = 18
42.362892,-71.092323,Speed = 22
42.362820,-71.091446,Speed = 17
42.362747,-71.090569,Speed = 23
42.362675,-71.089691,Speed = 14
42.362602,-71.088814,Speed = 19
42.362530,-71.087936,Speed = 16
42.362457,-71.087059,Speed = 16
42.362385,-71.086182,Speed = 21
```

Это файл, содержащий данные из GPS-приемника.

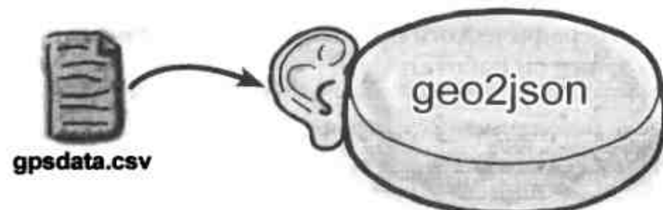
Заставляет операционную систему отправлять данные из файла в стандартный ввод программы.

Нам больше не нужно вводить GPS-данные вручную, поэтому ввод не смешивается с выводом.

```
File Edit Window Help Don'tCrossTheStreams
> ./geo2json < gpsdata.csv
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
...
...
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
>
```

Теперь единственное, что мы видим, это данные в формате JSON, которые выводятся программой.

Оператор < говорит операционной системе, что стандартный ввод программы должен быть соединен не с клавиатурой, а с файлом *gpsdata.csv*. Поэтому мы можем отправлять данные в программу из файла. Теперь нам осталось только перенаправить ее **вывод**.



...а с помощью оператора > вы можете перенаправлять стандартный вывод

Чтобы перенаправить стандартный вывод в файл, нужно применить оператор >:

Сейчас перенаправляем как входной стандартный поток, так и выходной.

```
> ./geo2json < gpsdata.csv > output.json
>
```

Теперь вывод программы будет записан в output.json.

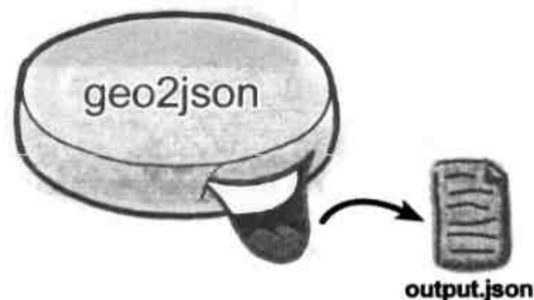
```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
{latitude: 42.362965, longitude: -71.093201, info: 'Speed = 18'},
{latitude: 42.362892, longitude: -71.092323, info: 'Speed = 22'},
{latitude: 42.362820, longitude: -71.091446, info: 'Speed = 17'},
{latitude: 42.362747, longitude: -71.090569, info: 'Speed = 23'},
{latitude: 42.362675, longitude: -71.089691, info: 'Speed = 14'},
{latitude: 42.362602, longitude: -71.088814, info: 'Speed = 19'},
{latitude: 42.362530, longitude: -71.087936, info: 'Speed = 16'},
{latitude: 42.362457, longitude: -71.087059, info: 'Speed = 16'},
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
```

На экране не останется никакого вывода — он целиком уйдет в файл output.json

output.json

Поскольку мы перенаправили данные на стандартный вывод, то на экране мы ничего не увидим. Зато теперь программа создала файл под названием *output.json*.

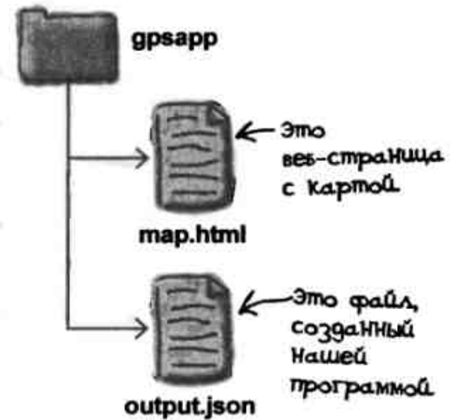
Этот файл как раз и нужен для картографического приложения. Посмотрим, будет ли он работать.



Тест-драйв

Пришло время проверить, поможет ли созданный файл с данными нанести информацию на карту. Скопируем веб-страницу с картографической программой в директорию, где находится файл *output.json*, и откроем ее в браузере:

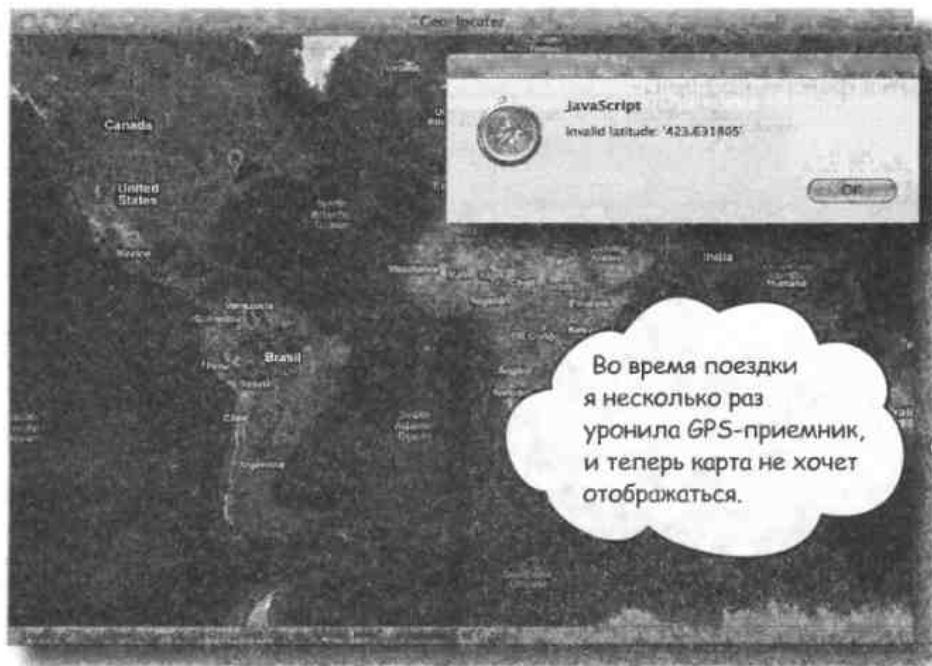
Загрузите веб-страницу по ссылке:
http://oreillyhfc.appspot.com/map.html



Карта работает.
Карта внутри веб-страницы смогла прочитать данные из нашего файла.

С некоторыми данными возникают проблемы

Похоже, наша программа способна считывать данные из GPS-приемника и переводить их в формат, понятный картографическому приложению. Но через несколько дней появилась проблема.



Что же случилось? Проблема в том, что в файле GPS-приемника появились **плохие данные**:

```
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 423.63182, longitude: -71.095833, info: 'Speed = 22'},
```

Десятичный разделитель у этого числа находится не в том месте.

Программа `geo2json` не выполняет никакой проверки считываемых данных — она просто переформатирует числа и отправляет их на стандартный вывод.

Это легко исправить. Нужно всего лишь проверить правильность данных.

Упражнение

Вы должны добавить в программу geo2json некий код, который бы проверял корректность значений широты и долготы. Ничего сверхъестественного. Если широта или долгота выходят за пределы ожидаемого диапазона — просто выводим сообщение об ошибке и завершаем программу со статусом 2:

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
..........................................................................











    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

Если широта < -90 или > 90, то ошибка со статусом 2. Если долгота < -180 или > 180, то ошибка со статусом 2.

Упражнение
Решение

Вы должны были добавить в программу `geo2json` **некий код, который бы проверял корректность значений широты и долготы. Если бы широта или долгота оказались за пределами ожидаемого диапазона, нужно было вывести сообщение об ошибке и завершить программу со статусом 2:**

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    if ((latitude < -90.0) || (latitude > 90.0)) {
      printf("Неправильная широта: %f\n", latitude);
      return 2;
    }
    if ((longitude < -180.0) || (longitude > 180.0)) {
      printf("Неправильная долгота: %f\n", longitude);
      return 2;
    }

    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

В этих строчках проверяется соответствие широты и долготы заданному диапазону.

В этих строчках выводятся простые сообщения об ошибках.

В этих строчках происходит выход из главной функции со статусом ошибки 2.

Тест-драйв

Итак, теперь у нас есть код, который проверяет, находятся ли широта и долгота в заданном диапазоне. Но достаточно ли этого, чтобы наша программа смогла справиться с плохими данными? Давайте посмотрим.

Скомпилируем код и пропустим через программу некорректные данные.

В этой строчке программа скомпилируется заново.

Затем мы снова запускаем программу с плохими данными.

Что за... Где же сообщение об ошибке?

Имелось в виду «Что за неприятность».

```
File Edit Window Help DontCrossTheStreams
> gcc geo2json.c -o geo2json
> ./geo2json < gpsdata.csv > output.json
>
```

Мы сохраним программный вывод в файле output.json

И куда делись все точки?

Хм... странно. Мы добавили код для проверки ошибки, но при запуске программа не *вывела* на экран ничего нового. И теперь на карте нет ни одной точки. Что из этого следует?

Сила мозга

Изучите код. Что, по-вашему, произошло? Выполняет ли код задуманное? Почему не было выведено ни одного сообщения об ошибке? Почему картографическая программа посчитала файл *output.json* неверным?

Разбор кода

Картографической программе не нравится файл *output.json*, так что давайте его откроем и посмотрим, что там внутри:

Это файл output.json

```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
Invalid latitude: 423.631805
```

О, сообщение об ошибке тоже было перенаправлено в выходной файл.

Открыв файл, вы сразу можете видеть, что *в точности* произошло. Программа обнаружила, что с некоторыми фрагментами данных возникла проблема, и сразу же завершила работу. Она не обработала никаких данных, а только *вывела* сообщение об ошибке. Дело в том, что мы **перенаправили стандартный вывод** в файл *output.json*, а это значит, что туда же перенаправилось и сообщение об ошибке. Поэтому программа тихо завершилась, а мы так и не поняли, в чем же была проблема.

Теперь мы *могли* бы проверить статус завершения программы, но на самом деле мы хотим увидеть сообщение об ошибке на экране.

Как же отобразить сообщение об ошибке, если мы перенаправляем вывод?

Уголок ботана

Когда наша программа находит проблему в данных, она завершается со статусом 2. Но как нам проверить этот статус, если программа уже завершилась? Все зависит от того, какую операционную систему вы используете. В Mac, Linux и других разновидностях Unix (или в Cygwin на Windows) вы можете отобразить статус ошибки вот так:

```
File Edit Window Help
$ echo $?
2
```

Если вы используете командную строку в Windows, все будет выглядеть немного иначе:

```
File Edit Window Help
C:\> echo %ERRORLEVEL%
2
```

Обе команды делают одно и то же: они выводят на экран число, которое возвращает программа при завершении своей работы.

Было бы здорово иметь особый выходной поток для ошибок, чтобы они не смешивались со стандартным выводом. Но это всего лишь мечты...

Представляем стандартный поток ошибок

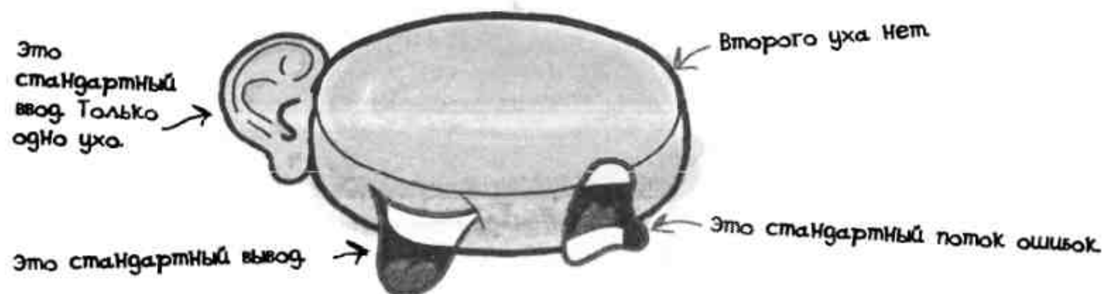
Стандартный вывод является *обычным* способом вывода данных из программы. Но что если случится что-то *непредвиденное*, например ошибка? Вероятно, вам бы хотелось поступать с сообщениями об ошибках немного иначе, чем с обычным выводом.

Вот почему был придуман **стандартный поток ошибок**. Это *второй выходной поток данных*, созданный для отправки сообщений об ошибках.

В основной массе люди имеют два уха и один рот, процессоры устроены немного иначе. У каждого процесса есть **одно ухо** (стандартный ввод) и **два рта** (стандартный вывод и стандартный поток ошибок).

Человек

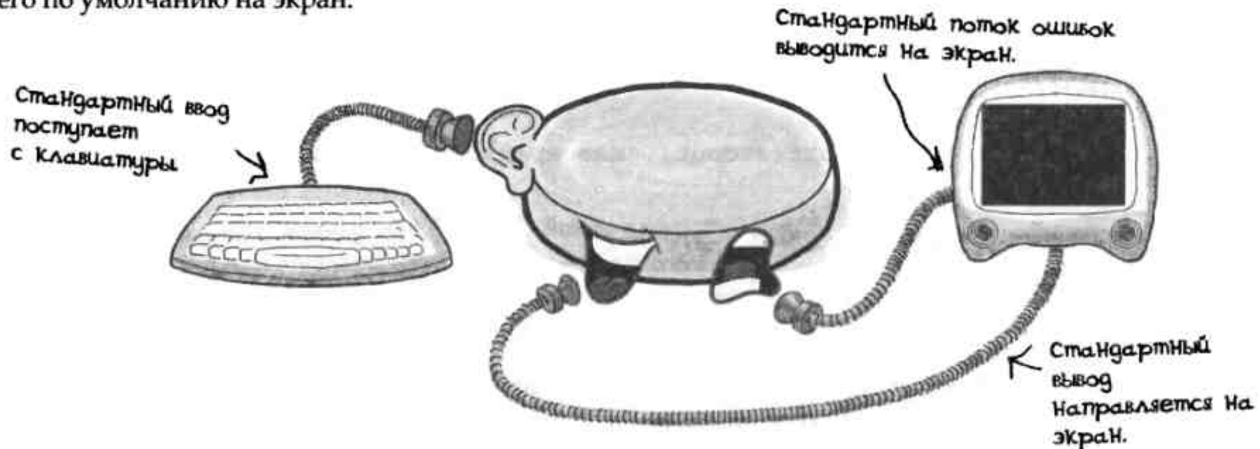
Процесс



Давайте посмотрим, как операционная система обращается с этими потоками данных.

По умолчанию стандартный поток ошибок выводится на экран

Помните, мы говорили, что при создании нового процесса операционная система подключает стандартный ввод к клавиатуре, а стандартный вывод к экрану? Так вот, одновременно с этим она создает и поток данных для ошибок и, как в случае со стандартным выводом, направляет его по умолчанию на экран.



Это означает, что даже если перенаправить входной и выходной потоки так, чтобы они использовали файлы, стандартный поток ошибок будет продолжать отправлять данные на экран.

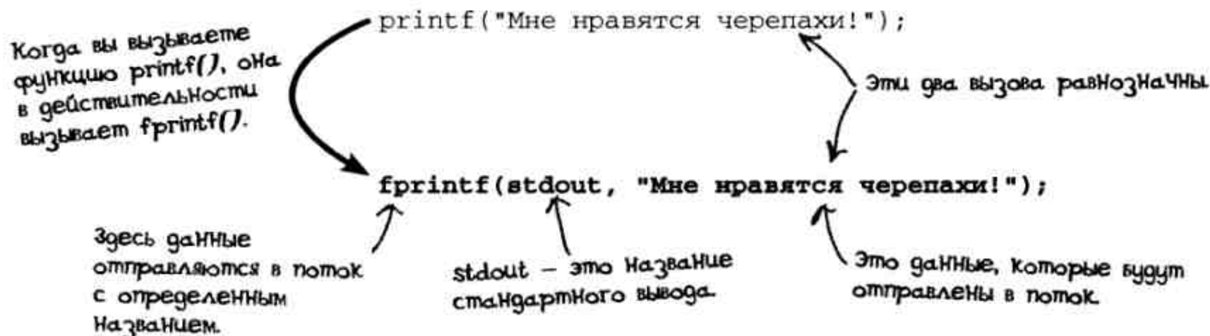
И это здорово, потому что по умолчанию **любые сообщения об ошибках из стандартного потока ошибок будут видны на экране**, даже если стандартный выходной поток перенаправлен в какое-то другое место.

Таким образом, мы можем исправить проблему со скрытыми сообщениями об ошибках, просто выводя их через стандартный поток ошибок.

Но как мы это сделаем?

Функция fprintf() печатает в файловый дескриптор

Мы уже видели, как функция `printf()` отправляет данные в стандартный поток ошибок. Но мы вам *не сказали*, что `printf()` — это всего лишь разновидность более общей функции под названием `fprintf()`:



Функция `fprintf()` позволяет выбрать поток, в который вы хотите отправить текст. Вы можете заставить `fprintf()` печатать в **stdout** (стандартный вывод) или **stderr** (стандартный поток ошибок).

не бывает Глупых Вопросов

В: Есть stdout и stderr. А есть ли stdin?

О: Есть. И, как вы уже, наверное, догадались, он указывает на стандартный ввод.

В: Могу ли я производить печать в него?

О: Нет, в стандартный ввод печатать нельзя.

В: Могу ли я из него что-нибудь читать?

О: Да, используя `fscanf()`. Это то же самое, что и `scanf()`, но с возможностью указать поток данных.

В: То есть fscanf(stdin, ...) — это то же, что и scanf(...)?

О: Да, они идентичны. В действительности `scanf(...)` просто вызывает `fscanf(stdin, ...)`.

В: Могу ли я перенаправить стандартный поток ошибок?

О: Да. Оператор > перенаправляет стандартный вывод, для стандартного потока ошибок используется 2>

В: Значит, я могу написать geo2json 2> errors.txt?

О: Да.

Давайте обновим наш код с учетом использования fprintf

Всего пару изменений, и мы можем сделать так, чтобы сообщения об ошибках выводились в стандартный поток ошибок.

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    if ((latitude < -90.0) || (latitude > 90.0)) {
      printf("Invalid latitude: %f\n", latitude);
      fprintf(stderr, "Неправильная широта: %f\n", latitude);
      return 2;
    }
    if ((longitude < -180.0) || (longitude > 180.0)) {
      printf(stderr, "Invalid longitude: %f\n", longitude);
      fprintf(stderr, "Неправильная долгота: %f\n", longitude);
      return 2;
    }
    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

Мы используем fprintf вместо printf().

В качестве первого параметра нам нужно указать stderr.

Это значит, что теперь код будет работать точно так же, *но* сообщения об ошибках появятся в стандартном потоке ошибок, а не в стандартном выводе.

Давайте запустим код и посмотрим.

Тест-драйв

Если мы перекомпилируем программу и пропустим через нее неправильные данные из GPS-приемника, то вот что случится.

```
File Edit Window Help ControlErrors
> gcc geo2json.c -o geo2json
> ./geo2json-page21  < gpsdata.csv > output.json
Invalid latitude: 423.631805
```

Прекрасно. На этот раз сообщения об ошибках видны на экране, несмотря на то что мы перенаправили стандартный вывод в файл *output.json*.

Стандартный поток ошибок создавался специально для того, чтобы отделить сообщения об ошибках от обычного вывода. Но запомните: `stderr` и `stdout` — это всего лишь выходные потоки данных. Ничто не мешает использовать их в любых других целях.

Давайте проверим на практике приобретенные навыки работы со стандартным выводом и стандартным потоком ошибок.

КЛЮЧЕВЫЕ МОМЕНТЫ

- `printf()` отправляет данные на *стандартный вывод*.
- По умолчанию стандартный вывод идет на экран.
- Вы можете *перенаправлять* стандартный вывод в файл, используя в командной строке оператор **>**.
- `scanf()` считывает данные из *стандартного ввода*.
- По умолчанию данные в стандартный ввод направляются из клавиатуры.
- Вы можете перенаправлять стандартный вывод для чтения файла, используя в командной строке **оператор <**.
- Стандартный поток ошибок зарезервирован для вывода сообщений об ошибках.
- Вы можете перенаправлять стандартный поток ошибок, используя оператор **2>**.

СОВЕРШЕННО СЕКРЕТНО

У нас есть основания полагать, что следующая программа была использована для передачи секретных сообщений:

```
#include <stdio.h>

int main()
{
  char word[10];
  int i = 0;
  while (scanf("%9s", word) == 1) {
    i = i + 1;
    if (i % 2)
      fprintf(stdout, "%s\n", word);
    else
      fprintf(stderr, "%s\n", word);
  }
  return 0;
}
```

i % 2 означает «остаток от деления на 2».

Мы перехватили файл под названием *secret.txt* и клочок бумаги с инструкциями:

ПОДЛОДКА КУПИТЬ
ВСПЛЫВЕТ ШЕСТЬ НА ЯИЦ
ПОВЕРХНОСТЬ И В НЕМНОГО
ДЕВЯТЬ МОЛОКА ВЕЧЕРА

secret.txt

Запустите следующим образом:
secret_messages < secret.txt > message1.txt 2> message2.txt

> перенаправит стандартный вывод.

2> перенаправит стандартный поток ошибок.

Ваша миссия — расшифровать два секретных сообщения. Ниже напишите свои ответы.

Сообщение 1	Сообщение 2
....	....
....	....
....	....
....	....
....	....
....	....
....	....

СОВЕРШЕННО СЕКРЕТНО. РЕШЕНО

У нас есть основания полагать, что следующая программа была использована для передачи секретных сообщений:

```
#include <stdio.h>

int main()
{
  char word[10];
  int i = 0;
  while (scanf("%9s", word) == 1) {
    i = i + 1;
    if (i % 2)
      fprintf(stdout, "%s\n", word);
    else
      fprintf(stderr, "%s\n", word);
  }
  return 0;
}
```

Мы перехватили файл под названием *secret.txt* и клочек бумаги с инструкциями:

```
ПОДЛОДКА КУПИТЬ
ВСПЛЫВЕТ ШЕСТЬ НА ЯИЦ
ПОВЕРХНОСТЬ И В НЕМНОГО
ДЕВЯТЬ МОЛОКА ВЕЧЕРА
```

secret.txt

Запустите следующим образом:
secret_messages < secret.txt > message1.txt 2> message2.txt

Вам нужно было расшифровать два секретных сообщения и ниже записать свои ответы.

Сообщение 1	Сообщение 2
ПОДЛОДКА	КУПИТЬ
ВСПЛЫВЕТ	ШЕСТЬ
НА	ЯИЦ
ПОВЕРХНОСТЬ	И
В	НЕМНОГО
ДЕВЯТЬ	МОЛОКА
ВЕЧЕРА	

Откровение об Операционной Системе

Интервью этой недели:
Важна ли Операционная Система?

Head First: Операционная Система, мы так рады, что сегодня ты смогла найти для нас время.

О/С: Распределение времени — это мой конек.

Head First: Ты согласилась на интервью при условии анонимности, это правда?

О/С: Обойдем этот вопрос стороной. Зови меня просто О/С.

Head First: Имеет ли какое-нибудь значение тип, к которому ты принадлежишь?

О/С: Многие люди очень серьезно относятся к тому, какую операционную систему использовать. Но с точки зрения простых программ на Си мы ведем себя практически одинаково.

Head First: Благодаря стандартной библиотеке Си?

О/С: Да. Если вы пишете на Си, суть не меняется. Как я всегда говорю: если выключить свет, нас не отличишь. Понимаете, о чем я?

Head First: О, конечно. Сейчас ты отвечаешь за загрузку программ в память?

О/С: Я превращаю их в процессы, все верно.

Head First: Важная работа?

О/С: Хотелось бы верить. Вы не можете просто забросить программу в память и надеяться, что она сама все сделает, знаете ли. Необходимо тщательно все подготовить. Мне нужно выделить для нее память, подключить ее к стандартным потокам данных, чтобы она могла взаимодействовать с такими устройствами, как экран и клавиатура.

Head First: То же самое ты недавно сделала и для программы `geo2json`?

О/С: Да. Она далеко пойдет.

Head First: Что ты имеешь ввиду?

О/С: Эта простая текстовая программа — настоящий инструмент.

Head First: А, понятно. Тебе приходится работать со многими инструментами?

О/С: Такова жизнь. Вообще все зависит от операционной системы. Unix-подобные операционные системы используют множество инструментов для выполнения разных задач. Windows делает это не так активно, но для нее они все равно важны.

Head First: Создание простых инструментов, взаимодействующих между собой, — это практически философия, не так ли?

О/С: Это образ жизни. Иногда сложную задачу легче решить, если разбить ее на несколько простых.

Head First: После чего для каждой задачи пишется свой инструмент...

О/С: Именно. Затем с помощью операционной системы (то есть меня) эти инструменты соединяются между собой.

Head First: Есть ли у такого подхода какие-нибудь преимущества?

О/С: Главное преимущество — в простоте. Набор небольших программ легче тестировать. Еще один плюс заключается в том, что написанный однажды инструмент вы можете использовать и в других проектах.

Head First: Что насчет недостатков?

О/С: Ну, такие инструменты с виду не очень хороши. Они, как правило, работают в командной строке, поэтому не обладают тем, что вы называете «внешней привлекательностью».

Head First: Это важно?

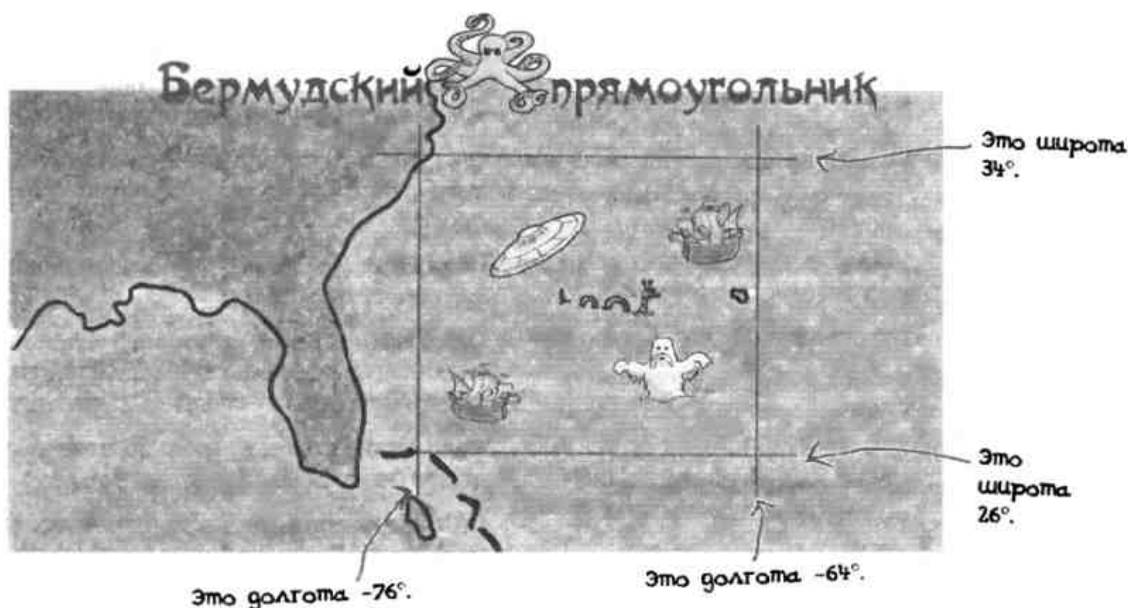
О/С: Не настолько, как может показаться. Если у вас есть набор серьезных инструментов, способных выполнять важную работу, вы всегда можете подключить их к симпатичному интерфейсу, будь то настольное приложение или веб-сайт.

Head First: Ну ладно, спасибо тебе, О/С. Было очень прия…ааааа (зевает)…

Простые инструменты обладают гибкостью

Одно из преимуществ простых инструментов — это их гибкость. Если вы написали программу, которая действительно хорошо выполняет одну задачу, у вас есть все шансы использовать ее во многих других ситуациях. Скажем, если вы создали программу для поиска текста в файле, то вполне возможно, что вы найдете ей не одно применение.

Вспомните нашу утилиту `geo2json`. Мы создали ее, чтобы помочь выводить информацию о велосипедных поездках, правильно? Но ведь ничто не мешает использовать эту программу и в несколько других целях... например, мы можем исследовать ...



Чтобы увидеть, насколько гибок наш инструмент, давайте используем его для решения совершенно другой проблемы. Придумаем нечто более сложное, чем простой вывод координат на карту. Допустим, как и прежде, мы хотим прочитать весь набор GPS-данных, но вместо того чтобы выводить все подряд, давайте отобразим информацию только о тех точках, которые находятся внутри Бермудского прямоугольника.

Это значит, что мы будем выводить только те данные, которые соответствуют следующим условиям:

```
((latitude > 26) && (latitude < 34))

((longitude > -76) && (longitude < -64))
```

С чего же начать?

Не изменяйте утилиту geo2json

Наша утилита `geo2json` выводит всю информацию, которая в нее поступает. Так что же нам делать? Должны ли мы *модифицировать* программу, чтобы помимо *экспорта* данных она их еще и *проверяла*?

Да, мы *могли* бы это сделать. Но не забывайте, что простой инструмент:

выполняет одну задачу и делает это хорошо.

На самом деле нам бы не хотелось изменять утилиту `geo2json`, потому что она должна выполнять только одну задачу. Если мы адаптируем программу для чего-то более сложного, то тем самым создадим проблемы нашим пользователям, которые ожидают, что их инструмент будет работать, как прежде.

Итак, что же делать, если мы не хотим изменять geo2json?

Советы по разработке простых инструментов

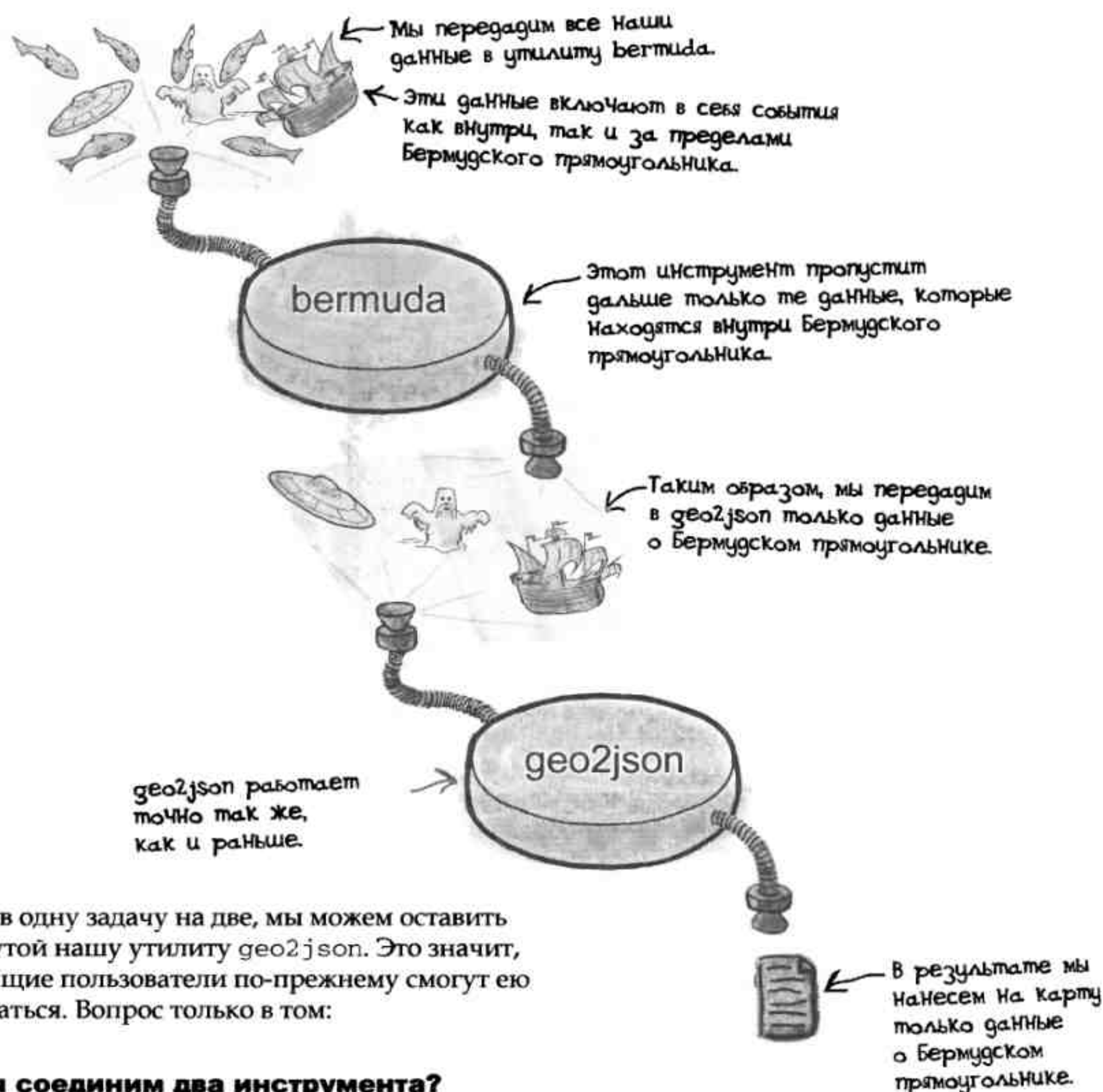
Простые инструменты, как `geo2json`, должны отвечать следующим требованиям.

* Они могут считывать данные из стандартного входного потока.
* Они могут выводить данные на стандартный выходной поток.
* Они работают с *текстовыми* данными, а не с малоизвестными бинарными форматами.
* Каждый из них выполняет *одну простую задачу*.

Для разных задач требуются разные инструменты

Если мы хотим игнорировать данные, которые находятся вне Бермудского прямоугольника, мы должны создать для этого отдельный инструмент.

Итак, у нас будет **два** инструмента: новая утилита **bermuda**, которая отфильтровывает данные, не входящие в Бермудский прямоугольник, и наша первоначальная программа geo2json, которая преобразует оставшуюся информацию для отображения на карте.



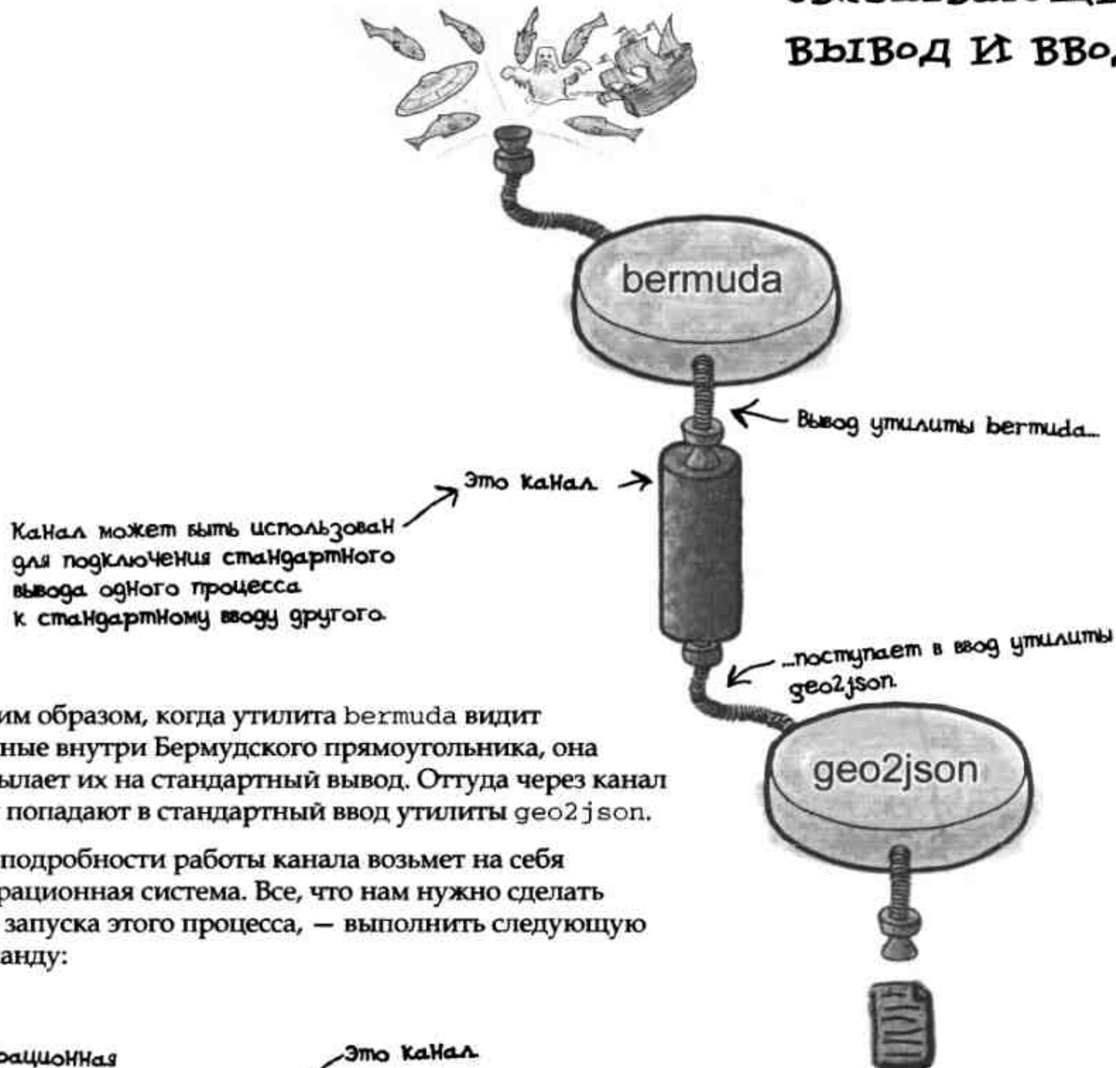
Разделив одну задачу на две, мы можем оставить нетронутой нашу утилиту geo2json. Это значит, что текущие пользователи по-прежнему смогут ею пользоваться. Вопрос только в том:

как мы соединим два инструмента?

Соедините ваш ввод и вывод с помощью канала

Мы уже видели, как можно использовать перенаправление, чтобы подключать *стандартные ввод* и *вывод* к файлам. А теперь мы соединим **стандартный вывод** утилиты **bermuda** со **стандартным вводом geo2json**:

Значок | — это канал, связывающий вывод и ввод.



Таким образом, когда утилита `bermuda` видит данные внутри Бермудского прямоугольника, она посылает их на стандартный вывод. Оттуда через канал они попадают в стандартный ввод утилиты `geo2json`.

Все подробности работы канала возьмет на себя операционная система. Все, что нам нужно сделать для запуска этого процесса, — выполнить следующую команду:

```
bermuda | geo2json
```

Операционная система запустит обе программы одновременно.

Это канал

Вывод утилиты bermuda станет вводом утилиты geo2json.

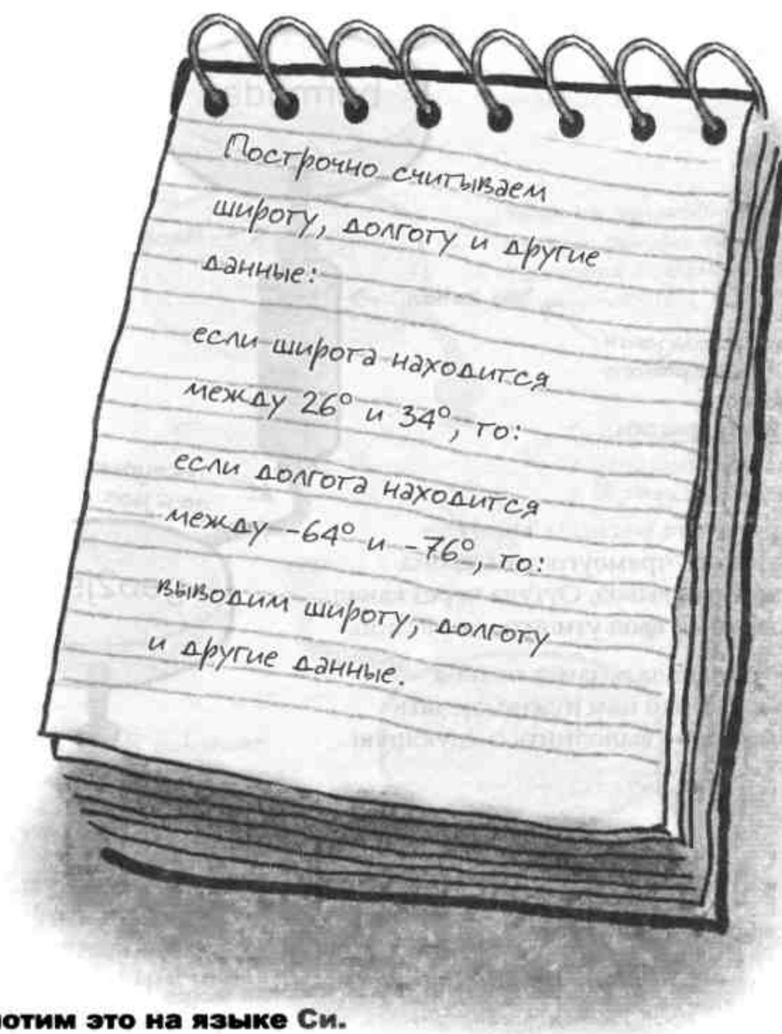
Теперь нам нужно написать утилиту `bermuda`.

Утилита bermuda

Утилита `bermuda` будет работать по тому же принципу, что и `geo2json`: она будет построчно считывать набор данных из GPS-приемника и затем отправлять их на стандартный вывод.

Но у нее есть две существенные особенности. Во-первых, на стандартный вывод `bermuda` будет отправлять не *все* данные, а только те, что находятся внутри Бермудского прямоугольника. Во-вторых, утилита выведет данные все в том же формате CSV, который используется для хранения данных в GPS-приемнике.

Вот как выглядит псевдокод для этого инструмента:



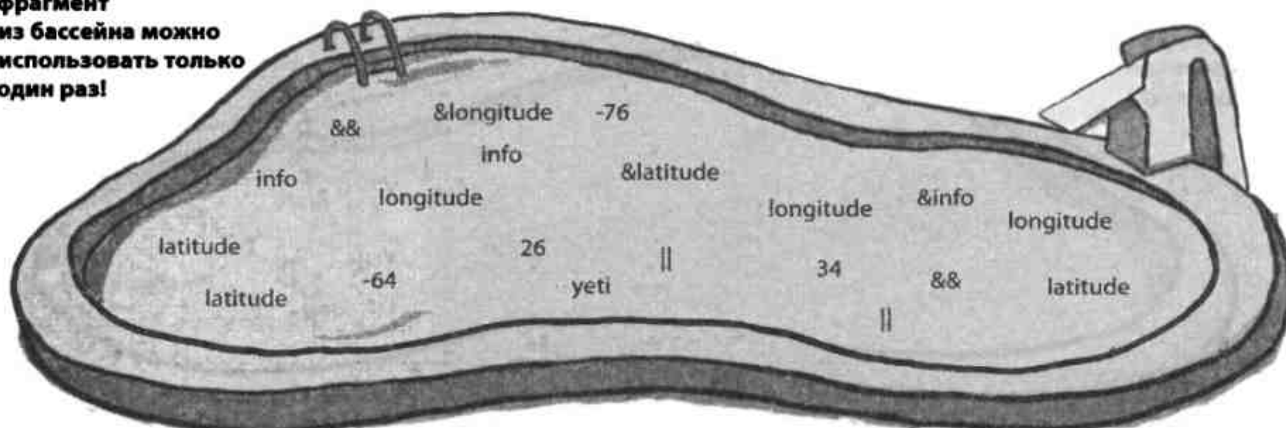
Давайте воплотим это на языке Си.

Головоломка у бассейна

Ваша **задача** — дописать код для программы bermuda. Используйте фрагменты кода из бассейна для заполнения пустых строк снизу. Учтите, вам пригодятся не все фрагменты.

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  while (scanf("%f,%f,%79[^\n]",.................. ,.................. ,..................) == 3)
    if ((.................. >..................) .................. (..................< ..................))
      if ((.................. >..................) .................. (..................< ..................))
        printf("%f,%f,%s\n",.................. ,.................. ,..................);

  return 0;
}
```

Примечание: каждый фрагмент из бассейна можно использовать только один раз!

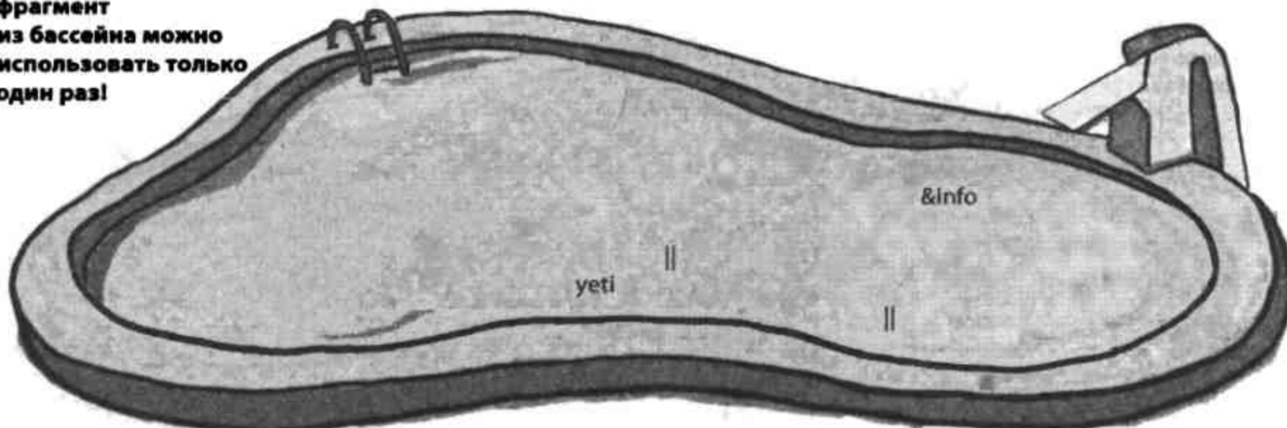


Головоломка у бассейна. Решение

Ваша **задача** состояла в том, чтобы дописать код для программы bermuda, используя фрагменты кода из бассейна для заполнения пустых строк снизу.

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3)
    if ((latitude > 26) && (latitude < 34))
      if ((longitude > -76) && (longitude < -64))
        printf("%f,%f,%s\n", latitude, longitude, info);

  return 0;
}
```

Примечание: каждый фрагмент из бассейна можно использовать только один раз!



Тест-драйв

Сделайте это!

Теперь, когда мы закончили писать утилиту bermuda, самое время проверить, как она работает в сочетании с geo2json. Посмотрим, сможем ли мы обнаружить внутри Бермудского прямоугольника что-нибудь странное.

Вы можете загрузить файл *spooky.csv* по адресу: *http://oreillyhfc.appspot.com/spooky.csv*

Скомпилировав оба инструмента, мы можем запустить консоль и выполнить две программы одновременно с помощью команды:

```
(./bermuda | ./geo2json) < spooky.csv > output.json
```

Не забывайте: если вы работаете в операционной системе Windows, вам не нужно писать ./.

Соединив две утилиты вместе, вы можете обращаться с ними, как с единой программой.

Это канал, соединяющий процессы.

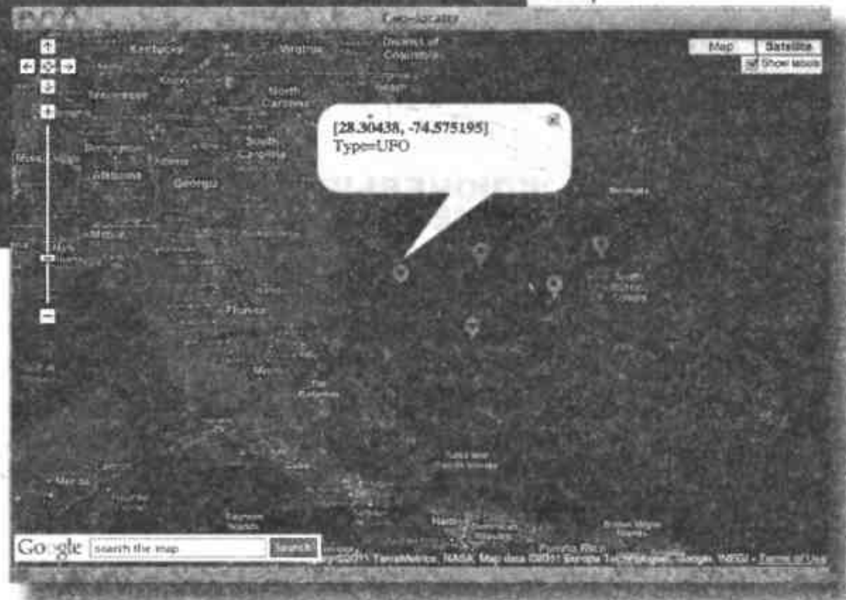
Это файл, содержащий все события.

Утилита bermuda отфильтровывает события, которые мы хотим игнорировать.

Утилита geo2json преобразует события в формат JSON.

Сохраняем вывод в этот файл.

Два инструмента, соединенные каналом, работают как одна программа. Таким образом, мы можем перенаправлять стандартные ввод и вывод так же, как делали это ранее.



Замечательно, программа работает!

не бывает Глупых Вопросов

В: Почему так важно, чтобы простые инструменты использовали стандартные потоки для ввода и вывода?

О: Потому что использование стандартных потоков облегчает соединение инструментов с помощью каналов.

В: Почему это так важно?

О: Как правило, простые инструменты не решают в одиночку всю проблему целиком, а берут на себя небольшую техническую задачу, например преобразование данных из одного формата в другой. Но, соединив их вместе, вы можете решать серьезные проблемы.

В: Чем на самом деле является канал?

О: Все подробности скрыты в недрах операционной системы. Каналы могут создаваться из фрагментов памяти или временных файлов. Главное, что данные последовательно принимаются на одном конце и отправляются на другом.

В: Если две программы соединены каналом, должна ли первая из них закончить свою работу, прежде чем вторая сможет начать?

О: Нет. Обе программы будут работать одновременно. Когда первая будет выводить данные, вторая сможет их принимать.

В: Почему в простых инструментах используется текст?

О: Это наиболее открытый формат. Если простой инструмент использует текст, то любой программист сможет легко прочитать и понять вывод программы с помощью текстового редактора. Бинарные форматы обычно запутанны и сложны для понимания.

В: Могу ли я соединить с помощью канала несколько программ?

О: Да, просто добавьте оператор | между именами программ. Набор соединенных между собой процессов называется *конвейером*.

В: Допустим, несколько процессов соединены между собой с помощью каналов. У каких именно процессов я перенаправлю стандартные ввод и вывод, если применю операторы > и <?

О: Оператор < отправит содержимое файла к первому процессу в конвейере. Оператор > перехватит стандартный вывод у последнего процесса в конвейере.

В: Действительно ли нужны скобки, когда я запускаю программу `bermuda` с `geo2json`?

О: Да. Они нужны для уверенности, что файл с данными прочитан стандартным потоком ввода программы `bermuda`.

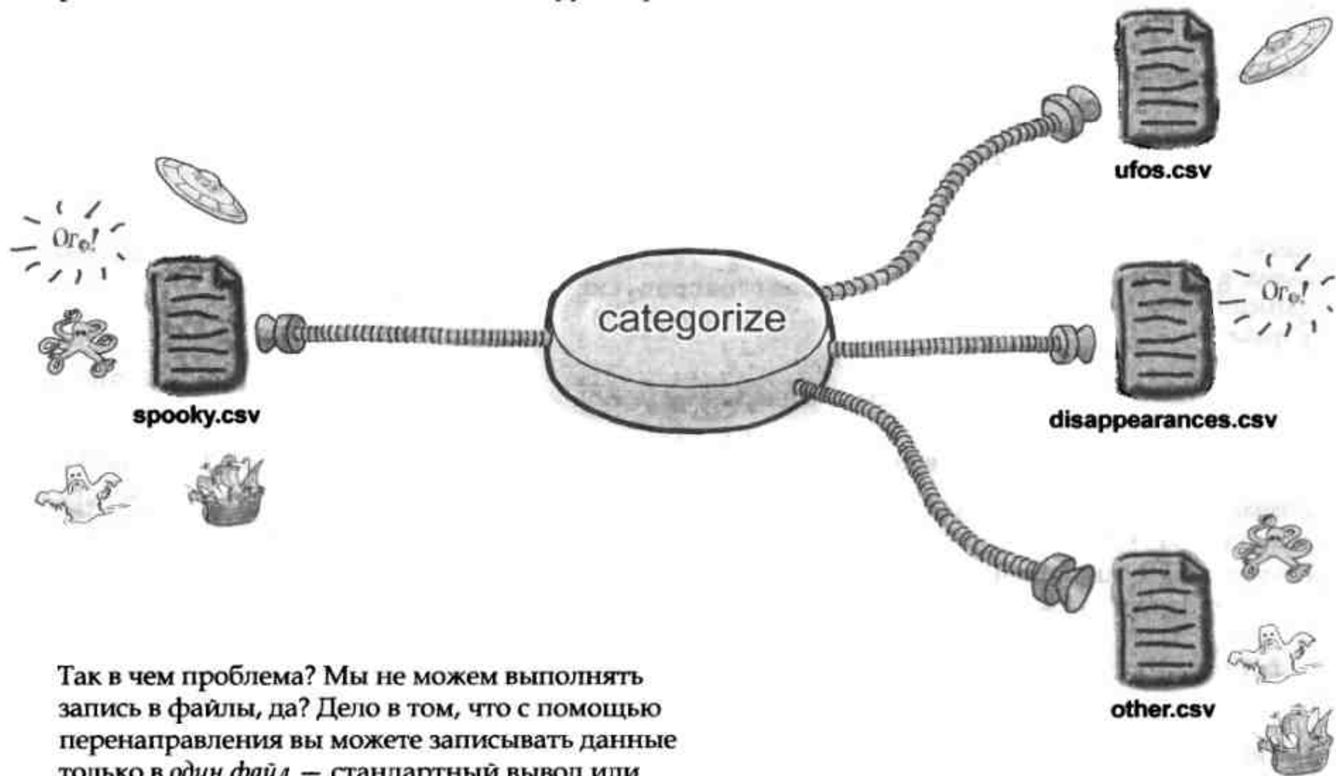
КЛЮЧЕВЫЕ МОМЕНТЫ

- Если вы хотите выполнить особую задачу, подумайте над тем, чтобы написать для нее отдельный простой инструмент.
- Разрабатывайте инструменты таким образом, чтобы они работали со стандартными вводом и выводом.
- Простые инструменты, как правило, считывают и записывают текстовые данные.
- С помощью **канала** вы можете подключить стандартный вывод одного процесса к стандартному вводу другого.

Но что если мы хотим вывести данные в несколько файлов?

Итак, мы узнали, как считывать данные из одного файла и записывать их в другой с помощью перенаправления. Но что если программа должна делать нечто более сложное, например отправлять данные в **более чем один файл**?

Представьте, что вам необходимо создать еще один инструмент, который будет считывать данные из одного файла, а затем разбивать их на части и записывать в другие файлы.



Так в чем проблема? Мы не можем выполнять запись в файлы, да? Дело в том, что с помощью перенаправления вы можете записывать данные только в *один файл* — стандартный вывод или стандартный поток ошибок. Как же быть?

Создайте собственный поток данных

При запуске программы операционная система снабжает ее тремя потоками данных: стандартным вводом, стандартным выводом и стандартным потоком ошибок. Но иногда вам необходимо создавать и другие потоки данных прямо на лету.

К счастью, операционная система не ограничивает вас теми потоками данных, которые предоставляются при запуске приложения. Вы можете создать свой собственный поток во время работы программы.

Каждый поток данных создается с помощью функции **fopen()** и представлен указателем на файл (FILE):

Здесь создается поток для чтения данных из файла → Это имя файла → Это режим, «r» значит read («считывать»).

```
FILE *in_file = fopen("input.txt", "r");
```

Здесь создается поток для записи данных в файл → Это имя файла → Это режим, «w» значит write («записывать»).

```
FILE *out_file = fopen("output.txt", "w");
```

У функции fopen() **два** параметра — *имя файла* и *режим*. Режимы могут быть следующими: «**w**» (запись в файл), «**r**» (чтение из файла) и «**a**» (добавление данных в конец файла).

Создав поток данных, вы можете производить в него печать точно так же, как раньше, с помощью функции **fprintf()**. Но что если нужно прочитать данные из файла? В этом вам поможет функция **fscanf()**:

Режимы для потоков данных:
«w» = write («записывать»),
«r» = read («считывать»),
«a» = append («добавлять»).

```
fprintf(out_file, "Не одевайте %sв сочетании с %s", "красным", "зеленым");
```

```
fscanf(in_file, "%79[^\n]\n", sentence);
```

Закончив работу с потоком данных, вам нужно *его закрыть*. На самом деле все подобные потоки закрываются автоматически при завершении программы, но все же лучше делать это вручную:

```
fclose(in_file);
fclose(out_file);
```

Теперь давайте применим это на практике.

Наточите свой карандаш

Это код программы, которая считывает все данные из файла GPS, после чего записывает их в один из трех других файлов. Попробуйте заполнить недостающие фрагменты:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  char line[80];
  FILE *in = fopen("spooky.csv",  ..............);
  FILE *file1 = fopen("ufos.csv", ................ );
  FILE *file2 = fopen("disappearances.csv", ................ );
  FILE *file3 = fopen("others.csv", ................ );
  while ( ................ (in, "%79[^\n]\n", line) == 1) {
    if (strstr(line, "НЛО"))
      ................ (file1, "%s\n", line);
    else if (strstr(line, "Исчезновение"))
      ................ (file2, "%s\n", line);
    else
      ................ (file3, "%s\n", line);
  }
  ................(file1);
  ................(file2);
  ................(file3);
  return 0;
}
```

не бывает Глупых Вопросов

В: Сколько потоков данных я могу открыть?

О: Это зависит от операционной системы, но обычно процесс может иметь до 256 потоков. Ключевой момент состоит в том, что их число ограниченно, поэтому после завершения работы с потоком убедитесь, что он закрыт.

В: Почему `FILE` пишется большими буквами?

О: Так исторически сложилось. Когда-то структура `FILE` была определена в виде макроса. А названия макросов обычно пишутся большими буквами. Позже мы об этом поговорим.

Наточите свой карандаш
Решение

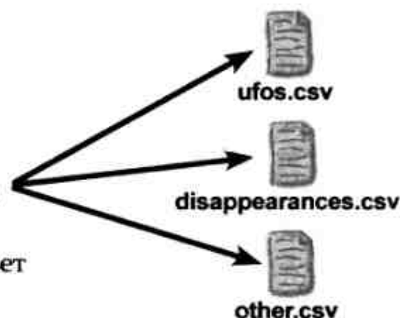
Это код программы, которая считывает все данные из файла GPS, после чего записывает их в один из трех других файлов. Вам нужно было заполнить недостающие фрагменты.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  char line[80];
  FILE *in = fopen("spooky.csv", ..."r"...);
  FILE *file1 = fopen("ufos.csv", ..."w"... );
  FILE *file2 = fopen("disappearances.csv", ..."w"... );
  FILE *file3 = fopen("others.csv", ..."w"... );
  while (..fscanf... (in, "%79[^\n]\n", line) == 1) {
    if (strstr(line, "НЛО"))
      ...fprintf... (file1, "%s\n", line);
    else if (strstr(line, "Исчезновение"))
      ...fprintf... (file2, "%s\n", line);
    else
      ...fprintf... (file3, "%s\n", line);
  }
  ...fclose...(file1);
  ...fclose...(file2);
  ...fclose...(file3);
  return 0;
}
```

Программа запускается, но...

Если вы скомпилируете и запустите программу с помощью:

```
gcc categorize.c -o categorize && ./categorize
```



она строка за строкой прочитает файл *gpsdata.csv* и разобьет данные на три других файла: *ufos.csv*, *disappearances.csv* и *other.csv*.

Это все замечательно, но что если пользователь захотел разделить данные по-другому? Что если он захотел производить поиск по другим словам или делать запись в другие файлы? Возможно ли это без необходимости каждый раз компилировать программу заново?

У функции main() есть кое-что, о чем мы пока не говорили

При написании любой программы вы должны предоставлять пользователю возможность управления ее работой. Программе с графическим интерфейсом, вероятно, понадобится управление настройками. Консольная утилита, такая как `categorize`, должна позволять пользователю передавать ей **аргументы командной строки**:

Это первое слово, которое нужно фильтровать.

Все данные о русалочках будут сохранены в этом файле.

Это означает, что мы хотим искать слово «Элвис».

```
./categorize русалочка mermaid.csv Элвис elvises.csv the_rest.csv
```

Все упоминания об Элвисе будут сохранены здесь.

Все остальное уходит в этот файл.

Но как нам прочитать аргументы командной строки **внутри программы**? Все функции **main()**, которые мы создавали до сих пор, не содержали никаких аргументов. Но дело в том, что мы можем использовать *две* формы главной функции. Вот вторая форма:

```
int main(int argc, char *argv[])
{
    .... Делаем что-то ....
}
```

Функция `main()` может принимать аргументы командной строки в виде **массива строк**. По сути это *массив символьных указателей на строки*, поскольку в языке Си нет встроенных строк. Выглядит это так:

```
"./categorize" "русалочка" "mermaid.csv" "Элвис" "elvises.csv" "the_rest.csv"
```

Это argv[0]. Это argv[1]. Это argv[2]. Это argv[3]. Это argv[4]. Это argv[5].

В качестве первого аргумента на самом деле выступает имя запускаемой программы.

Как и с любым другим массивом в Си, нам нужен способ узнавать его длину. Вот почему у функции `main()` есть два параметра. Значение `argc` — это количество элементов в массиве.

Благодаря аргументам командной строки ваш код становится более гибким, поэтому стоит подумать над тем, как, по вашему мнению, пользователь мог бы *повлиять* на выполнение программы. Это сделает вашу программу более ценной

Будьте осторожны!

Первый аргумент содержит имя программы, запущенной пользователем.

*Это значит, что первый **настоящий** аргумент командной строки —* `argv[1]`.

Хорошо, давайте посмотрим, как сделать программу categorize чуть более гибкой.

Магнитики с кодом

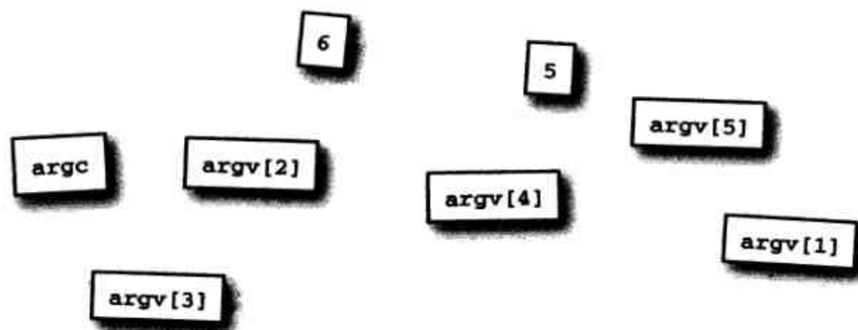
Это модифицированная версия программы `categorize`, **которая может считывать из командной строки ключевые слова для поиска и файлы для записи. Посмотрим, сможете ли вы расставить магнитики на подходящие места.**

Программа запускается с помощью

```
./categorize русалочка mermaid.csv Элвис elvises.csv the_rest.csv
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char line[80];

  if ( .................. != .................. ) {
    fprintf(stderr, "Вы должны передать 5 аргументов\n");
    return 1;
  }
  FILE *in = fopen("spooky.csv", "r");

  FILE *file1 = fopen( .................. , "w");

  FILE *file2 = fopen( .................. , "w");

  FILE *file3 = fopen( .................. , "w");
```

```
  while (fscanf(in, "%79[^\n]\n", line) == 1) {

    if (strstr(line, .................. ))
      fprintf(file1, "%s\n", line);

    else if (strstr(line, .................. ))
      fprintf(file2, "%s\n", line);
    else
      fprintf(file3, "%s\n", line);
  }
  fclose(file1);
  fclose(file2);
  fclose(file3);
  return 0;
}
```



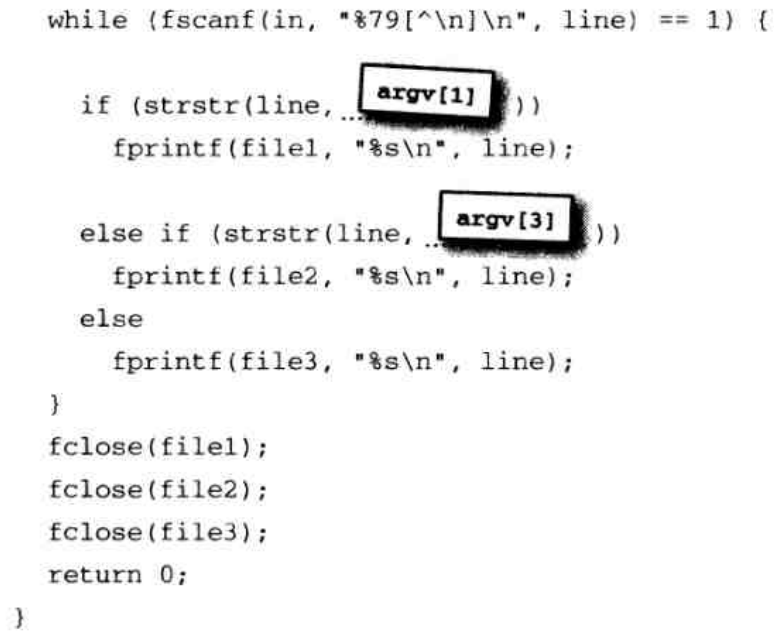
Магнитики с кодом Решение

Это модифицированная версия программы `categorize`, которая может считывать из командной строки ключевые слова для поиска и файлы для записи. Вам нужно было расставить магнитики на подходящие места.

Программа запускается с помощью

`./categorize русалочка mermaid.csv Элвис elvises.csv the_rest.csv`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char line[80];

  if ( argc != 6 ) {
    fprintf(stderr, "Вы должны передать 5 аргументов\n");
    return 1;
  }
  FILE *in = fopen("spooky.csv", "r");

  FILE *file1 = fopen( argv[2] , "w");

  FILE *file2 = fopen( argv[4] , "w");

  FILE *file3 = fopen( argv[5] , "w");
```



```
  while (fscanf(in, "%79[^\n]\n", line) == 1) {

    if (strstr(line, argv[1]))
      fprintf(file1, "%s\n", line);

    else if (strstr(line, argv[3]))
      fprintf(file2, "%s\n", line);
    else
      fprintf(file3, "%s\n", line);
  }
  fclose(file1);
  fclose(file2);
  fclose(file3);
  return 0;
}
```

5

Тест-драйв

Хорошо, давайте опробуем новую версию нашего кода. Нам нужен файл с тестовыми данными под названием *spooky.csv*.

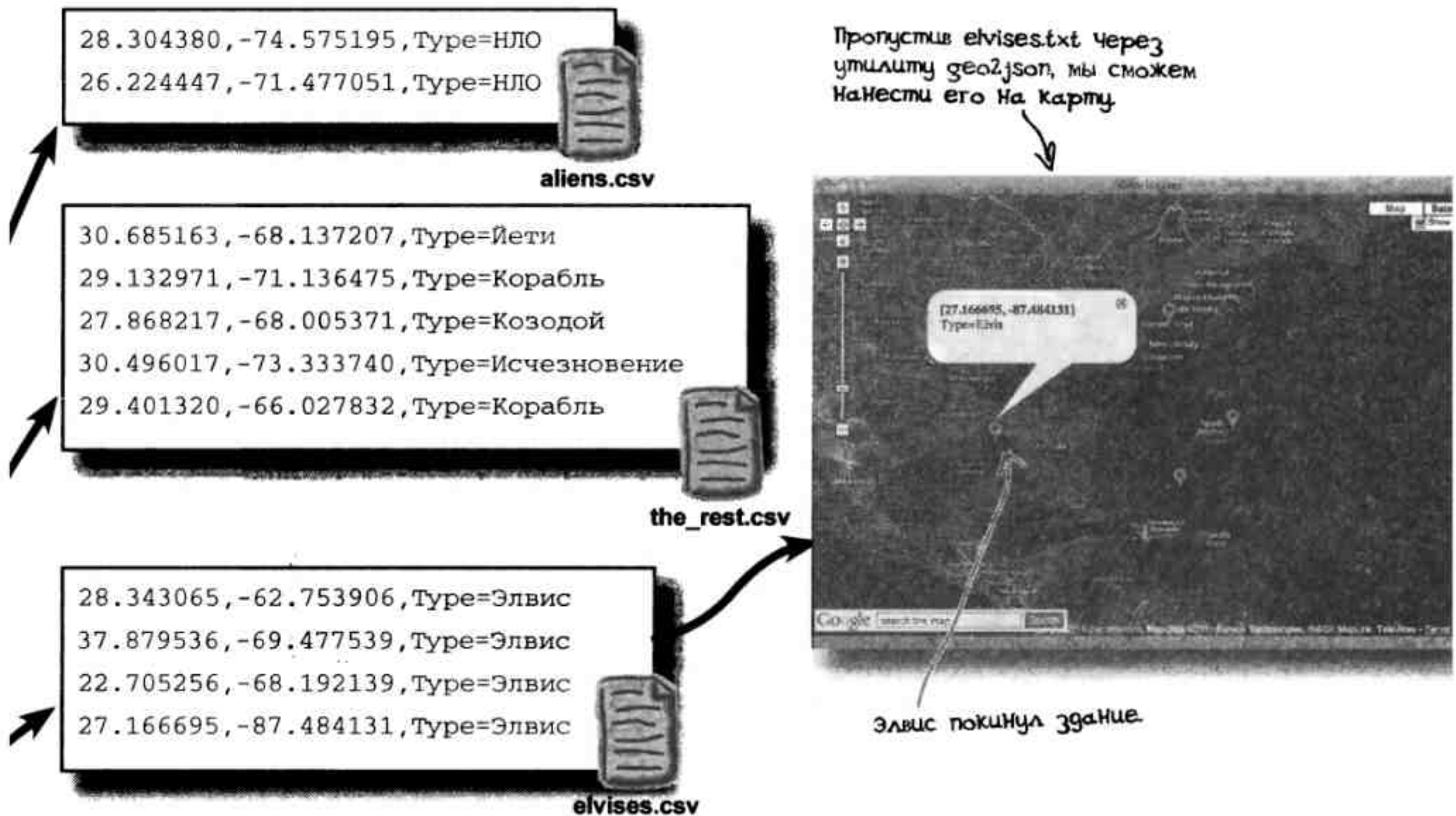
```
30.685163,-68.137207,Type=Йети
28.304380,-74.575195,Type=НЛО
29.132971,-71.136475,Type=Корабль
28.343065,-62.753906,Type=Элвис
27.868217,-68.005371,Type=Козодой
30.496017,-73.333740,Type=Исчезновение
26.224447,-71.477051,Type=НЛО
29.401320,-66.027832,Type=Корабль
37.879536,-69.477539,Type=Элвис
22.705256,-68.192139,Type=Элвис
27.166695,-87.484131,Type=Элвис
```

spooky.csv

Теперь нам нужно запустить программу `categorize`, передав ей несколько аргументов командной строки — текст для поиска и имена файлов для записи:

```
File Edit Window Help ThankYouVeryMuch
> categorize НЛО aliens.csv Элвис elvises.csv the_rest.csv
```

В ходе своей работы программа создаст следующие файлы:



Проверка безопасности

Хотя в нашей лаборатории мы никогда не допускаем ошибок (кхе-кхе), в реальных программах важно проверять наличие проблем перед открытием файла для чтения или записи. К счастью, если при открытии потока данных проблема все-таки возникла, функция `fopen()` вернет значение 0. Следовательно, если вы хотите проверить, произошла ли ошибка, вы должны привести этот код:

```
FILE *in = fopen("i_dont_exist.txt", "r");
```

к следующему виду:

```
FILE *in;
if (!(in = fopen("dont_exist.txt", "r"))) {
  fprintf(stderr, "Не удалось открыть файл.\n");
  return 1;
}
```

Накладные расходы в нашей пиццерии

Скорее всего, любой программе, которую вы пишете, потребуются параметры. Программе для чата нужны настройки, в игре пользователь захочет выбрать форму для пятен крови. Но, если вы пишете консольную утилиту, вам, вероятно, нужно будет снабдить ее **параметрами командной строки**.

Параметры командной строки — это небольшие ключи, которые часто можно увидеть у консольных утилит:

```
ps -ae
```

← Выводим все процессы, включая окружения, в которых они работают.

```
tail -f logfile.out
```

← Отображаем последние строки файла и все данные, которые будут добавлены в его конец.

Позвольте библиотеке сделать работу за вас

Параметры командной строки используются во многих программах, поэтому для более простой работы с ними существует специальная библиотечная функция. Она называется **getopt()** и при каждом вызове возвращает следующий параметр, который найдет в командной строке.

Давайте посмотрим, как это работает. Представьте, что у нас есть программа, которая может принимать набор разных параметров:

Деликатный путеводитель по стандартам

Заголовок *unistd.h* на самом деле не входит в состав стандартной библиотеки Си. Он предоставляет доступ к некоторым POSIX-библиотекам. Стандарт POSIX был попыткой создать набор функций, общий для всех популярных операционных систем.

Используем 4 двигателя. Режим крутости активирован.

```
rocket_to -e 4 -a Бразилиа Токио Лондон
```

Этой программе нужны два параметра: один принимает значение (`-e = engies`), а второй просто *включает* или *выключает* режим (`-a = awesomeness`). Вы можете обрабатывать эти параметры, вызывая функцию `getopt()` в цикле следующим образом:

```
#include <unistd.h>

...

while ((ch = getopt(argc, argv, "ae:")) != EOF)
  switch(ch) {
  ...
  case 'e':
    engine_count = optarg;
  ...
  }

argc -= optind;
argv += optind;
```

Вам необходимо подключить заголовок.

Это значит, что параметры «a» и «e» являются действительными.

Здесь размещен код для обработки каждого параметра.

«:» означает, что для параметра «e» требуется аргумент.

Здесь мы считываем аргумент для параметра «e».

optind хранит количество строк, считанных из консоли, для получения параметров.

В этих завершающих строчках мы пропускаем параметры, которые уже были прочитаны.

Внутри цикла мы имеем инструкцию `switch`, предназначенную для обработки каждого допустимого парметра. Строка **ae:** говорит функции `getopt()` о том, что допустимыми параметрами являются `a` и `e`. Символ `e`, за которым следует `:`, говорит `getopt()`, что за `-e` должен следовать дополнительный аргумент, на который будет указывать переменная **optarg**.

Когда цикл завершится, мы подкорректируем переменные `argv` и `argc`, чтобы пропустить все остальные парметры и перейти к главным аргументам командной строки. В результате наш массив `argv` примет следующий вид:

```
Бразилиа Токио Лондон
```

Это argv[0]. Это argv[1]. Это argv[2].

Будьте осторожны!

После обработки параметров нулевой аргумент больше не является именем программы.

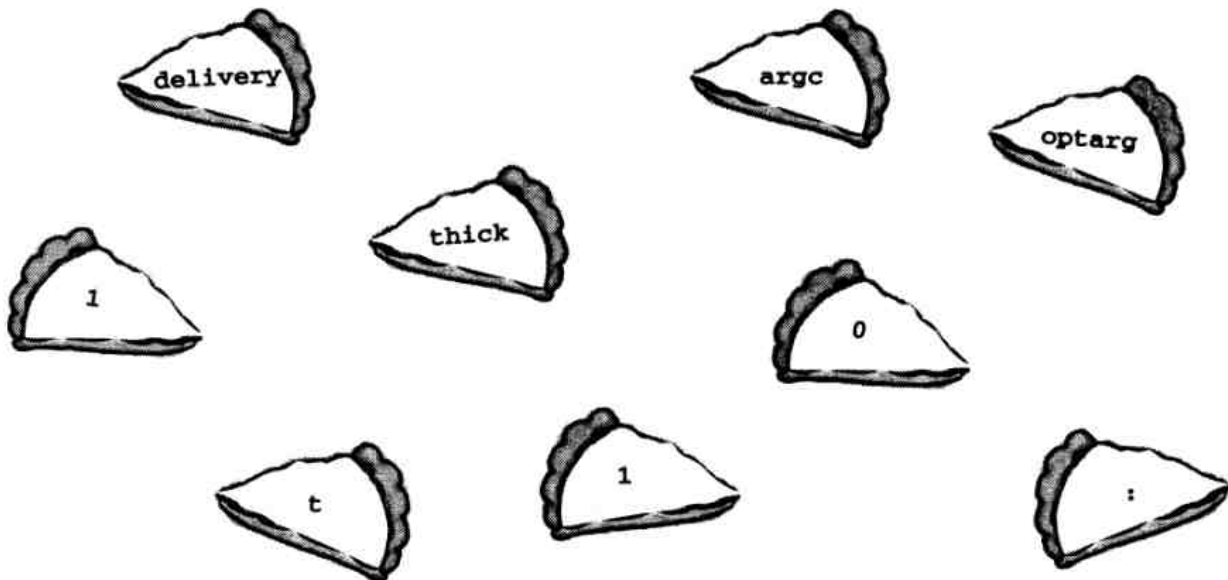
Вместо этого `argv[0]` *будет указывать на первый аргумент командной строки, следующий за параметрами.*

Кусочки пиццы

Похоже, кто-то прикарманил часть нашего кода для пиццы. Посмотрим, сможете ли вы поместить кусочки пиццы на свои места и восстановить программу `order_pizza`.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *delivery = "";
  int thick = 0;
  int count = 0;
  char ch;


  while ((ch = getopt(argc, argv, "d........................ ..............................")) != EOF)
    switch (ch) {
    case 'd':


      ........................   =  ................................ ;
      break;
    case 't':


      ........................   =  ................................ ;
      break;
    default:
      fprintf(stderr, "Неизвестный параметр: '%s'\n", optarg);


      return ................................;
    }
```

```
  argc -= optind;
  argv += optind;

  if (thick)
    puts("Пышное тесто.");

  if (delivery[0])
    printf("Будет доставлено %s.\n", delivery);

  puts("Ингредиенты:");




  for (count = ..............................; count < ..............................; count++)
    puts(argv[count]);
  return 0;
}
```



Кусочки пиццы. Решение

Похоже, кто-то прикарманил часть нашего кода для пиццы. Вам нужно было поместить кусочки пиццы на свои места и восстановить программу `order_pizza`.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *delivery = "";
  int thick = 0;
  int count = 0;
  char ch;
```

Так как параметр «d» принимает аргумент, за ним следует двоеточие.

```
  while ((ch = getopt(argc, argv, "d:t")) != EOF)
    switch (ch) {
    case 'd':
```

```
      delivery = optarg;
      break;
    case 't':
```

```
      thick = 1;
      break;
    default:
      fprintf(stderr, "Неизвестный параметр: '%s'\n", optarg);
```

```
      return 1;
    }
```

```
  argc -= optind;
  argv += optind;

  if (thick)
    puts("Пышное тесто.");

  if (delivery[0])
    printf("Будет доставлено %s.\n", delivery);

  puts("Ингредиенты:");
```

После обработки параметров первый ингредиент будет равняться argv[0].

```
  for (count = ...0...; count < ...argc...; count++)
    puts(argv[count]);
  return 0;
}
```

Цикл будет продолжаться, пока счетчик не сравняется с argc.

Тест-драйв

Теперь мы можем проверить в деле программу для заказа пиццы.

Компилируем программу.

При первых двух вызовах программы мы не используем никаких параметров.

Затем мы пробуем использовать параметр «d», передавая ей аргумент «сейчас».

Затем параметр «t». Помните, этот параметр не предусматривает никаких аргументов.

Наконец мы попытаемся пропустить аргументы для параметра «d», вследствие чего возникает ошибка.

```
File Edit Window Help Anchovies?
> gcc order_pizza.c -o order_pizza
> ./order_pizza Анчоусы
Ингредиенты:
Анчоусы
> ./order_pizza Анчоусы Ананас
Ингредиенты:
Анчоусы
Ананас
> ./order_pizza -d now Анчоусы Ананас
Будет доставлено сейчас.
Ингредиенты:
Анчоусы
Ананас
> ./order_pizza -d now -t Анчоусы Ананас
Пышное тесто.
Будет доставлено сейчас.
Ингредиенты:
Анчоусы
Ананас
> ./order_pizza -d
order_pizza: option requires an argument -- d
Неизвестный параметр: '(null)'
>
```

Работает!

Ну что ж, в этой главе мы охватили много материала. Подробно рассмотрели стандартный ввод, стандартный вывод и стандартный поток ошибок. Узнали, как взаимодействовать с файлами с помощью перенаправления и наших собственных потоков данных.
И наконец, мы научились работать с аргументами и параметрами командной строки.

Многие Си-программисты тратят свое время на создание простых инструментов. Большая часть утилит в таких операционных системах, как Linux, тоже написана на языке Си. Если вы станете тщательнее проектировать свои программы и будете уверены в том, что они **хорошо выполняют** какую-то **одну задачу**, то вскоре сможете стать крутым разработчиком на Си.

не бывает Глупых Вопросов

В: Могу ли я сочетать параметры в виде `-td now` вместо `-d now -t`?

О: Да, можете. Функция `getopt()` сделает все за вас.

В: А что насчет изменения порядка следования парметров?

О: Нет никакой разницы в том, как вы запишете парметры: `-d now -t`, `-t -d now` или `-td now`.

В: Значит, если программа видит, что значение в командной строке начинается с -, она будет рассматривать его, как параметр?

О: Если она прочитает его до того, как доберется до главных аргументов командной строки, то да.

В: Но что если в качестве аргументов командной строки мы хотим передать отрицательные числа, например `set_temperature -c -4`? Не воспримет ли программа 4 как параметр, а не как аргумент?

О: Чтобы избежать неопределенности, вы можете отделить главные аргументы от параметров, используя `--`. То есть вам нужно будет написать `set_temperature -c -- -4`. Функция `getopt()`, дойдя до символов `--`, прекратит считывание параметров, поэтому оставшаяся часть строки будет прочитана как набор простых аргументов.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Существует две версии функции `main()`: одна — с аргументами командной строки, другая — без.
- Аргументы командной строки передаются в функцию `main()` в виде двух значений: количества аргументов и массива указателей на них.
- Параметры командной строки являются аргументами, у которых имеется префикс `-`.
- Функция `getopt()` помогает работать с параметрами командной строки.
- Описать допустимые параметры можно путем передачи функции `getopt()` значений наподобие `ae:`.
- Если после параметра идет двоеточие (`:`), то она может принимать дополнительный аргумент.
- Функция `getopt()` запишет аргументы параметров с помощью переменной `optarg`.
- После того как все параметры будут прочитаны, вам необходимо пропустить их, используя переменную `optind`.

Ваш инструментарий языка Си

Вы изучили главу 3, пополнив свои навыки умением создавать простые инструменты. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

В Си такие функции, как printf() и scanf(), используют для взаимодействия стандартных ввода и вывода.

По умолчанию стандартный вывод направлен на экран.

Стандартный поток ошибок — это отдельный выходной поток данных, предназначенный для сообщений об ошибках.

С помощью перенаправления вы можете изменить подключение стандартных ввода, вывода и потока ошибок.

По умолчанию стандартный ввод считывает данные с клавиатуры.

Вы можете производить печать в стандартный поток для ошибок с помощью fprintf(stderr,...).

Аргументы командной строки передаются в функцию main() в виде массива строковых указателей.

Вы можете создавать собственные потоки данных с помощью fopen("имя_файла", mode)

Функция getopt() упрощает чтение параметров командной строки.

Существует три режима:
«w» — для записи,
«r» — для чтения,
«a» — для добавления.

4 Использование нескольких исходных файлов

Разбиваем на части, собираем вместе

Для большой программы лучше не использовать один объемный исходный файл.

Представьте себе, какой сложной и трудоемкой может оказаться поддержка одного-единственного файла программы в масштабах предприятия. В этой главе вы узнаете, как с помощью Си разбить исходный код на **небольшие легко управляемые части**, а потом собрать их в **одну большую программу**. Заодно сможете более подробно изучить некоторые **тонкости, касающиеся типов данных**, и познакомиться с утилитой `make`.

Угадайте тип данных

В Си есть всего несколько типов данных: символы и целые числа, числа с плавающей точкой для обычных вычислений, а также числа с плавающей точкой для по-настоящему точных научных расчетов. С некоторыми из них вы можете ознакомиться на следующей странице. Посмотрим, сможете ли вы определить, какие типы данных используются в каждом из примеров.

Помните: во всех примерах используются разные типы данных.

Расстояние от стартовой площадки до Проксимы Центавра (в световых годах)

Количество звезд во Вселенной, которые мы не будем посещать

Количество минут до запуска

Каждый символ на табло обратного отсчета

Это числа с десятичным разделителем.

Числа с плавающей точкой

- float
- double

Целые числа

- short
- long
- int
- char

Именно так! В Си символы хранятся в виде кодов. Это значит, что они тоже являются числами!

Общее количество компонентов в ракете

int

Количество топлива, которое потребуется для ракеты (в галлонах)

float

Угадайте тип данных. Решение

В Си есть всего несколько типов данных: символы и целые числа, числа с плавающей точкой для обычных вычислений, а также числа с плавающей точкой для по-настоящему точных научных расчетов. С некоторыми из них вы можете ознакомиться на следующей странице. Вам нужно было определить, какие типы данных используются в каждом из примеров.

Помните: во всех примерах используются разные типы данных.

Расстояние от стартовой площадки до Проксима Центавра (в световых годах)

`double`

Количество звезд во вселенной, которые мы не будем посещать

`long`

Количество минут до запуска

`short`

Каждый символ на табло обратного отсчета

`char`

Давайте разберемся, почему так...

Краткое руководство по типам данных

char

Каждый символ хранится в памяти в виде соответствующего кода. И это всего лишь число. Для компьютера увидеть Z — это то же самое, что увидеть алгебраическое число 90.

90 – это ASCII-код для Z

int

Если вам необходимо сохранить целое число, в общем случае вы можете использовать тип int. На большинстве компьютеров этот тип поддерживает числа вплоть до нескольких миллионов.

short

Но иногда вам может понадобиться сэкономить немного памяти. Зачем использовать int, если вы хотите хранить числа не больше нескольких сотен или тысяч? Для этого и предназначен тип short. Числа этого типа, как правило, занимают в два раза меньше места, чем int.

long

А что если вы хотите сохранить **действительно большое число**? Для этого был придуман тип long. На некоторых компьютерах он занимает в *два* раза больше памяти, чем int, и может хранить числа вплоть до **миллиардов**. Однако поскольку большинство компьютеров могут работать с действительно большими числами типа int, то на многих машинах для int и long выделяется *одинаковое* количество памяти.

float

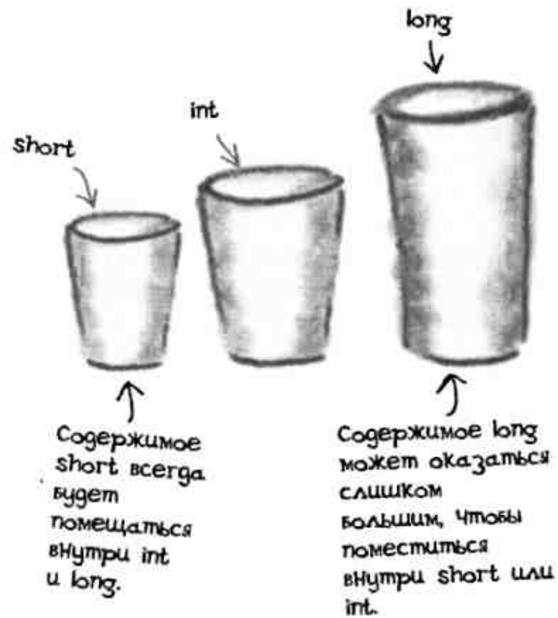
float является основным типом для хранения чисел с плавающей точкой. Для большинства обычных дробных значений, таких, например, как количество молока в вашей чашке с моккачино, вы можете использовать float.

double

А что если вам нужна действительно высокая **точность**? Если вы хотите произвести вычисления, учитывая как можно больше **знаков после запятой**, тогда вам может пригодиться тип double. Он занимает в два раза больше памяти, чем float, используя дополнительное место для хранения *более крупных и точных значений*.

Не помещайте что-то большое во что-то маленькое

Относитесь осторожно к обмену значениями, следите за тем, чтобы тип значения соответствовал типу переменной, которой вы собираетесь его присвоить.



Разные типы данных используют разные объемы памяти. Убедитесь, что вы не пытаетесь присвоить переменной слишком большое значение. Переменные типа `short` занимают меньше памяти, чем переменные типа `int`, последние, в свою очередь, уступают по размерам типу `long`.

Нет никакой проблемы в том, чтобы сохранить значение типа `short` внутри переменной типа `int` или `long`. В памяти останется достаточно места, и ваш код будет работать как положено:

```
short x = 15;
int y = x;
printf("Значение y = %i\n", y);
```

Этим мы скажем, что y = 15.

Проблемы появятся, когда вам необходимо будет сделать обратное: например, присвоить значение `int` переменной типа `short`.

```
int x = 100000;
short y = x;
print("Значение y = %hi\n", y);
```

%hi — это код, предназначенный для форматирования значения типа short.

Иногда компилятор может определить, что вы пытаетесь сохранить слишком большое значение внутри маленькой переменной, и выдаст вам предупреждение. Но в большинстве случаев он оказывается недостаточно проницательным, в результате чего код будет компилироваться без замечаний. И вот тогда, запустив программу, вы обнаружите, что компьютер не может поместить число 100 000 в переменную типа `short`. Он упакует в нее столько ноликов и единичек, сколько сумеет, но в итоге внутри переменной окажется *совсем не то* число, которое вы ей присваивали:

```
Значение y = -31072
```

Уголок ботана

Так почему же большое число внутри переменной типа `short` стало отрицательным? Числа хранятся в двоичном виде. Вот как на самом деле выглядит 100 000:

```
x <- 0001 1000 0110 1010 0000
```

Но когда компьютер попытался сохранить это значение внутри типа `short`, ему было предоставлено всего несколько байт в памяти. Программа сохранила только *правую часть* числа:

```
y <- 1000 0110 1010 0000
```

Значения *со знаком*, которые в двоичном виде начинаются с 1 в первом бите, воспринимаются как отрицательные числа. В десятичной системе это укороченное значение будет выглядеть так:

```
-31072
```

Приведение типа float к целому числу

Как по-вашему, что выведет на экран следующий фрагмент кода?

```
int x = 7;
int y = 2;
float z = x / y;
printf("z = %f\n", z);
```

Ответом будет **3.0000**. Почему? Видите ли, x и y имеют тип int, а при делении целых чисел вы всегда получаете число, округленное до целого, в данном случае это **3**.

А что делать, если при расчетах с целыми числами вам нужно получить дробные результаты? Вы можете сперва поместить все числа внутрь переменных типа float, но для этого нужно будет написать лишний код. Вместо этого вы можете использовать **приведение**, чтобы преобразовывать числа на лету:

```
int x = 7;
int y = 2;
float z = (float)x / (float)y;
printf("z = %f\n", z);
```

(float) *приведет* целочисленное значение к типу float. В результате вычисления пройдут так, словно вы изначально использовали числа с плавающей точкой. Фактически, если компилятор видит, что вы складываете, вычитаете, умножаете или делите дробное значение и целое число, он выполнит приведение типов автоматически. Это значит, что в ряде случаев вы можете избежать явного преобразования чисел:

```
float z = (float)x / y;
```

Компилятор автоматически приведет y к типу float.

Чтобы повлиять на то, как компьютер будет интерпретировать числа, вы можете использовать некоторые ключевые слова перед типами данных:

unsigned

Беззнаковое число всегда будет положительным. И поскольку нет необходимости записывать отрицательные значения, то переменные с этим ключевым словом могут хранить бо́льшие числа. Таким образом, unsigned int может записывать числа от 0 до максимального значения, примерно вдвое большего, чем максимальное число, которое может храниться внутри типа int. Существует также ключевое слово signed, но оно почти никогда не используется, поскольку все типы данных имеют знак по умолчанию.

```
unsigned char c;
```

Здесь, вероятно, будут храниться числа от 0 до 255.

long

Все правильно, чтобы сделать тип данных длиннее, вы можете ставить перед ним ключевое слово long. long int — это удлиненная версия int, способная хранить более широкий диапазон чисел. long long длиннее, чем просто long. Вы можете также использовать long для чисел с плавающей точкой. Однако диапазон значений у long double не больше, чем у double, при этом просто увеличивается точность числа.

Тип long long применим только в стандартах C99 и C11.

```
long double d;
```

Очень ОЧЕНЬ точное число.

Упражнение

В закусочной Head First появилась новая программа, которая помогает официантам обслуживать столики. Код автоматически формирует итоговый счет и добавляет к каждому наименованию сумму налога. Посмотрим, сможете ли вы заполнить пропущенные места.

Примечание: в этой программе может использоваться несколько типов данных, но какие из них дадут ожидаемый результат?

```
#include <stdio.h>

.................. total = 0.0;
.................. count = 0;
.................. tax_percent = 6;

.................. add_with_tax(float f);
{
  ..................tax_rate = 1 + tax_percent / 100 .................. ;
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}

int main()
{
  ..................val;
  printf("Цена блюда: ");
  while (scanf("%f", &val) == 1) {
    printf("Итого на текущий момент: %.2f\n", add_with_tax(val));
    printf("Цена блюда: ");
  }
  printf(("\nИтоговый счет: %.2f\n", total);
  printf("Количество блюд: %hi\n", count);
  return 0;
}
```

%2f выводит дробь с двумя знаками после запятой

%hi используется для форматирования чисел типа short.

Упражнение Решение

В закусочной Head First появилась новая программа, которая помогает официантам обслуживать столики. Код автоматически формирует итоговый счет и добавляет к каждому наименованию сумму налога. Вам нужно было заполнить пропуски.

Примечание: в этой программе могло использоваться несколько типов данных, но какие из них дали бы ожидаемый результат?

```
#include <stdio.h>

float total = 0.0;
short count = 0;
short tax_percent = 6;

float add_with_tax(float f);
{
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}

int main()
{
  float val;
  printf("Цена блюда: ");
  while (scanf("%f", &val) == 1) {
    printf("Итого на текущий момент: %.2f\n", add_with_tax(val));
    printf("Цена блюда: ");
  }
  printf("\nИтоговый счет: %.2f\n", total);
  printf("Количество блюд: %hi\n", count);
  return 0;
}
```

Чтобы подсчитать наличные, нам нужно небольшое число с плавающей точкой (→ float total).

В заказе не будет много блюд, поэтому мы можем выбрать тип short (→ count).

Мы возвращаем небольшое значение, описывающее наличные, поэтому оно будет иметь тип float (→ add_with_tax).

Для этой дроби подойдет тип float (→ tax_rate).

Добавляя .0, мы делаем так, чтобы вычисления проходили с дробными числами. Если мы запишем это как 100, то получим целое число.

1 + tax_percent / 100 вернет 1, так как в целочисленной арифметике 6 / 100==0.

Любая цена легко поместится в тип float (→ val).

Подробнее о размерах типов данных

Типы данных имеют разные размеры на разных платформах. Но как узнать, насколько велик тип int или сколько байтов занимает double? К счастью, в стандартной библиотеке Си есть несколько заголовков с некоторыми подробностями. Эта программа покажет вам размеры типов int и float:

```
#include <stdio.h>
#include <limits.h>  <- Здесь содержатся значения для целочисленных типов, таких как int и char.
#include <float.h>   <- Здесь содержатся значения для типов float и double.

int main()
{
  printf("Значение INT_MAX равно %i\n", INT_MAX);
  printf("Значение INT_MIN равно %i\n", INT_MIN);
  printf("int занимает %li байт\n", sizeof(int));

  printf("Значение FLT_MAX равно %f\n", FLT_MAX);
  printf("Значение FLT_MIN равно %.50f\n", FLT_MIN);
  printf("float занимает %li байт\n", sizeof(float));

  return 0;
}
```

Это максимальное значение.

Это минимальное значение.

sizeof возвращает количество байтов, которые занимает тип данных.

Скомпилировав и запустив этот код, вы увидите что-то вроде:

```
File Edit Window Help HowBigIsBig
Значение INT_MAX равно 2147483647
Значение INT_MIN равно -2147483648
Int занимает 4 байт
Значение FLT_MAX равно 340282346638528859811704183484516925440.000000
Значение FLT_MIN равно 0.000000000000000000000000000000000000011754943508 22
float занимает 4 байт
```

Возможно, на своем личном компьютере вы увидите другие значения. А что если вы хотите узнать подробности о типах char или double? Или long? Нет проблем. Просто замените INT и FLT на CHAR (для char), DBL (для double), SHRT (для short) или LNG (для long).

не бывает
Глупых Вопросов

В: Почему типы данных отличаются на разных операционных системах? Разве не было бы меньше путаницы, если бы они везде были одинаковыми?

О: Си использует разные типы данных на разных операционных системах и процессорах, потому что это позволяет максимально задействовать аппаратные ресурсы.

В: Каким образом?

О: Во времена, когда создавался язык Си, большинство компьютеров были 8-битными. Сейчас превалируют 32- и 64-битные системы. Поскольку в Си не определен конкретный размер типов данных, со временем этот язык адаптировался к новым условиям. С появлением новых компьютеров Си продолжит использовать их мощь на полную силу.

В: Что означают эти 8 и 64 бита?

О: Строго говоря, эти цифры могут указывать на несколько вещей: например, на размер инструкций в центральном процессоре или на количество данных, которые процессор способен прочитать из памяти. Под битовым размером главным образом понимают объем чисел, с которыми компьютер может справиться.

В: И как это связано с размерами типов `int` и `double`?

О: Если компьютер лучше всего оптимизирован для работы с 32-битными числами, было бы логично, чтобы базовый тип данных — `int` — занимал 32 бита.

В: Я понимаю, как работают целые числа, такие как `int`, но каким образом хранятся дроби? Как представляет компьютер числа с десятичным разделителем?

О: Это сложно. Большинство компьютеров использует стандарт, опубликованный на сайте IEEE (*http://tinyurl.com/6defkv6*).

В: Мне действительно нужно понимать принцип работы чисел с плавающей точкой?

О: Нет. Абсолютное большинство разработчиков используют типы `float` и `double`, не вдаваясь в подробности.

О нет... Это же безработные актеры...

Не все люди рождены стать программистами. Похоже, что несколько честолюбивых актеров в перерыве *между съемками* решили немного подзаработать, «почистив» код нашей программы для формирования итогового счета.

Они были очень довольны тем, как выглядит переписанный код... но у них возникла одна крошечная проблема.

Код больше не компилируется.

Давайте посмотрим, что случилось с кодом

Вот что актеры сделали с кодом. Как вы видите, они внесли всего несколько изменений:

```
#include <stdio.h>

float total = 0.0;
short count = 0;
/* Это 6%. Что намного меньше суммы, которую я плачу своему агенту… */
short tax_percent = 6;

int main()
{
  /* Эй, я ходил на фильм с Вэлом Килмером */
  float val;
  printf("Цена блюда: ");
  while (scanf("%f", &val) == 1) {
    printf("Итого на текущий момент: %.2f\n", add_with_tax(val));
    printf("Цена блюда: ");
  }
  printf("\nИтоговый счет: %.2f\n", total);
  printf("Количество блюд: %hi\n", count);
  return 0;
}

float add_with_tax(float f)
{
  float tax_rate = 1 + tax_percent / 100.0;
  /* А что насчет чаевых? Уроки вокала не бесплатные */
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}
```

В код добавили несколько комментариев и еще **поменяли порядок следования функций**. Больше никаких изменений сделано не было.

Таким образом, вроде бы не должно быть никаких проблем. Похоже, код готов к работе, не так ли? И все бы замечательно, но тут **код скомпилировали**…

Тест-драйв

Итак, если вы откроете консоль и попробуете скомпилировать программу, произойдет следующее:

```
File Edit Window Help StickToActing
> gcc totaller.c -o totaller && ./totaller
totaller.c: In function "main":
totaller.c:14: warning: format "%.2f" expects type
"double", but argument 2 has type "int"
totaller.c: At top level:
totaller.c:23: error: conflicting types for "add_with_tax"
totaller.c:14: error: previous implicit declaration of
"add_with_tax" was here
```

Разочарование.

Нехорошо получилось. Что значит этот `error: conflicting types for 'add_with_tax'`? Что такое *previous implicit declaration*? И почему компилятор думает, что в строчке, которая выводит текущий итог, находится значение типа `int`? Разве мы не предусмотрели там число с плавающей точкой?

Компилятор проигнорирует изменения, внесенные в комментарии, следовательно, они не играют никакой роли. Значит, проблема вызвана **изменениями в порядке следования функций**. Но если это так, то почему компилятор не сообщил что-то вроде:

Действительно, почему компилятор нам здесь совсем не помогает?

Чтобы понять, что же в действительности произошло, нам необходимо на время представить себя компилятором и посмотреть на все с его точки зрения. Тогда станет ясно, что компилятор на самом деле *старается помочь.*

Компиляторы не любят сюрпризы

Так что же происходит, когда компилятор видит эту строчку кода?

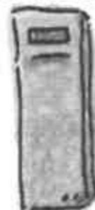
```
printf("Итого на текущий момент: %.2f\n", add_with_tax(val));
```

1. **Компилятор находит вызов функции, которую он не распознает.**
Прежде чем пожаловаться, компилятор предполагает, что узнает больше об этой функции по мере чтения исходного файла. Он просто запоминает, что в дальнейшем ему нужно поискать эту функцию. К сожалению, здесь и возникает проблема...

Эй, здесь вызывается функция, о которой я никогда не слышал. Но я пока возьму это себе на заметку и позже все выясню.

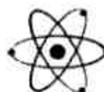
2. **Компилятору нужно знать, какой тип данных вернет функция.**
Пока компилятору неизвестен тип данных, возвращаемых функцией, он делает ***предположение***. Компилятор предполагает, что это будет `int`.

Готов поспорить, эта функция возвращает int. Большинство именно так и делают.



3. **Дойдя в коде до настоящей функции, он возвращает ошибку о конфликте типов «conflicting types for 'add_with_tax'».**
Все из-за того, что компилятор думает, будто у него есть две функции с одним и тем же именем. Одна из них настоящая, та, что находится в файле. Другая — та, что, по предположению компилятора, должна была вернуть `int`.

Функция под именем add_with_tax(), возвращающая **float**? Но у меня тут записано, что у нас уже есть одна такая, и она возвращает int...



Сила мозга

Компьютер делает предположение, что функция возвращает `int`, хотя на самом деле она возвращает `float`. Как бы вы решили эту проблему на месте создателей языка Си?

Эй! Мне правда **все равно**, как язык Си решает эту проблему. Просто расположите функции **в правильном порядке**!

Мы могли бы просто вернуть прежнюю очередность функций, чтобы они вызывались после того, как будут определены.

Меняя очередность функций, мы можем избежать ситуации, когда компилятор делает всякие опасные предположения о типе данных, который они возвращают. Но всегда описывая функции в определенном порядке, мы порождаем несколько новых проблем.

Исправление очередности функций — это головная боль

Представим, что вы добавили в свой код новую классную функцию, от которой все в восторге:

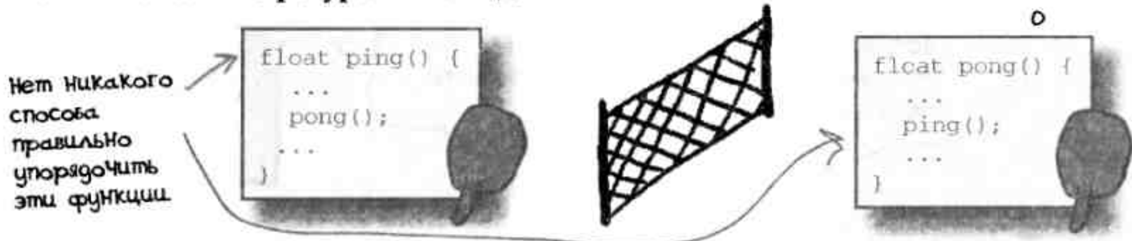
```
int do_whatever(){...}
float do_something_fantastic(int awesome_level) {...}
int do_stuff() {
  do_something_fantastic(11);
}
```

Что произойдет, если *потом* вы решите, что вызов функции `do_something_fantastic()` внутри существующей `do_whatever()` сделает вашу программу только *лучше*? Вам придется **переместить функцию** выше. Большинство программистов предпочитают тратить свое время на улучшения возможностей написанного ими кода. Было бы неплохо, если бы вам не нужно было менять очередность кода только для того, чтобы удовлетворить компилятор.

В некоторых ситуациях правильной очередности не существует

Вам слово, Сесил!

Это довольно редкий случай, но не исключено, что когда-нибудь вы напишете **взаимно рекурсивный код**:



Если у вас есть две функции, которые вызывают *друг друга*, то **одна из них всегда будет вызываться в файле до того, как будет определена**.

Учитывая оба обстоятельства, было бы здорово определять функции в наиболее удобном порядке на момент написания кода. Но как?

Отделяем объявление от определения

Помните, как компилятор сделал себе пометку о функции, которую он ожидает найти дальше в файле? ***Мы можем явно сообщить ему, какую функцию он должен ожидать,*** избавив от необходимости делать предположение. Когда мы рассказываем компилятору о функции, это называется **объявлением функции**:

Объявление говорит компилятору, какое возвращаемое значение нужно ожидать. → **float add_with_tax();** ← Оно заканчивается точкой с запятой

У объявления нет тела с кодом.

Объявление — это всего лишь **сигнатура** — запись о том, какая функция будет вызвана, какие параметры она примет и **какой тип данных вернет**.

После объявления функции компилятору больше не нужно делать предположений; ему будет все равно, даже если вы определите функцию после ее вызова.

Итак, если у вас в коде есть множество функций и вы не хотите волноваться о порядке их размещения в файле, то можете объявить их списком в начале кода:

```
float do_something_fantastic();
double awesomeness_2_dot_0();
int stinky_pete();
char make_maguerita(int count);
```

Но что еще лучше, Си позволяет вам вынести все эти объявления за *пределы кода* и поместить их в **заголовочный файл**. Мы уже использовали заголовочные файлы, чтобы подключать код из стандартной библиотеки Си:

```
#include <stdio.h>
```

В этой строчке подключается содержимое заголовочного файла под названием stdio.h

Давайте посмотрим, как создавать собственные заголовочные файлы.

Создаем ваш первый заголовочный файл

Для создания заголовка необходимо сделать **две вещи**.

1. **Создать новый файл с расширением .h.**
Если вы пишете программу под названием `totaller.c`, создайте файл с именем *totaller.h* и запишите внутри него свои объявления:

```
float add_with_tax(float f);
```

totaller.h

Функцию `main()` в заголовочный файл добавлять не нужно, поскольку вы больше нигде не будете ее вызывать.

2. **Подключить заголовочный файл к главной программе.**
Вы должны добавить еще одну директиву `#include` в верхней части свой программы:

Добавьте эту директиву include к уже имеющимся.

```
#include <stdio.h>
#include "totaller.h"
...
```

totaller.c

При написании имени заголовочного файла убедитесь, что вы используете двойные кавычки вместо угловых скобок. В чем разница? Когда компилятор видит директиву `#include` с угловыми скобками, он предполагает, что сможет найти заголовочный файл в одной из директорий, где размещен код библиотеки. Однако **наш** заголовок находится в одном каталоге с файлом *.c*. Используя двойные кавычки, мы говорим компилятору, что нужно искать локальный файл.

В названиях локальных заголовков могут содержаться имена директорий, но, как правило, вы будете размещать их в одном каталоге с .c-файлами.

Когда компилятор видит в коде директиву `#include`, он считывает содержимое заголовочного файла, словно оно было добавлено в код.

Запись объявлений в отдельный заголовочный файл делает ваш основной код немного короче и дает еще одно *существенное преимущество*, о котором мы узнаем через несколько страниц.

А пока давайте проверим, помог ли заголовочный файл исправить неразбериху.

#include — это директива препроцессора (или прекомпиляции).

Тест-драйв

Теперь, когда мы скомпилируем код, произойдет следующее:

```
File Edit Window Help UseHeaders
> gcc totaller.c -o totaller
```

На этот раз никаких сообщений об ошибках.

Компилятор считывает объявления функций из заголовочного файла, а это значит, что ему не нужно делать никаких предположений о возвращаемом типе функций. Не важно, в каком порядке они размещены.

Чтобы убедиться в отсутствии проблем, мы можем запустить сгенерированную программу и увидеть, что она работает так же, как и прежде.

```
File Edit Window Help UseHeaders
> ./totalling_fixed
Цена блюда: 1.23
Итого на текущий момент: 1.30
Цена блюда: 4.57
Итого на текущий момент: 6.15
Цена блюда: 11.92
Итого на текущий момент: 18.78
Цена блюда: ^D
Итоговый счет: 18.78
Количество блюд: 3
```

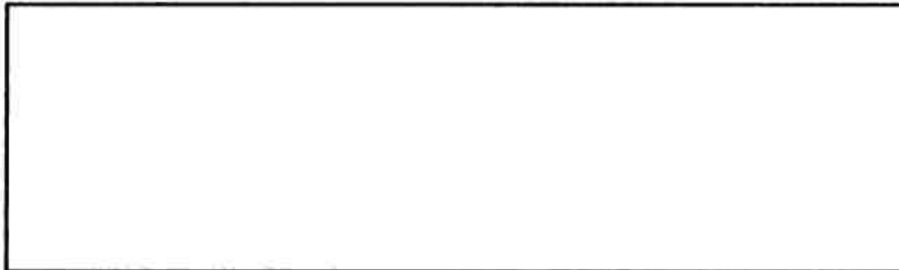
Здесь мы нажали Ctrl+D, чтобы программа прекратила запрашивать у нас цены

Поработайте Компилятором

Взгляните на программу, представленную ниже. Часть ее утеряна. Ваша задача — определить, как отреагирует компилятор на каждый из фрагментов, размещенных справа, если мы будем подставлять их вместо пропущенного кода.

Сюда вставляются возможные фрагменты кода.

```
#include <stdio.h>
```



```
  printf("День на Меркурии равен %f часам\n", day);
  return 0;
}

float mercury_day_in_earth_days()
{
  return 58.65;
}

int hours_in_an_earth_day()
{
  return 24;
}
```

Это фрагменты кода.

Отметьте варианты, которые, по вашему мнению, являются правильными.

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [ ] Вы можете скомпилировать код.
- [ ] Вы должны показать предупреждение.
- [ ] Программа будет работать.

```
float mercury_day_in_earth_days();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [ ] Вы можете скомпилировать код.
- [ ] Вы должны показать предупреждение.
- [ ] Программа будет работать.

```
int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [ ] Вы можете скомпилировать код.
- [ ] Вы должны показать предупреждение.
- [ ] Программа будет работать.

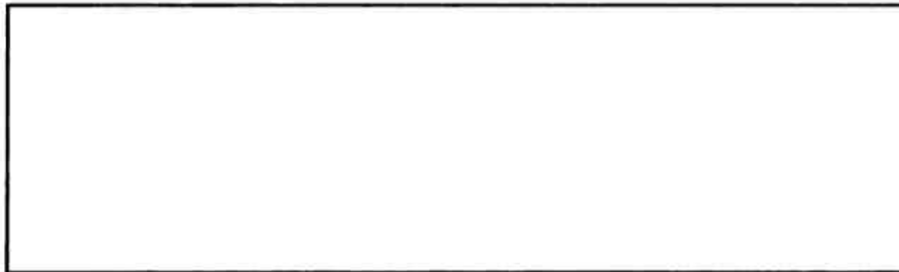
```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  int length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [ ] Вы можете скомпилировать код.
- [ ] Вы должны показать предупреждение.
- [ ] Программа будет работать.

Поработайте Компилятором. Решение

Взгляните на программу, представленную ниже. Часть ее утеряна. Вам необходимо было определить, как отреагирует компилятор на каждый из фрагментов, размещенных справа, если мы подставим их вместо пропущенного кода.

```
#include <stdio.h>
```



```
  printf("День на Меркурии равен %f часам\n", day);
  return 0;
}

float mercury_day_in_earth_days()
{
  return 58.65;
}

int hours_in_an_earth_day()
{
  return 24;
}
```

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

Вы можете скомпилировать код.

Вы должны показать предупреждение.

Программа будет работать.

Здесь выведется предупреждение, поскольку мы не объявили функцию hours_in_an_earth_day(), прежде чем ее использовать. Но программа все равно будет работать, подразумевая, что эта функция вернет int.

```
float mercury_day_in_earth_days();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

Вы можете скомпилировать код.

Вы должны показать предупреждение.

Программа будет работать.

```
int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

Программа не скомпилируется, потому что мы вызываем функцию типа float, не объявив ее заранее.

Вы можете скомпилировать код.

Вы должны показать предупреждение.

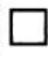
Программа будет работать.

Программа скомпилируется без предупреждений, но работать не будет, так как возникнет проблема с округлением.

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  int length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

Переменная length_of_day должна иметь тип float.

Вы можете скомпилировать код.

 Вы должны показать предупреждение.

Программа будет работать.

не бывает Глупых Вопросов

В: Значит, мне не нужно делать объявления для функций типа `int`?

О: Не обязательно, разве что в общем коде. Скоро мы поговорим об этом подробней.

В: Я запутался. Вы говорили о *прекомпиляции*, выполняемой компилятором? Зачем *компилятору* этим заниматься?

О: Строго говоря, компилятор отвечает только за этап компиляции: он преобразует код на языке Си в код ассемблера. Но в более широком смысле *компиляцией* называются все стадии на пути от исходного кода к исполняемому файлу. `gcc` позволяет вам контролировать этот процесс, производя как прекомпиляцию, так и компиляцию.

В: Что такое прекомпилятор?

О: Прекомпилятор (или препроцессор) занимается первой стадией превращения исходного кода на Си в готовый исполняемый файл. До начала *настоящей* компиляции препроцессор создает модифицированную версию исходников. В нашем случае на этапе прекомпиляции в главный файл помещается содержимое заголовочного файла.

В: Создает ли препроцессор какой-нибудь файл?

О: Нет. Компиляторы, как правило, используют каналы для передачи результатов своей работы между этапами компиляции. Это делает процесс более эффективным.

В: Почему одни заголовки заключены в кавычки, а другие — в угловые скобки?

О: Кавычки означают, что для поиска файла нужно использовать относительный путь. Если указать только имя файла без директории, компилятор будет искать его в текущем каталоге. Если используются угловые скобки, он сделает поиск по цепочке директорий.

В: В каких директориях компилятор будет искать заголовочные файлы?

О: Компилятор `gcc` знает, где расположены стандартные заголовки. В Unix-подобных операционных системах заголовочные файлы, как правило, находятся в таких директориях, как */usr/local/include*, */usr/include* и нескольких других.

В: Значит, так он работает для стандартных заголовков, таких как *stdio.h*?

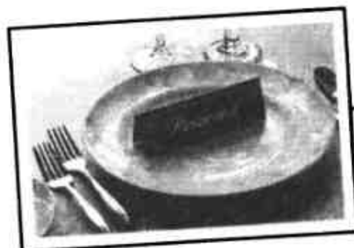
О: Да. В Unix-подобных ОС Вы можете найти файл *stdio.h* по адресу */usr/include/stdio.h*. Если у вас есть компилятор MinGW под Windows, заголовок, вероятно, будет находиться в *C:\MinGW\include\stdio.h*.

В: Могу ли я создавать свои собственные библиотеки?

О: Да. Позже мы покажем вам, как это делается.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Если компилятор находит вызов функции, о которой он ничего не знает, он делает предположение, что она возвращает `int`.
- Таким образом, если вы попытаетесь вызвать функцию до того, как она будет объявлена, могут возникнуть проблемы.
- Объявление функции говорит компилятору о том, как она будет выглядеть, прежде чем вы ее определите.
- Если определение функции будет находиться в верхней части исходного кода, у компилятора не возникнет проблем с типом, который она возвращает.
- Объявления функций часто помещают в заголовочные файлы.
- Вы можете заставить компилятор прочитать заголовочный файл, используя директиву `#include`.
- Компилятор будет обращаться с подключенным кодом так, будто он был набран прямо в исходном файле.



Этот столик зарезервирован...

Си — очень лаконичный язык. Здесь представлен весь набор зарезервированных слов (в произвольном порядке).

Любая программа на Си, которую вы когда-либо увидите, будет состоять из этих слов и еще нескольких символов. Если вы попробуете использовать их в качестве имен функций или переменных, компилятор очень сильно огорчится.

auto	if	break
int	case	long
char	register	continue
return	default	short
do	sizeof	double
static	else	struct
entry	switch	extern
typedef	float	union
for	unsigned	goto
while	enum	void
const	signed	volatile

Если у вас есть набор часто используемых возможностей...

file_hider

Считывает содержимое файла и создает его зашифрованную версию с помощью метода XOR.

message_hider

Считывает последовательности строк из стандартного ввода и выводит их зашифрованные версии в стандартный вывод, используя XOR-шифрование.

Поучаствовав в написании нескольких программ на Си, скорее всего, вы обнаружите, что некоторые их функции и возможности вам бы хотелось использовать и в других приложениях. К примеру, взгляните на размещенные справа технические требования к двум программам.

Шифрование методом XOR — это очень простой способ закодировать фрагмент текста, слагая по модулю каждый его символ с каким-то значением. Он не очень безопасный, зато довольно легкий в использовании. Причем код для шифрования текста может применяться для его расшифровки. Вот функция для кодирования какой-нибудь строки:

```
void encrypt(char *message)
{
  char c;
  while (*message) {
    *message = *message ^ 31;
    message++;
  }
}
```

void означает, что мы ничего не возвращаем.

Передаем в функцию указатель на массив.

Мы переберем в цикле массив и заменим каждый его символ на зашифрованную версию.

Это значит, что мы приплюсуем к каждому символу число 31.

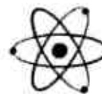
Арифметические действия с символами? Мы можем это делать, потому что char — численный тип данных.

...было бы неплохо поделиться кодом

Очевидно, что обе эти программы должны использовать одну и ту же функцию encrypt(). Значит, вы можете просто скопировать код из одной программы в другую, верно? В этом нет ничего страшного, если речь идет о небольшом объеме кода, но что если его действительно много? И что делать, если позже понадобится изменить принцип работы функции encrypt()? Если у нас будет две копии этой функции, то изменения придется вносить сразу в нескольких местах.

Чтобы ваш код масштабировался как следует, необходимо найти способ повторного использования общих фрагментов кода. Взять каким-то образом набор функций и сделать их доступными для множества других программ.

Как вы этого добьетесь?



Сила мозга

Представьте, что у вас есть набор функций, который вы хотели бы сделать общим для разных программ. Как бы вы поступили, будь вы создателем языка Си?

Вы можете разделить код на отдельные файлы

Если у вас есть фрагмент кода, которым вы хотели бы поделиться с несколькими файлами, было бы логично вынести его в отдельный исходный файл с расширением *.c*. Если компилятор сумеет каким-то образом подключить этот код во время компиляции, вы сможете использовать его сразу в нескольких приложениях. И если вдруг вам понадобится внести изменения в общую часть, достаточно будет сделать это только в одном месте.

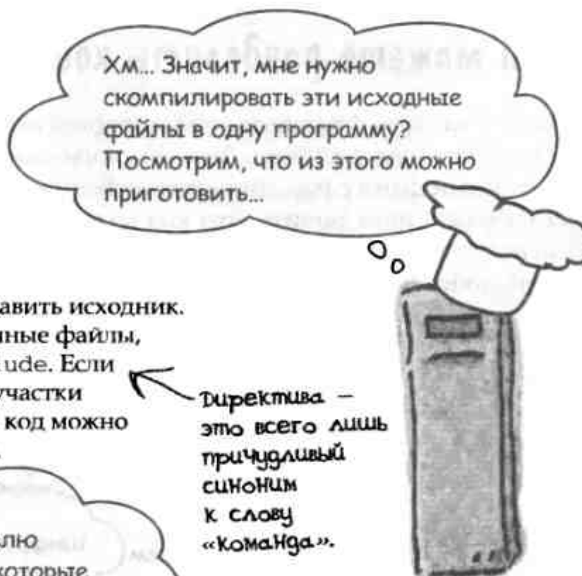
Если мы хотим использовать отдельный *.c*-файл для общего кода, то у нас возникает *проблема*. До сих пор мы создавали программы из одиночных *.c*-файлов. То есть если у нас была программа `blitz_hack`, она создавалась из единственного исходного файла *blitz_hack.c*.

Но теперь мы хотим отдать компилятору **набор исходных файлов** и сказать: «Иди и сделай из них программу». Как нам этого добиться? Какой синтаксис компилятора `gcc` нужно использовать? И что более важно, *каким образом* компилятор создаст одну исполняемую программу из нескольких файлов? Как это будет работать? Как он свяжет их вместе?

Чтобы понять, каким образом компилятор может создавать одну программу из нескольких файлов, давайте посмотрим, как работает компиляция...

Взгляд на компиляцию изнутри

Чтобы понять, как компилятор объединяет несколько исходных файлов в одну-единственную программу, нужно заглянуть за кулисы и посмотреть, как на самом деле происходит компиляция.



1 **Прекомпиляция: исправление исходника.**
Первое, что нужно сделать компилятору, — это исправить исходник. Необходимо добавить все дополнительные заголовочные файлы, о которых было сказано, с помощью **директив** `#include`. Если потребуется, дополнить или пропустить некоторые участки программы. Как только это будет сделано, исходный код можно считать готовым для непосредственной компиляции.

Директива — это всего лишь причудливый синоним к слову «команда».

Он может делать это с помощью таких команд, как #define и #ifdef. Позже вы узнаете, как ими пользоваться.

2 **Компиляция: трансляция в ассемблер.**
Несмотря на то что Си кажется довольно низкоуровневым языком программирования, на самом деле этого уровня *недостаточно* для понимания компьютером. В действительности компьютер воспринимает очень низкоуровневые инструкции в **машинных кодах**. И прежде чем исходный код на Си превратится в машинный, он конвертируется в **символьные команды на языке ассемблер**:

```
movq  -24(%rbp), %rax
movzbl(%rax), %eax
movl  %eax, %edx
```

Выглядит довольно непонятно, правда? Язык ассемблера описывает отдельные инструкции, которые должны быть выполнены центральным процессором в ходе работы программы. У компилятора есть целый набор рецептов для каждого элемента языка Си. Благодаря этим рецептам компилятор знает, как преобразовать выражение или вызов функции в последовательность инструкций на ассемблере. Но даже ассемблер недостаточно низкоуровневый для компьютера. Вот почему ему нужно...

3 **Ассемблирование: генерирование объектного кода.**
Компилятору нужно будет *ассемблировать* символьные команды в *машинный*, или **объектный код**. По сути это двоичный код, который и будет выполняться внутри центрального процессора.

В машинном коде это довольно грязная шутка. →

```
10010101 00100101 11010101 01011100
```

Итак, на этом мы закончили? Оригинальный код на языке Си был преобразован в нолики и единички, которые нужны для компьютерных схем. Нет, остался еще один шаг. Если мы предоставим компьютеру несколько файлов для компиляции их в программу, компилятор сгенерирует отдельный объектный код для каждого из них. Но чтобы эти объектные файлы объединились в одну исполняемую программу, должно произойти еще кое-что…

4 **Линковка: все собирается в один файл.**
Получив отдельные фрагменты объектного кода, мы должны собрать их вместе, словно части головоломки, чтобы сформировать **исполняемую программу**. Компилятор соединит в одно целое части объектного кода, которые осуществляют вызовы функций друг из друга. Линковка также гарантирует, что программа сможет должным образом вызывать библиотечный код. В итоге код будет записан в исполняемый файл с использованием формата, который поддерживается операционной системой. Это важно, поскольку формат файла позволяет операционной системе загружать программу в память и выполнять ее.

Как же сказать gcc, что мы хотим сделать одну исполняемую программу из нескольких отдельных исходных файлов?

Для разделяемого кода потребуется отдельный заголовочный файл

Если мы собираемся использовать код из *encrypt.c* в разных программах, нам нужно будет как-то сообщить им об этом коде. Это возможно с помощью заголовочного файла.

Мы подключим заголовок внутри encrypt.c

```
void encrypt(char *message);
```

encrypt.h

```
#include "encrypt.h"

void encrypt(char *message)
{
  char c;
  while (*message) {
    *message = *message ^ 31;
    message++;
  }
}
```

encrypt.c

Нам необходимо подключить encrypt.h в нашу программу

Здесь мы используем заголовочный файл, поэтому можем поменять функции местами. Мы можем **сообщить другим приложениям о функции `encrypt()`**:

```
#include <stdio.h>
#include "encrypt.h"

int main()
{
  char msg[80];
  while (fgets(msg, 80, stdin)) {
    encrypt(msg);
    printf("%s", msg);
  }
}
```

message_hider.c

Мы подключим encrypt.h, чтобы программа имела объявление функции encrypt().

Мы поместили *encrypt.h* в главную программу, чтобы компилятор знал достаточно о функции `encrypt()` и мог скомпилировать код. На этапе линковки компилятор сможет связать вызов `encrypt(msg)` в файле *message_hider.c* с самой функций `encrypt()` из *encrypt.h*.

Наконец, чтобы собрать все вместе, нам всего лишь нужно передать `gcc` все исходные файлы:

```
gcc message_hider.c encrypt.c -o message_hider
```

Разделяемые переменные

Мы показали вам, как разделять функции между разными файлами. Но что если вы захотите создать общие переменные? Файлы с исходным кодом, как правило, содержат свои собственные переменные, благодаря чему удается избежать конфликта переменных с одинаковыми именами. Но если вы действительно хотите поделиться переменными, вам необходимо объявить их в своем заголовочном файле, используя в качестве префикса ключевое слово **`extern`**:

```
extern int passcode;
```

Тест-драйв

Давайте посмотрим, что произойдет, когда мы скомпилируем программу message_hider:

Мы должны скомпилировать код из двух исходных файлов.

Запустив программу, мы сможем ввести текст и увидеть его в зашифрованном виде.

Мы даже можем передать программе содержимое файла encrypt.h для шифрования.

```
File Edit Window Help Shhh...
> gcc message_hider.c encrypt.c -o message_hider
> ./message_hider
Я секретное сообщение
V?~r?~?1z|mzk?rz11~xz
> ./message_hider < encrypt.h
ipv{?zq|mfok7|w~m5?rz11~xz6$
>
```

Программа message_hider использует функцию encrypt() из encrypt.c.

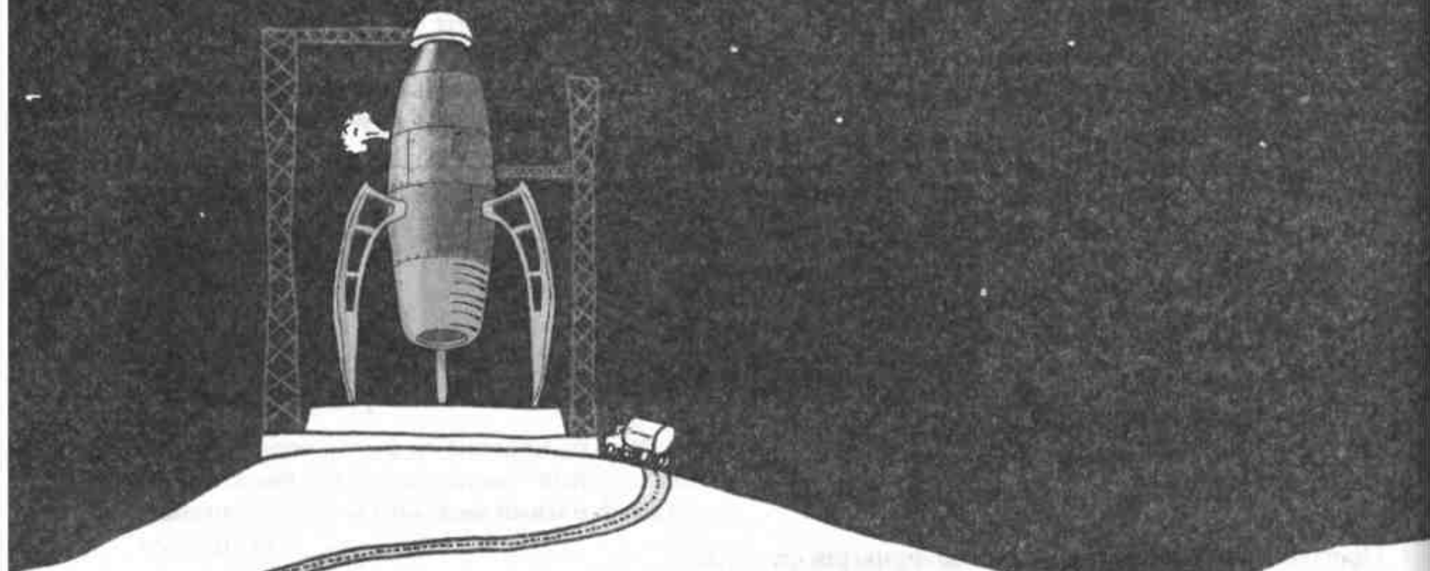
Программа работает. Теперь, когда функция encrypt() вынесена в отдельный файл, мы можем использовать ее в любой выбранной нами программе. Если мы когда-нибудь захотим внести изменения в функцию encrypt(), чтобы сделать ее чуть более безопасной, нам всего лишь понадобится исправить файл *encrypt.c*.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Вы можете делиться своим кодом, поместив его в отдельный .c-файл.
- Разместите объявление функции в отдельном заголовочном файле с расширением *.h*.
- Подключите заголовок к каждому исходному файлу, который будет использовать общий код.
- Перечислите все .c-файлы в команде компилятора.

Сходим с проложенной лыжни

Напишите свою собственную программу с использованием функции encrypt(). Не забывайте, что вы можете вызывать ее и для расшифровки текста.



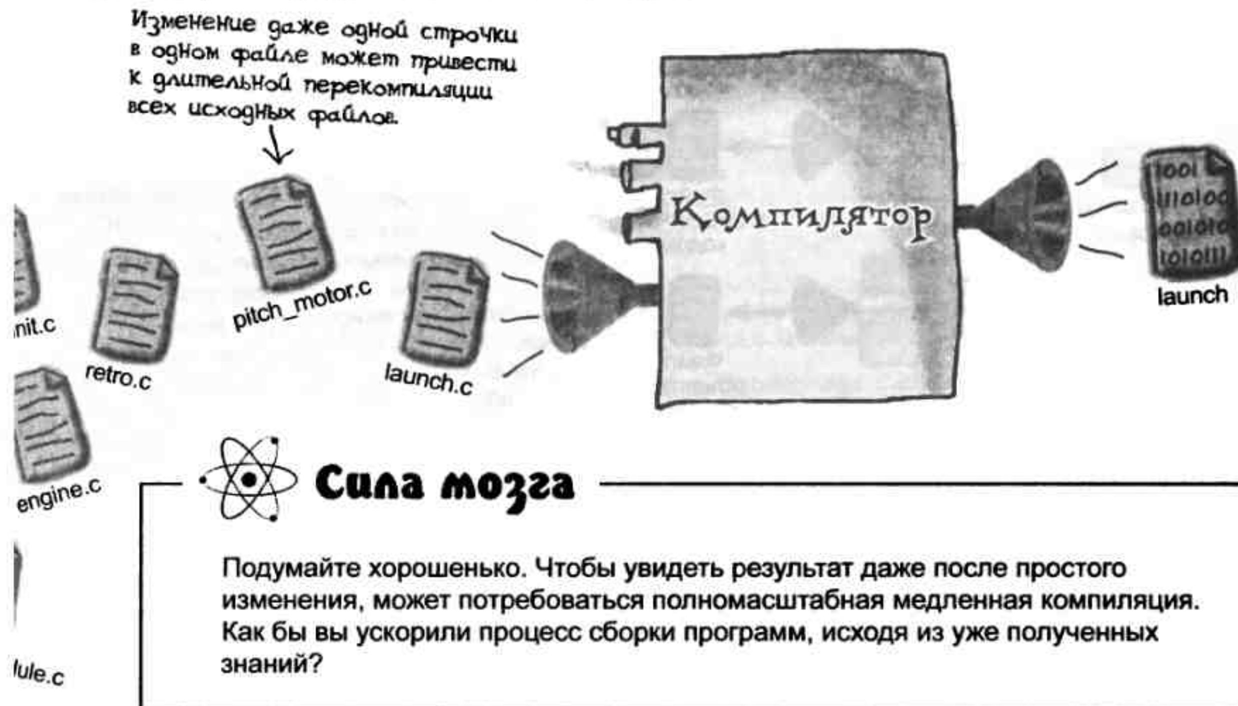
Слушай! Каждый раз, когда я вношу простое изменение в **один** файл, приходится ждать целую вечность, прежде чем все заново скомпилируется! А я ведь работаю по расписанию...
ullage_motor

Это ведь не космические технологии... правда?

Благодаря разделению вашего кода на отдельные исходные файлы, вы можете не только *делиться* им с разными программами, но и создавать *по-настоящему большие приложения*. Почему? Потому что вы можете дробить свою программу на маленькие **самодостаточные** фрагменты кода. Вместо того чтобы иметь дело с одним *большим* исходником, вы можете работать со множеством более *простых* файлов, которые легче понимать, поддерживать и тестировать.

Итак, вы можете создавать действительно большие программы — это преимущество. А как насчет недостатков? Недостаток заключается в том, что... вы можете создавать действительно большие программы. Компиляторы языка Си — очень эффективные программные продукты. Они применяют к вашим программам некоторые по-настоящему сложные трансформации: модифицируют ваш исходник, объединяют вместе сотни файлов, не израсходовав всю доступную память, и даже на лету оптимизируют написанный вами код. И несмотря на такое обилие возможностей, им все равно удается быстро работать.

Однако при создании программ на основе нескольких файлов время компиляции кода начинает играть важную роль. Скажем, компиляция большого проекта займет не больше минуты. Вроде бы немного, но этого достаточно, чтобы нарушить ход ваших мыслей. Изменив одну-единственную строчку кода, вы захотите увидеть результат как можно быстрее. И если каждый раз придется ждать целую минуту, это очень снизит темп работы.



Сила мозга

Подумайте хорошенько. Чтобы увидеть результат даже после простого изменения, может потребоваться полномасштабная медленная компиляция. Как бы вы ускорили процесс сборки программ, исходя из уже полученных знаний?

Не нужно перекомпилировать каждый файл

Если вы изменили всего один или два файла с исходным кодом, будет слишком неразумно перекомпилировать все исходники вашей программы. Подумайте, что будет, когда вы выполните команду, подобную следующей:

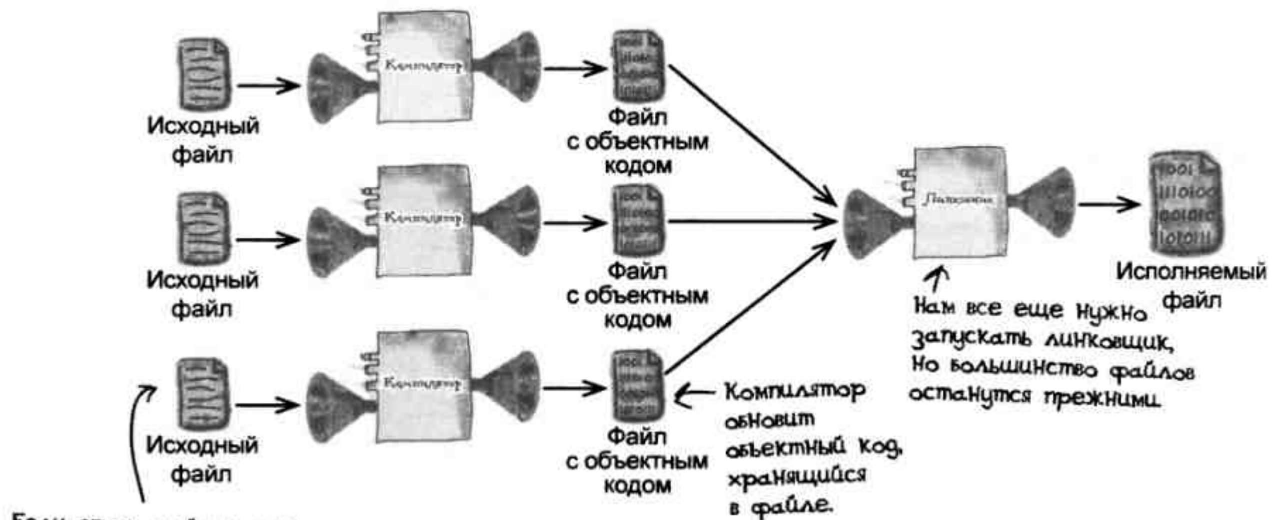
Здесь мы пропустили несколько имен файлов.

```
gcc reaction_control.c pitch_motor.c ... engine.c -o launch
```

Что же произойдет? *Для каждого исходного файла* запустится препроцессор, компилятор и ассемблер. Даже для тех, которые не менялись. И если код остался прежним, то и **объектный код**, сгенерированный для этих файлов, тоже не изменится. Так что же нам необходимо предпринять, если компилятор производит объектный код для всех файлов?

Вы можете сохранять копии скомпилированного кода

Если вы скажете компилятору сохранить сгенерированный им объектный код в файл, то его не нужно будет создавать заново, пока исходный код не изменится. Если исходник *изменился*, мы можем опять создать объектный код для **одного этого файла** и передать компилятору на линковку весь набор объектного кода.



Изменив один файл, мы должны будем заново создать для него объектный код, но нам *не придется* делать то же для всех остальных файлов. Затем мы можем передать все файлы с объектным кодом линковщику и создать новую версию программы.

Как заставить gcc сохранять объектный код в файл? И как после этого добиться, чтобы эти объектные файлы были слинкованы вместе компилятором?

Сначала вы компилируете исходный код в объектные файлы

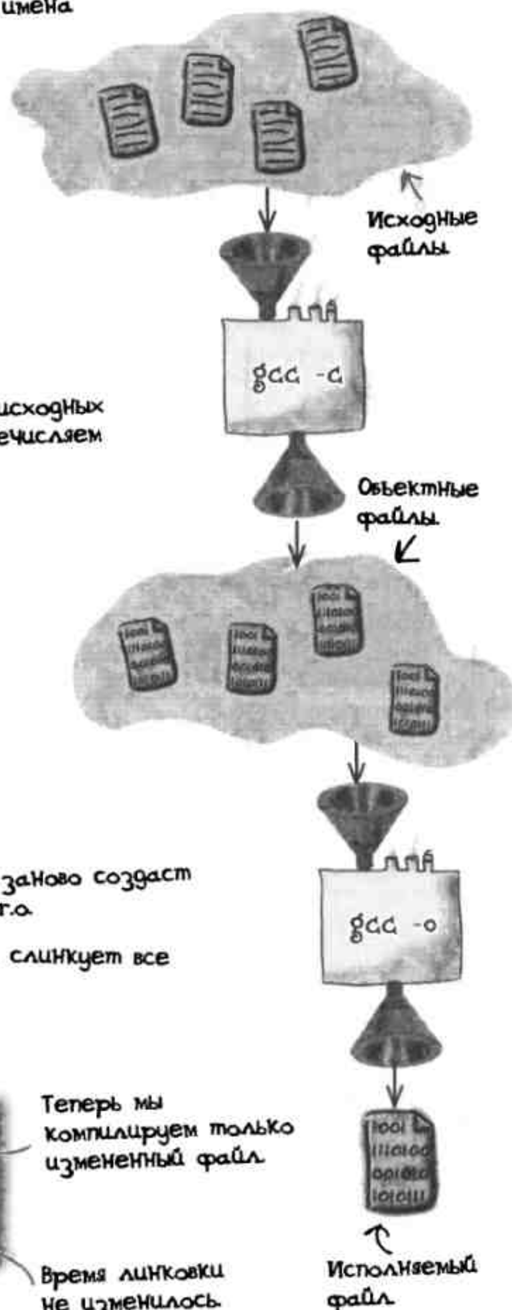
Команда gcc -c скомпилирует код, но не слинкует его.

Мы хотим получить объектный код для каждого исходного файла. Это делается с помощью следующей команды:

```
gcc -c *.c
```

Здесь создается объектный код для каждого файла. Операционная система подставит вместо *.c имена всех .c-файлов.

Вместо `*.c` будут подставлены все файлы с расширением *.c*, которые находятся в текущей директории. Ключ `-c` скажет компилятору, что вы хотите создать для каждого из них объектный файл и что при этом не нужно объединять их в полноценную исполняемую программу.



Затем вы объединяете их

Теперь, когда у нас есть набор объектных файлов, мы можем слинковать их вместе с помощью простой команды. Мы передадим их компилятору вместо имен исходников:

```
gcc *.o -o launch
```

Это похоже на команды компиляции, которые мы использовали ранее. Только вместо исходных файлов мы перечисляем объектные. Сюда будут подставлены все объектные файлы, находящиеся в директории.

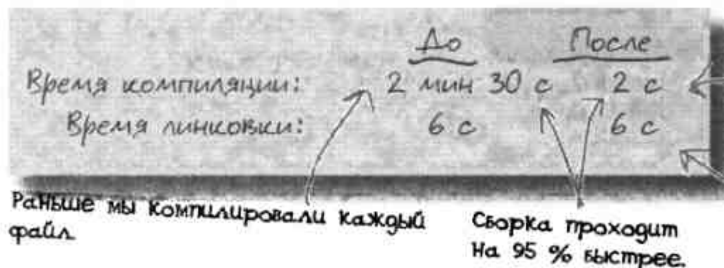
Компилятор достаточно умен, чтобы отличить объектные файлы от исходных, поэтому он пропустит большую часть этапов компиляции и просто объединит их в исполняемую программу под названием `launch`.

Что ж, теперь мы, как и прежде, имеем скомпилированную программу. Но вместе с тем у нас есть набор объектных файлов, которые уже были слинкованы. Таким образом, если мы изменим один из файлов, нам нужно будет перекомпилировать только его, а затем повторно слинковать приложение:

```
gcc -c thruster.c
gcc *.o -o launch
```

Это файл, который был изменен. Эта команда заново создаст файл thruster.o. Эта команда слинкует все вместе.

Даже несмотря на то, что нам приходится вводить две команды, мы экономим *кучу* времени.



	До	После
Время компиляции:	2 мин 30 с	2 с
Время линковки:	6 с	6 с

Теперь мы компилируем только измененный файл.

Время линковки не изменилось.

Большое упражнение

Здесь представлен некоторый код, используемый для управления системой двигателей ракеты. У каждого файла есть временная отметка. Как вы думаете, какие из них нужно создать заново, чтобы сделать актуальным исполняемый файл `ems`? Обведите файлы, которые, по вашему мнению, нуждаются в обновлении.

thruster.c
11:43

turbo.c
12:15

graticule.c
14:52

servo.c
13:47

thruster.o
11:48

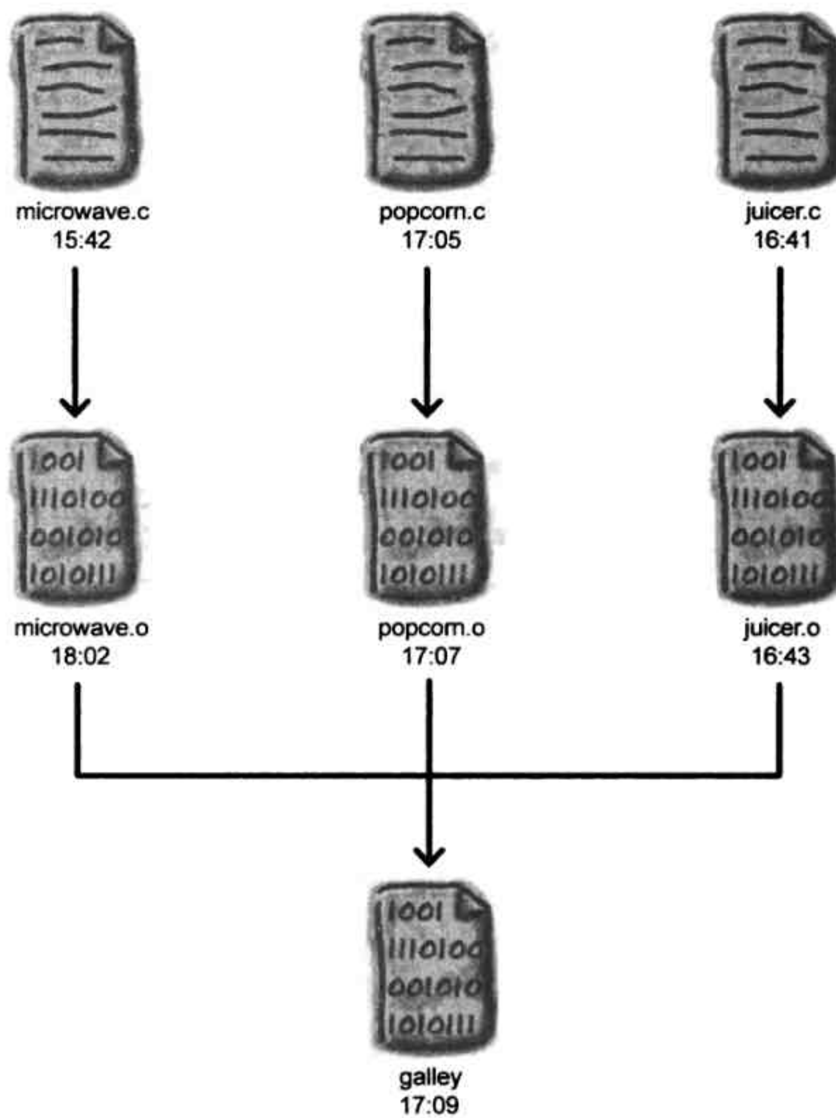
turbo.o
12:22

graticule.o
14:25

servo.o
13:46

ems
14:26

Мы также должны проверить код для приложения galley. Взгляните на время под файлами. Какие из этих файлов нужно обновить?



Большое упражнение
Решение

Здесь представлен некоторый код, используемый для управления системой двигателей ракеты. У каждого файла есть временная отметка. Вам нужно было обвести файлы, которые необходимо создать заново, чтобы сделать актуальным исполняемый файл `ems`.

thruster.c 11:43

turbo.c 12:15

graticule.c 14:52

servo.c 13:47

Файл graticule.o нужно перекомпилировать, поскольку он старше последней версии своего исходного кода.

Файл servo.o нужно перекомпилировать, поскольку он старше своего исходного кода.

thruster.o 11:48

turbo.o 12:22

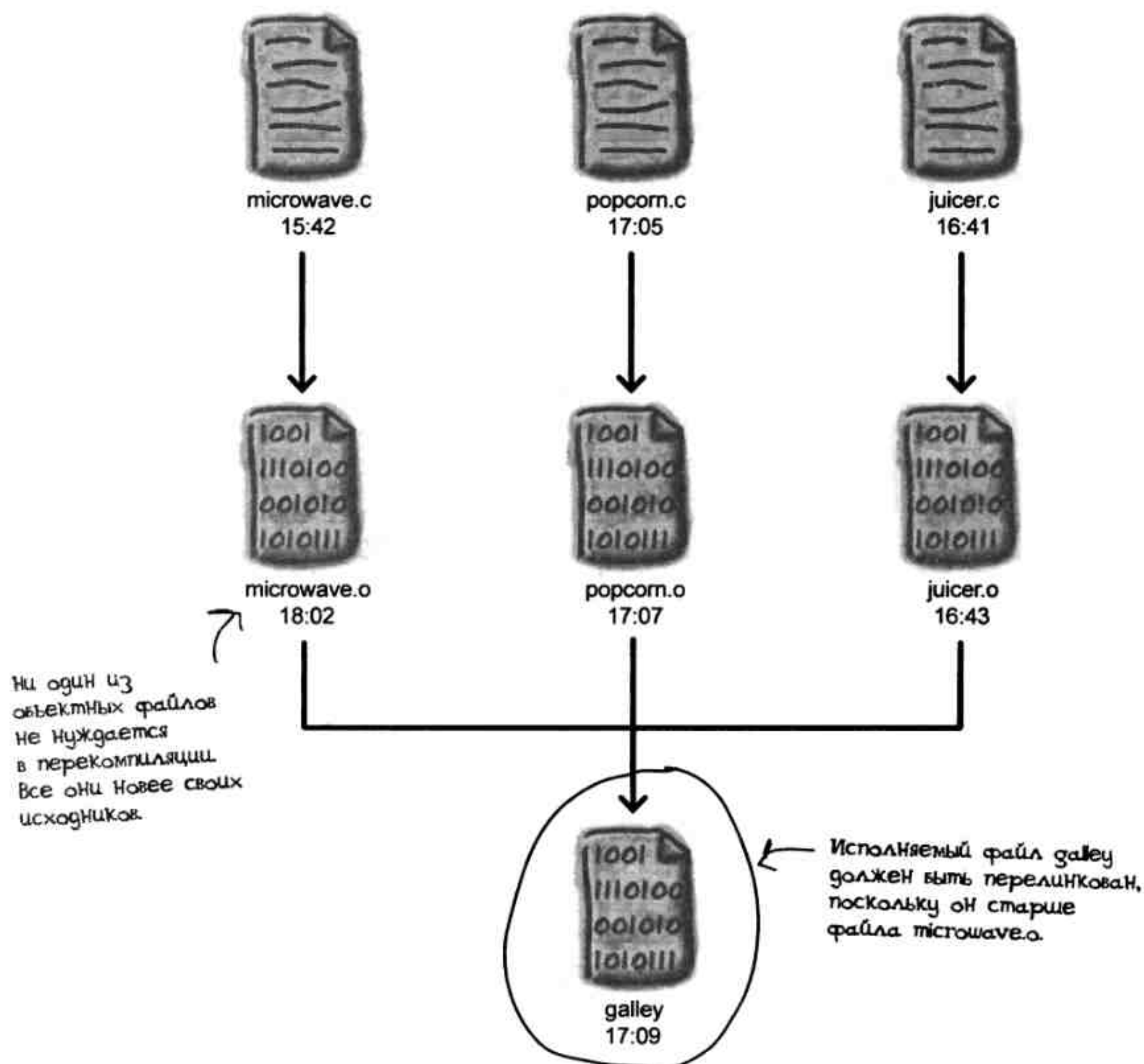
graticule.o 14:25

servo.o 13:46

ems 14:26

Раз мы изменили graticule.o и servo.o, то нам нужно перелинковать и исполняемый файл ems.

Вы также должны были проверить код для приложения galley и решить, какие из этих файлов нужно обновить.



Сложно уследить за всеми файлами

Это правда: частичная компиляция быстрее, но вам приходится более тщательно все обдумывать, чтобы не забыть перекомпилировать все необходимые файлы.

Если вы работаете только с одним исходным файлом, то все довольно просто. Но если таких файлов несколько, можно очень легко забыть перекомпилировать некоторые из них. Как следствие, заново скомпилированная программа не подхватит все внесенные вами изменения. Конечно, перед *поставкой* финальной версии приложения вы всегда можете выполнить полную перекомпиляцию *каждого* файла, но вам не захочется этого делать в процессе написания кода.

Даже несмотря на то что поиск файлов для компиляции — это **машинальная работа**, упустить какие-нибудь изменения очень легко, если заниматься этим вручную.

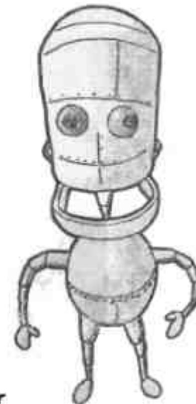
Можно ли как-то **автоматизировать этот процесс**?

Было бы здорово иметь инструмент, который мог бы автоматически перекомпилировать только измененные исходные файлы. Но это всего лишь мечты...

Автоматизируйте процесс сборки с помощью утилиты make

До тех пор пока вы успеваете следить за всеми измененными файлами, компиляция приложений будет проходить действительно быстро. Однако, несмотря на всю замысловатость процесса, его легко автоматизировать. Представьте, что у вас есть файл (например, объектный), сгенерированный из какого-то другого (например, исходного) файла:

Если файл thruster.c новее, вам нужно сделать перекомпиляцию. → **thruster.c** ⟶ **thruster.o** ← Если thruster.o новее, перекомпиляция вам не нужна.



Как узнать, что *thruster.o* нуждается в перекомпиляции? Нужно просто посмотреть на время изменения обоих файлов. Если файл *thruster.o* старше *thruster.c*, его нужно создать заново. Иначе он будет неактуален.

Это довольно простое правило. А если что-то работает по простому правилу, то не думайте об этом — **автоматизируйте**...

make — это инструмент, который может запускать команды компиляции вместо вас. Он проверит даты изменения исходных и сгенерированных файлов и перекомпилирует только те из них, которые являются устаревшими.

Но прежде чем заняться всем этим, нам нужно рассказать утилите об исходном коде. Она должна знать подробности о зависимостях между файлами и о том, как именно должна происходить сборка кода.

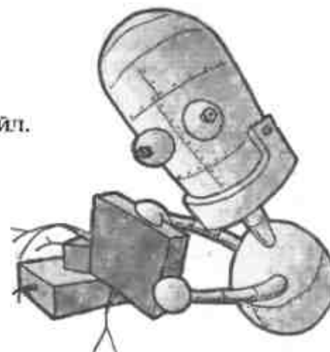
О чем должна знать утилита make?

Каждый файл, компилируемый утилитой `make`, называется **целевым** (target). Строго говоря, действие этой утилиты не ограничивается компиляцией файлов как таковых. Целевым может быть любой файл, *сгенерированный* из каких-либо других, например zip-архив, созданный вследствие сжатия набора файлов.

Каждая цель должна иметь *два атрибута*:

- **Зависимости.**
  Из каких файлов должен быть сгенерирован целевой файл.

- **Рецепт.**
  Набор инструкций, которые нужно выполнить для генерирования файла.



Зависимости и рецепт вместе составляют **правило**. Благодаря правилам утилита `make` знает все необходимое для генерирования целевого файла.

Как работает утилита make

Представим, что вы хотите скомпилировать *thruster.c* в какой-то объектный код в файле *thruster.o*. Что будет зависимостью, а что рецептом?

```
thruster.c ——> thruster.o
```

Файл *thruster.o* является ***целевым***, так как именно его мы хотим сгенерировать. Файл *thruster.c* — это зависимость, потому что он нужен для создания *thruster.o*. А что же будет рецептом? Команда компиляции, которая превратит *thruster.c* в *thruster.o*.

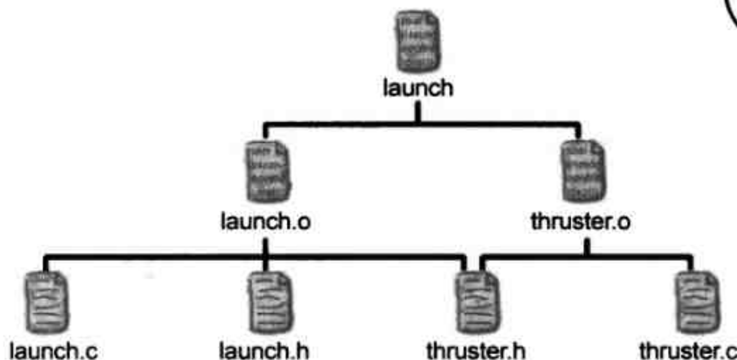
```
gcc -c thruster.c
```
← Это правило для создания thruster.o.

Разумно? Так как утилита `make` знает о зависимостях и рецепте, она сама может решать, нуждается ли файл *thruster.o* в перекомпиляции.

Но мы можем пойти дальше. Как только будет создан файл *thruster.o*, мы используем его для создания программы `launch`. Это значит, что файл `launch` тоже будет указан в качестве целевого, ведь мы хотим его сгенерировать. В качестве зависимостей для `launch` будут выступать объектные файлы, а рецептом станет команда:

```
gcc *.o -o launch
```

После того как мы сообщим утилите `make` все подробности о зависимостях и правилах, нам останется только сказать ей, чтобы она создала файл `launch`. Об остальном она сама позаботится.



Но как мы сообщим утилите make о зависимостях и рецептах? Давайте узнаем.

Будьте осторожны!

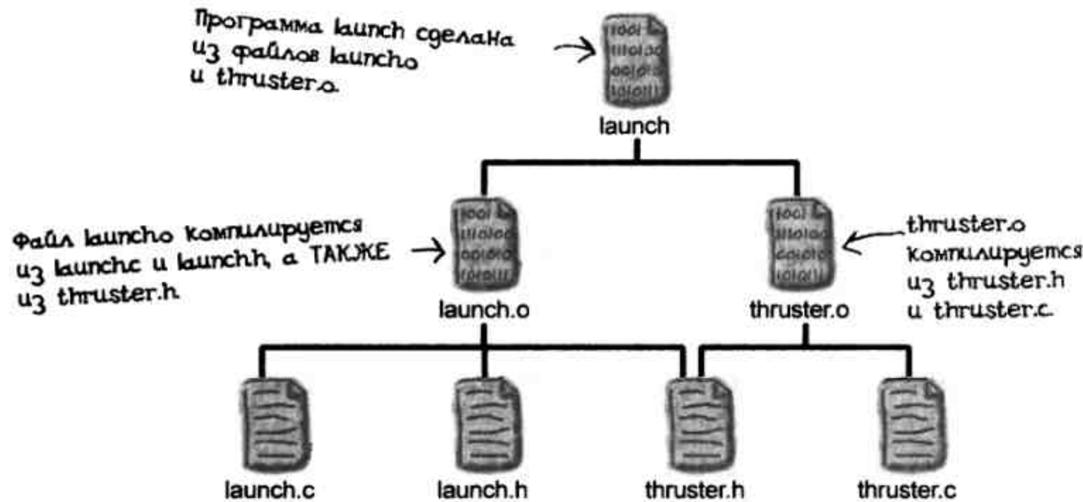
В Windows утилита `make` может называться по-другому.

Поскольку утилита `make` *пришла из мира Unix, в системе Windows она имеет несколько разновидностей. MinGW включает в себя версию под названием* `mingw32-make`, *а у Microsoft есть свой вариант, который называется* `NMAKE`.

Итак, я должен скомпилировать программу launch? Хм... Сначала мне необходимо перекомпилировать файл thruster.o, так как он устарел. Затем нужно заново слинковать исполняемый файл.

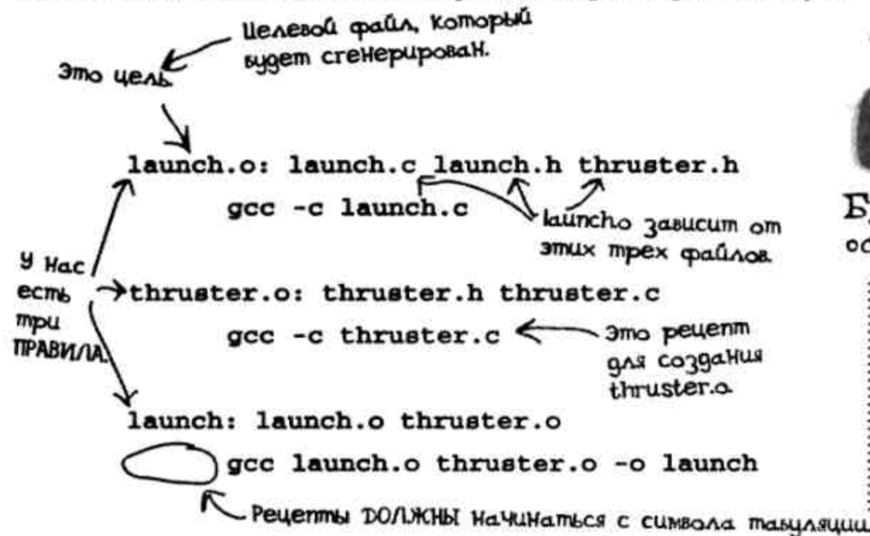
Сообщите утилите make о своем коде с помощью файла makefile

Все подробности о целях, зависимостях и рецептах должны быть размещены в файле под названием *makefile* (или *Makefile*). Чтобы понять, как это работает, представьте, что у вас есть пара исходных файлов, из которых создается программа `launch`.



Программа `launch` создана путем линковки файлов *launch.o* и *thruster.o*. А те, в свою очередь, компилируются из соответствующих исходных и заголовочных файлов. При этом *launch.o* также зависит от файла *thruster.h*, поскольку в нем содержится код, необходимый для вызова функции.

Вот как можно было бы описать процесс сборки в файле *makefile*:



Будьте осторожны!

Все рецепты ДОЛЖНЫ начинаться с символа табуляции.

Если в строчке с рецептом вы попытаетесь сделать отступ с помощью пробелов, сборка работать не будет.

Тест-драйв

Сохраните этот текст в файл под названием *Makefile* в ту же директорию, а затем откроете консоль и введете следующее:

```
File Edit Window Help MakeItSo
> make launch
gcc -c launch.c
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

Мы говорим утилите make создать файл launch

Сначала утилита make должна создать launch.o с помощью этой строчки

Затем утилите make необходимо сгенерировать thruster.o с помощью этой строчки

Наконец утилита make линкует объектные файлы, создавая программу launch

Как видите, утилита `make` смогла выполнить последовательность команд, необходимых для создания программы `launch`. Но что случится, если мы внесем изменения в файл *thruster.c* и запустим `make` снова?

```
File Edit Window Help MakeItSo
> make launch
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

Утилите make больше не нужно компилировать файл launch.c.

launch.o до сих пор актуален.

Утилита `make` может пропустить создание новой версии *launch.o*. Она просто скомпилирует *thruster.o*, после чего перелинкует программу.

не бывает Глупых Вопросов

В: Утилита make похожа на программу ant, которую я использовал в Java?

О: Точнее, это инструменты для сборки вроде ant и rake похожи на make. Утилита make была одним из самых первых средств для автоматической сборки программ из исходного кода.

В: Похоже, что для компиляции исходного кода приходится проделывать слишком много работы. Настолько ли полезна утилита make?

О: Да, она невероятно полезна. В небольших проектах она не сэкономит слишком уж много времени, но, когда у вас будет нечто большее, чем просто горстка файлов, компиляция и линковка кода могут стать очень утомительными.

В: Если я создам makefile для Windows, будет ли он работать на компьютерах с Mac или Linux?

О: Поскольку в таких файлах вызываются команды конкретной операционной системы, то иногда на другой операционной системе это может не сработать.

В: Могу ли я использовать утилиту make для чего-нибудь, кроме компиляции кода?

О: Да. Этот инструмент обычно используется для компиляции кода. Но он также может выступать в роли консольного инсталлятора или программы для управления исходниками. На самом деле утилита make пригодна для любой задачи, которую можно выполнять в командной строке.

БАЙКИ ИЗ СКЛЕПА

Почему в отступах используется табуляция?

Отступы легче делать с помощью пробелов. Так почему же в рецептах используется табуляция? Это цитата от создателя утилиты make Стюарта Фельдмана:

«Почему в первом столбце табуляция? ...Это сработало, поэтому так и осталось. А через несколько недель количество пользователей, большинство из которых — мои друзья, перевалило за дюжину, и мне не хотелось ломать базовые вещи. Остальное, к сожалению, уже история».

Уголок ботана

Утилита make освобождает вас от множества забот, связанных с компиляцией файлов. Но, если вам покажется, что даже она привносит недостаточно автоматизма, взгляните на инструмент под названием **autoconf**:

http://www.gnu.org/software/autoconf/

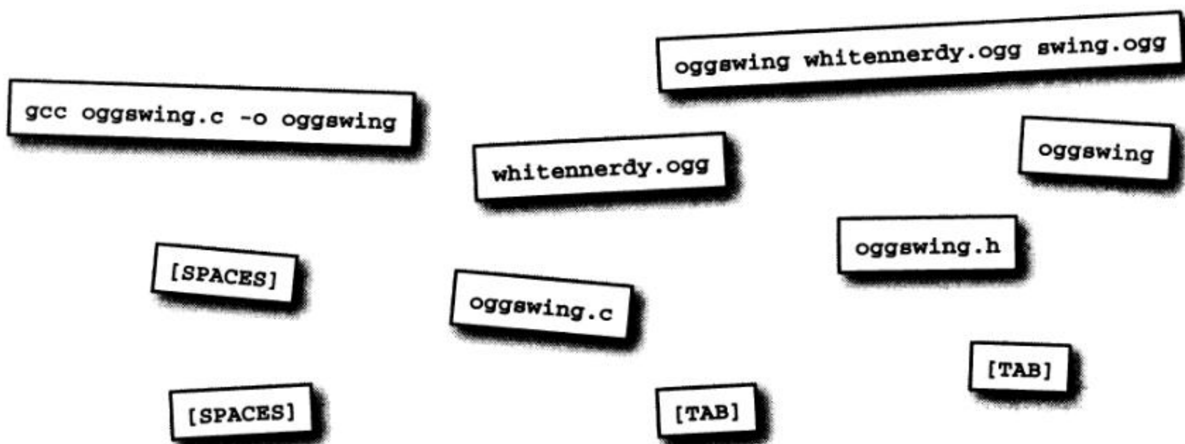
autoconf используется для генерации файлов makefile. Программисты на Си часто создают инструменты, чтобы автоматизировать процесс создания программного обеспечения. Число таких процессов растет, и многие из них доступны на веб-сайте организации GNU.

Магнитики с кодом

Эй, крошка, если ты еще не успела оценить последние музыкальные хиты, то ты просто *влюбишься* в программу, которую написали парни из бара Head First! `oggswing` — это программа, считывающая музыкальные файлы в формате Ogg Vorbis и создающая их свинговые версии! Замечательно! Посмотрите, сможете ли вы дописать *makefile*, который ее компилирует, чтобы потом сконвертировать *.ogg*-файл.

Здесь whitennerdy.ogg конвертируется в swing.ogg.

```
oggswing: ...................................................


swing.ogg:
```



Магнитики с кодом Решение

Эй, крошка, если ты еще не успела оценить последние музыкальные хиты, то ты просто *влюбишься* в программу, которую написали парни из бара Head First! oggswing — это программа, считывающая музыкальные файлы в формате Ogg Vorbis и создающая их свинговые версии! Вам нужно было дописать *makefile*, который ее компилирует, чтобы потом сконвертировать *.ogg*-файл:

```
oggswing: oggswing.c oggswing.h
[TAB] gcc oggswing.c -o oggswing

swing.ogg: whitennerdy.ogg oggswing
[TAB] oggswing whitennerdy.ogg swing.ogg
```

[SPACES]

[SPACES]

Уголок ботана

Утилита make способна на гораздо большее, чем мы успели уже обсудить. Подробнее с ней самой и ее возможностями вы можете ознакомиться в руководстве *GNU Make Manual* по адресу:

http://tinyurl.com/yczmjx

Пуск!

Если процесс сборки проходит действительно медленно, то утилита make заметно его ускорит. Большинство разработчиков настолько привыкли собирать код с ее помощью, что используют ее и для небольших программ. Это все равно что постоянно иметь рядом с собой по-настоящему заботливого программиста. Если у вас есть объемный код, то утилита make позаботится о том, чтобы в сборке участвовала только та его часть, которая нужна вам в определенный момент времени.

Иногда важно выполнить все вовремя...

КЛЮЧЕВЫЕ МОМЕНТЫ

- Компиляция большого количества файлов может занять существенное время.
- Скорость компиляции можно повысить, сохраняя объектный код в файлы **.o*.
- `gcc` может компилировать программы как из исходных, так и из объектных файлов.
- Утилита `make` может быть использована для автоматизации процесса сборки.
- Утилита `make` знает о зависимостях между файлами, поэтому она может компилировать только те из них, которые изменились.
- О процессе сборки утилита `make` узнает из файла *makefile*.
- Будьте внимательны при форматировании *makefile* — не забывайте, что в качестве отступов используется табуляция, а не пробелы.

Ваш инструментарий языка Си

Вы изучили главу 4, пополнив свои знания информацией о типах данных и заголовочных файлах. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

Тип char хранит числа.

Используйте тип long для по-настоящему больших целых чисел.

Используйте тип float для большинства чисел с плавающей точкой.

Используйте тип short для небольших целых чисел.

Используйте тип int для большинства целых чисел.

Используйте тип double для дробных чисел с высокой точностью.

Отделяйте объявление функции от ее определения.

Выносите объявления в заголовочный файл.

Сохраняйте объектный код в файлы, чтобы повысить скорость сборки.

Директива #include<> нужна для подключения библиотечных заголовков.

Директива #include"" нужна для подключения локальных заголовков.

Используйте утилиту make для управления процессом сборки.

Имя: **Дата:**

Лабораторная работа 1

Arduino

В этой лабораторной работе содержится описание программы, которую вы должны создать, применив полученные при чтении нескольких последних глав знания.

Пока это самый большой проект из тех, с которыми вы имели дело. Поэтому, прежде чем начинать, не торопитесь и внимательно все прочтите. Не переживайте, если у вас что-то не получится, — здесь нет никаких новых сведений о языке Си, если нужно, вы можете продолжить чтение книги и вернуться к этой лабораторной работе позже.

Мы предоставили вам некоторые подробности о структуре программы, сделав так, чтобы у вас было все необходимое для написания кода. Вы даже можете создать настоящее устройство.

Выполнение этой задачи целиком и полностью зависит от вас, и кода для ее решения у вас не будет.

Задание: сделайте так, чтобы ваше комнатное растение заговорило

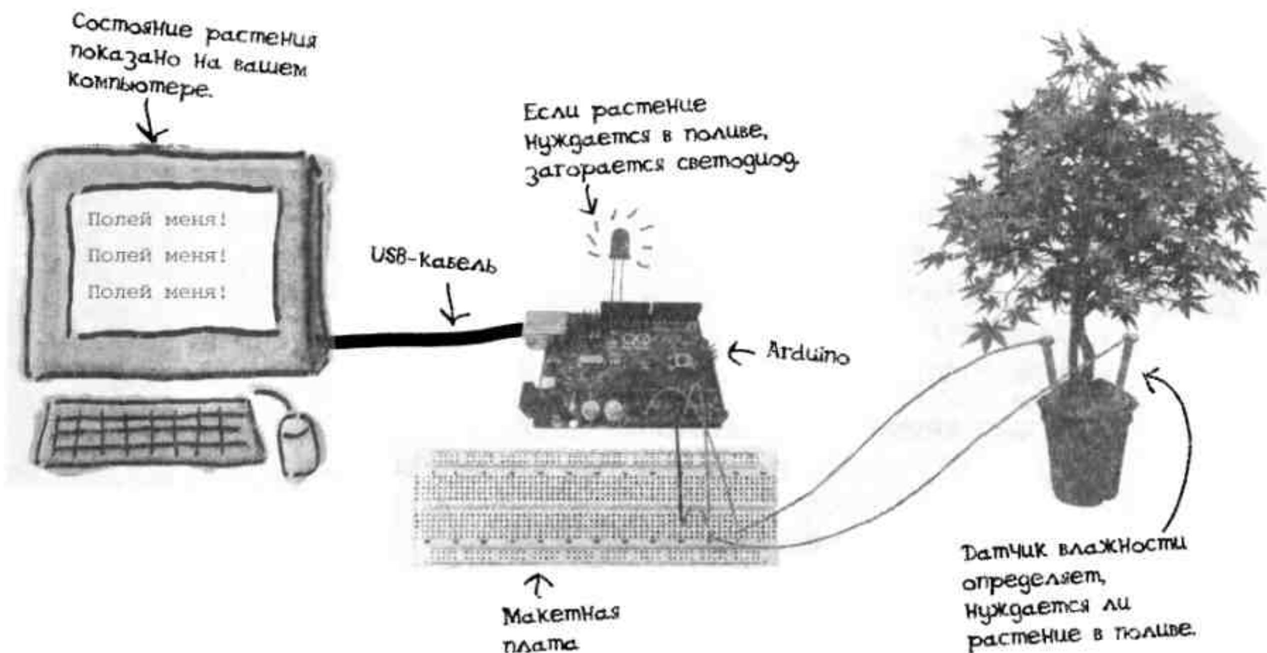
Мечтали ли вы когда-нибудь о том, чтобы ваши растения сами говорили, когда их нужно поливать? Что ж, с помощью Arduino они могут это делать! В этой лабораторной работе вы напишете систему мониторинга растений на основе Arduino на языке Си.

Вам предстоит сделать следующее.

Физическое устройство

У Arduino есть специальный датчик, измеряющий влажность почвы вашего растения. Если необходим полив, загорится светодиод, а на ваш компьютер периодически будет посылаться сообщение «Полей меня!».

После полива светодиод погаснет, а на ваш компьютер придет разовое сообщение «Спасибо, Сеймур!».



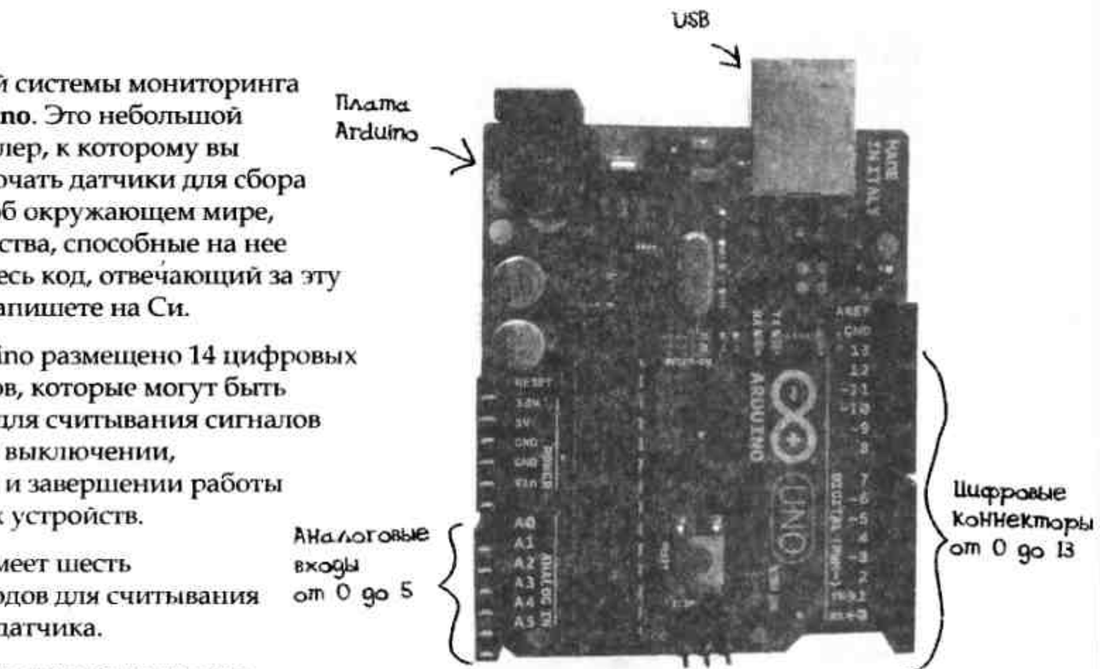
Arduino

Мозгом данной системы мониторинга является **Arduino**. Это небольшой микроконтроллер, к которому вы можете подключать датчики для сбора информации об окружающем мире, а также устройства, способные на нее реагировать. Весь код, отвечающий за эту реакцию, вы напишете на Си.

На плате Arduino размещено 14 цифровых входов/выходов, которые могут быть использованы для считывания сигналов о включении и выключении, а также начале и завершении работы подключенных устройств.

Плата также имеет шесть аналоговых входов для считывания напряжения с датчика.

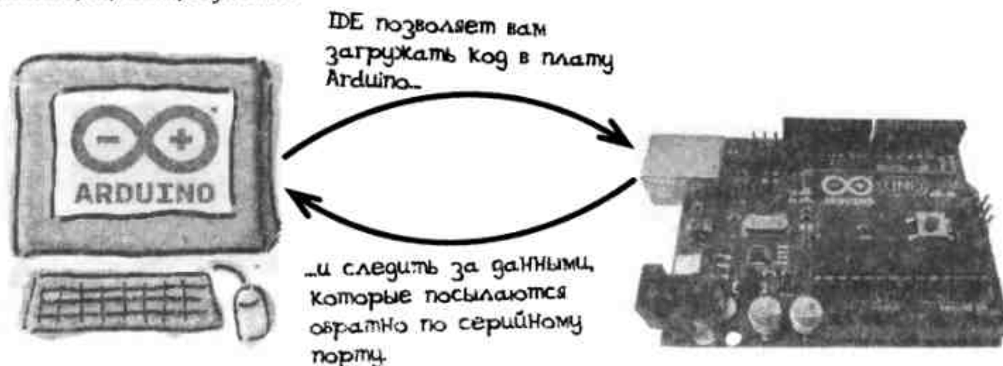
Arduino может питаться от вашего компьютера через USB-порт.



Arduino IDE

Программу на Си вы будете писать с помощью Arduino IDE. Эта среда разработки позволяет проверять и компилировать код, а также загружать его в Arduino через USB-порт. Кроме того, Arduino IDE включает в себя серийную консоль, с помощью которой вы можете видеть данные, отсылаемые Arduino обратно (если таковые имеются).

Arduino IDE бесплатна, вы можете получить ее по адресу *www.arduino.cc/en/Main/Software*.



Собираем физическое устройство

Вы начнете с создания физического устройства на основе Arduino. И хотя эта часть не обязательна, настоятельно рекомендуем ее выполнить. Ваши комнатные растения будут вам за это благодарны.

Мы использовали Arduino Uno.

Вам понадобятся:

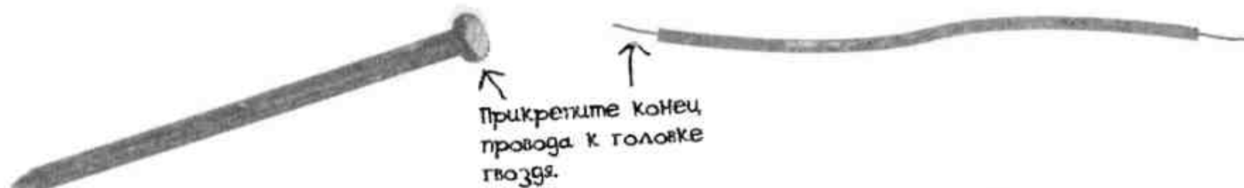
- 1 контроллер Arduino;
- 1 макетная плата;
- 1 светодиод;
- 1 резистор на 10 кОм;
- 2 оцинкованных гвоздя;
- 3 коротких провода-переходника;
- 2 длинных провода-переходника.

Создаем датчик влажности

Возьмите длинный провод-переходник и присоедините его к головке одного из оцинкованных гвоздей, припаяв или просто намотав на гвоздь.

После этого присоедините другой отрезок провода к еще одному оцинкованному гвоздю.

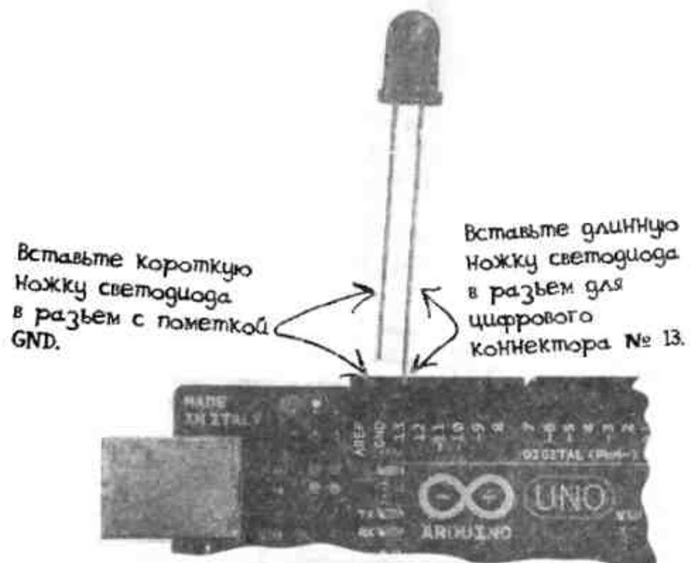
Принцип работы датчика влажности заключается в измерении электропроводимости между двумя гвоздями. Высокая проводимость должна означать высокую влажность, и наоборот.



Взгляните на светодиод. Вы увидите, что одна его ножка длиннее (плюс), а другая короче (минус).

Теперь давайте внимательно рассмотрим Arduino. На одном из его краев расположены разъемы для 14 цифровых коннекторов, пронумерованных от 0 до 13. Сразу за ними находится надпись GND. Вставьте положительный контакт светодиода в разъем под номером 13, а отрицательный — в разъем с пометкой GND.

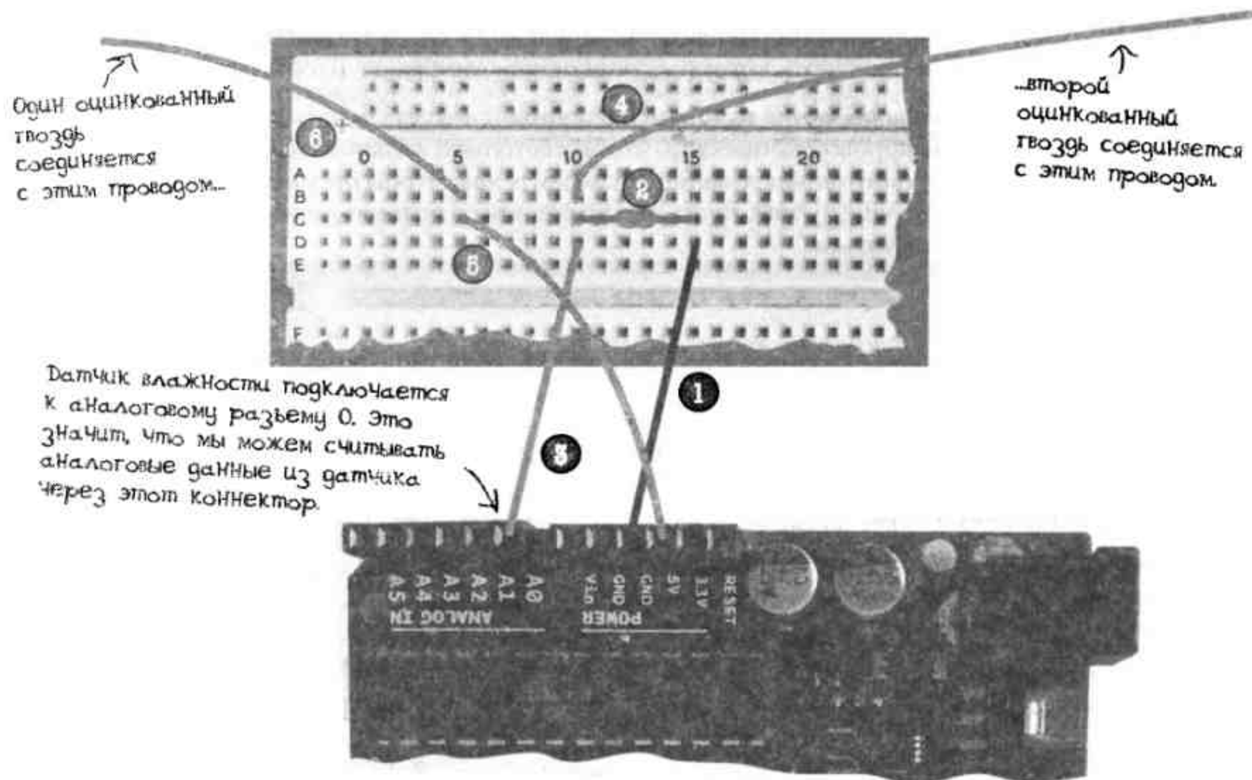
В результате светодиодом можно будет управлять с помощью цифрового коннектора № 13.



Подключаем датчик влажности

Подключите датчик влажности так, как описано ниже.

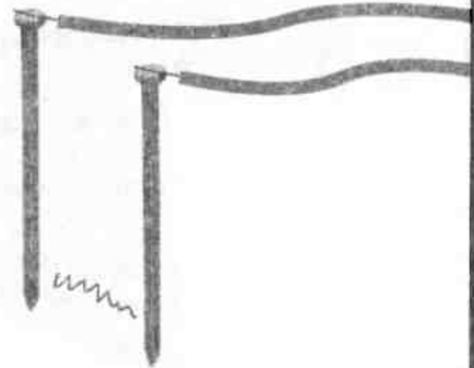
1. Соедините коротким проводом коннектор GND на контроллере Arduino с разъемом D15 на макетной плате.
2. Вставьте резистор на 10 кОм в разъемы макетной платы C15–C10.
3. Соедините коротким проводом аналоговый вход 0 с разъемом D10 на макетной плате.
4. Возьмите один из оцинкованных гвоздей и вставьте присоединенный к нему провод в разъем B10.
5. Соедините коротким проводом коннектор 5V на контроллере Arduino с разъемом C5 на макетной плате.
6. Возьмите другой оцинкованный гвоздь и вставьте присоединенный к нему провод в разъем B5.



Вот так выглядит аппаратная часть устройства. Теперь перейдем к коду на Си...

Вот что должен делать ваш код

Ваш код на языке Си для Arduino должен делать следующее.



Считывать показатели датчика влажности

Датчик влажности подключен к аналоговому входу. Вам нужно считывать из этого входа аналоговые значения.

В нашей лаборатории мы выяснили, что обычно растения нужно поливать, если влажность опускается ниже 800. Но у ваших домашних растений (особенно у кактуса) могут быть другие потребности в воде.

Производить запись в светодиод

Светодиод подключен к цифровому коннектору.

Когда растению больше не нужна вода, следует сделать запись в коннектор, к которому подключен светодиод, чтобы тот погас.

Когда растение нуждается в поливе, следует сделать запись в цифровой коннектор, чтобы светодиод включился. В качестве дополнительного задания сделайте так, чтобы он мигал. Или, еще лучше, пусть он мигает, когда условия вот-вот должны измениться.

Производить запись в серийный порт

Когда придет время полить растение, нужно начать регулярную отправку строки «Полей меня!» на серийный порт компьютера.

Когда у растения появится достаточно воды, в серийный порт делается одиночная запись строки «Спасибо, Сеймур!».

Предполагается, что Arduino подключен к компьютеру через USB.

Вот как должен выглядеть ваш код на Си

Программы на Си для Arduino имеют особенную архитектуру. Ваша программа должна реализовать следующее:

```
void setup()
{
/*Эта функция вызывается при запуске программы.
В сущности, она делает настройку платы. Размещайте
здесь любой код, связанный с инициализацией.*/
}
void loop()
{
/*Здесь размещается ваш главный код. Эта функция
выполняется в цикле снова и снова, позволяя вам
реагировать на ввод, полученный из датчиков. Он
останавливается только при выключении платы*/
}
```

Если хотите, можете добавить дополнительные функции и объявления, но без этих двух функций код работать не будет.

Проще всего писать код для Arduino с помощью Arduino IDE. Эта среда позволяет проверять и компилировать программы, а затем загружать их непосредственно на плату, чтобы посмотреть их в работе.

Arduino IDE поставляется вместе с библиотекой функций для Arduino и содержит множество полезных примеров с кодом. Переверните страницу, и вы увидите список наиболее полезных функций, необходимых при программировании этого микроконтроллера.

Вот несколько полезных функций для Arduino

При написании программы вам понадобятся некоторые из этих функций.

`void pinMode(int pin, int mode)`

Сообщает Arduino, в каком режиме будет работать цифровой коннектор — на ввод или на вывод. `mode` может равняться либо `INPUT`, либо `OUTPUT`.

`int digitalRead(int pin)`

Считывает значение из цифрового коннектора. Может возвращать либо `HIGH`, либо `LOW`.

`void digitalWrite(int pin, int value)`

Записывает значение в цифровой коннектор. *`value`* может равняться либо `HIGH`, либо `LOW`.

`int analogRead(int pin)`

Считывает значение из аналогового коннектора. Возвращаемое значение находится в диапазоне между 0 и 1023.

`void analogWrite(int pin, int value)`

Записывает значение в аналоговый коннектор. *`value`* находится в диапазоне между 0 и 1023.

`void Serial.begin(long speed)`

Говорит Arduino начать отправку и прием серийных данных. Скорость указывается в битах в секунду. Обычно *`speed`* принимает значение 9600.

`void Serial.println(val)`

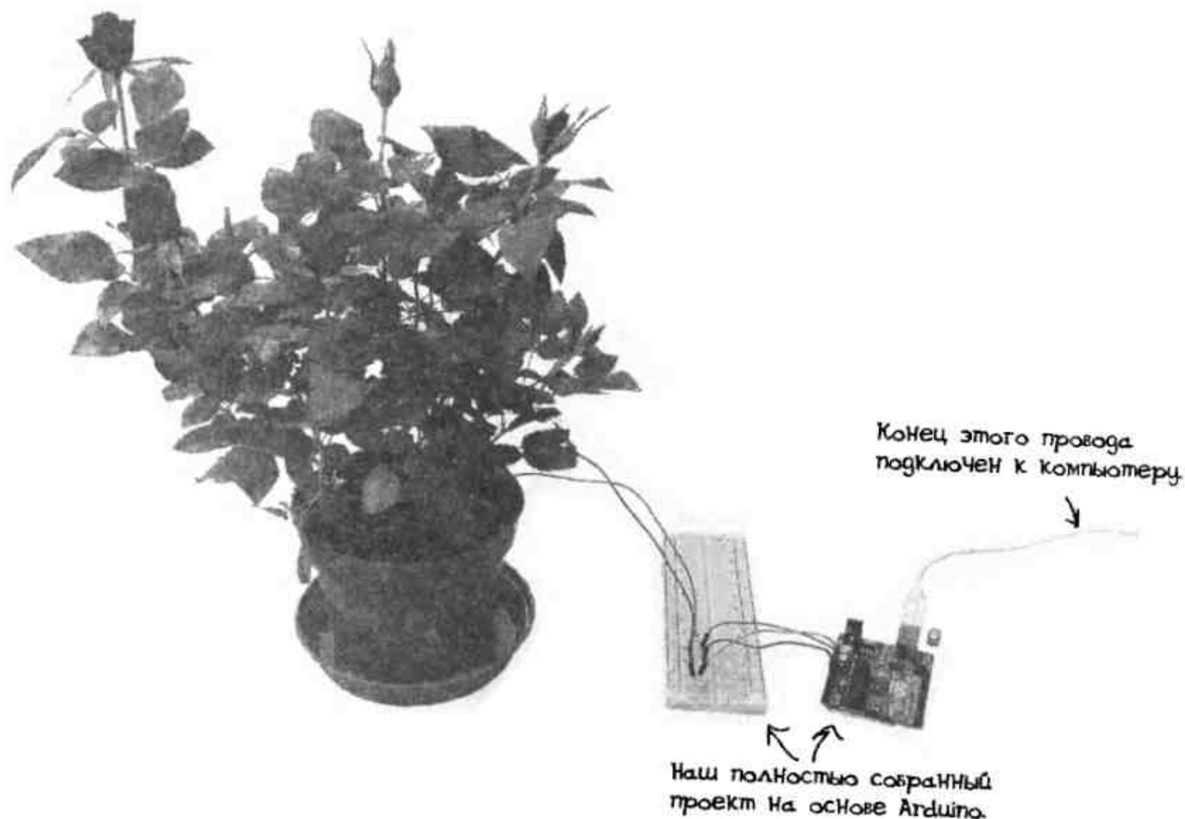
Выводит данные в серийный порт. *`val`* может иметь любой тип.

`void delay(long interval)`

Приостанавливает программу на интервал, измеряемый в миллисекундах.

Конечный продукт

Ваш проект можно считать завершенным, когда вы поместите датчик влажности в почву, подключите Arduino к компьютеру и начнете получать информацию о своем растении.



Если хотите, чтобы ваше растение действительно заговорило, можете загрузить с сайта нашей лаборатории скрипт, который будет выводить поток серийных данных на вашем компьютере: www.headfirstlabs.com/books/hfc

5 Структуры, объединения и битовые поля

Создавайте собственные структуры

В жизни далеко не все можно свести к простым числам.

До сих пор мы рассматривали базовые типы данных языка Си, но что если вы хотите выйти за пределы чисел и фрагментов текста и **смоделировать что-то, имеющее отношение к реальному миру**? Это можно сделать с помощью ***структур***, создавая свои собственные типы. Мы покажем, как **базовые типы данных** могут сочетаться в структурах, и даже рассмотрим так называемые ***объединения***, с помощью которых можно **передать неопределенность, встречающуюся в реальной жизни**. А если вам нужны простые ответы «да» или «нет», то ***битовые поля*** могут оказаться именно тем, что вам необходимо.

Иногда приходится иметь дело с большими объемами данных

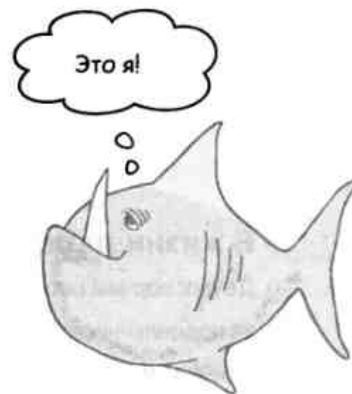
Вы уже знаете, что Си может работать со множеством разных типов данных: маленькими и большими числами, числами с плавающей точкой, символами и текстом. Но довольно часто, записывая информацию, имеющую отношение к чему-то из реального мира, необходимо использовать сразу несколько фрагментов данных. Рассмотрим следующий пример. У нас есть *две* функции, которые имеют дело с одними и теми же *вещами* из реального мира, принимая одинаковый набор данных:

const char* означает, что мы будем передавать строковые литералы.

Обе эти функции принимают один и тот же набор параметров.

```
/* Выводим запись, содержащуюся в каталоге */
void catalog(const char *name, const char *species, int teeth, int age)
{
  printf("("%s - это %s с %i зубами. Ему %i года\n",
    name, species, teeth, age);
}

/* Выводим ярлык для аквариума */
void label(const char *name, const char *species, int teeth, int age)
{
  printf("Кличка:%s\nРазновидность:%s\n%i года, %i зубов\n",
    name, species, teeth, age);
}
```

Пока что все не так уж и плохо, правда? Но уже из-за того, что вы передаете четыре фрагмента данных, код начинает выглядеть немного запутанно:



Мы дважды передаем те же четыре фрагмента данных.

```
int main()
{
  catalog("Зубастик", "пиранья", 69, 4);
  label("Зубастик", "пиранья", 69, 4);
  return 0;
}
```

Рыба всего одна, но мы передаем четыре фрагмента данных.

Так как же решить эту проблему? Как можно избежать передачи множества данных, описывая что-то одно?

Разговор в офисе

Джо: Конечно, это *сейчас* у нас четыре фрагмента, но что будет, если мы поменяем систему, чтобы записать очередной фрагмент данных о рыбе?

Фрэнк: Это всего лишь *еще один параметр.*

Джил: Да, это всего лишь один фрагмент данных, но нам придется добавлять его в *каждую функцию*, которой потребуется информация о рыбе.

Джо: Ага, особенно в большой системе, которая может состоять из *сотен* функций. И все из-за того, что мы добавили всего лишь *еще один фрагмент данных.*

Фрэнк: Верно подмечено. Как же нам этого избежать?

Джо: Легко. Мы просто сгруппируем данные в *единую сущность*. Что-то наподобие массива.

Джил: Не уверена, что это сработает. Массивы обычно хранят список данных *одного типа.*

Джо: Верно.

Фрэнк: Понятно. Мы записываем строки и целые числа, но не можем поместить все это в один массив.

Джил: Я думаю, что нет.

Джо: Да ладно, должен же в Си быть какой-то способ сделать это. Давайте подумаем, что нам нужно.

Фрэнк: Хорошо, нам нужно нечто такое, что позволило бы ссылаться сразу на целый набор данных с разными типами, как будто это всего лишь одна переменная.

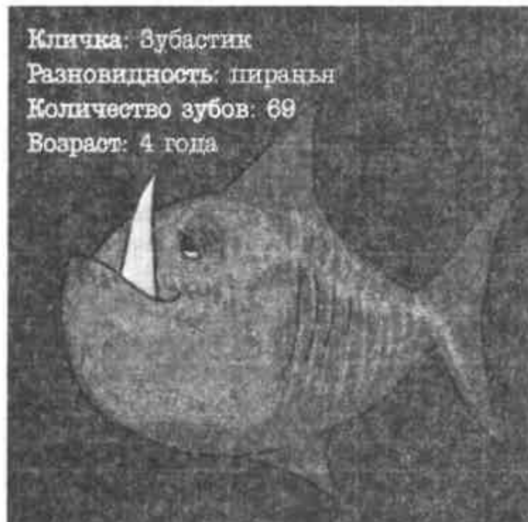
Джил: Не думаю, что мы сталкивались прежде с чем-то подобным, не так ли?

Нам нужно что-то такое, что позволило бы записывать набор разной информации в *один единственный фрагмент данных.*

Создавайте собственные структурированные типы данных с помощью структур

Если у вас есть набор данных, которые нужно объединить во *что-то одно*, вы можете использовать структуры. Ключевое слово `struct` — это сокращенное обозначение **структурированного типа данных**. Структуры позволяют все разрозненные переменные «завернуть» в один большой новый тип данных. Например:

```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
};
```



Тем самым мы создадим новый тип данных, основанный на совокупности других фрагментов данных. Фактически, он чем-то напоминает массив, за исключением того, что:

- **Он имеет фиксированную длину.**
- **Фрагменты данных внутри структуры имеют имена.**

Но как использовать данные, из которых состоит описанная вами структура? Что ж, это очень похоже на создание нового массива. Просто нужно убедиться, что вы передаете фрагменты данных в том порядке, в котором они определены внутри структуры:

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
```

struct fish — это тип данных.

snappy — это имя переменной.

Это кличка.

Это разновидность.

Это количество зубов.

Это возраст Зубастика.

не бывает Глупых Вопросов

В: Эй, погодите. Что это за `const char`?

О: `const char *` используется для строк, которые вы не хотите менять, то есть чаще всего для записи строковых литералов.

В: Хорошо. Значит, эта структура хранит строку?

О: В нашем случае нет. Здесь структура содержит всего лишь указатель на строку. Это означает, что она записывает адрес, по которому строка находится в памяти.

В: Но ведь там при желании можно хранить и всю строку целиком?

О: Да, если вы объявите символьный массив подобным образом: `char name[20];`.

Просто дайте им рыбу

Теперь, вместо того чтобы предоставлять функциям целую охапку отдельных фрагментов данных, вы можете просто передать нашу новую структуру:

```
/* Выводим запись, содержащуюся в каталоге */
void catalog(struct fish f)
{
  ...
}

/* Выводим ярлык для аквариума */
void label(struct fish f)
{
  ...
}
```

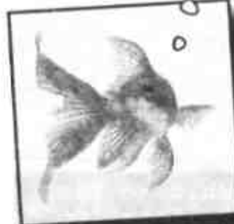
Это выглядит куда проще, не так ли? Не только потому, что функциям теперь нужен всего лишь *один фрагмент данных*, но и потому, что сам код, вызывающий эти функции, стало легче читать:

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
catalog(snappy);
label(snappy);
```

Таким образом, вы можете описывать собственные типы данных. Но как их потом *использовать*? Как наши функции смогут считывать отдельные фрагменты данных внутри структуры?

Сохраняя параметры внутри структур, вы делаете свой код более стабильным.

Эй, я хорооооошая!



Почему структура fish – это то, что вам нужно

Одно из значительных преимуществ передачи данных внутри структур заключается в том, что вы можете изменять содержимое структуры, не затрагивая функции, которые ее используют. Представим, например, что вы хотите добавить в `fish` дополнительное поле:

```
struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
    int favorite_music;
};
```

Функции `catalog()` и `label()` знают, что им будет передана структура `fish`. Но эти функции не знают (и их не волнует), что она теперь содержит дополнительные данные до тех пор, пока все нужные им поля на месте.

Из этого следует, что структуры не просто упрощают код для восприятия, но и позволяют лучше справляться с изменениями.

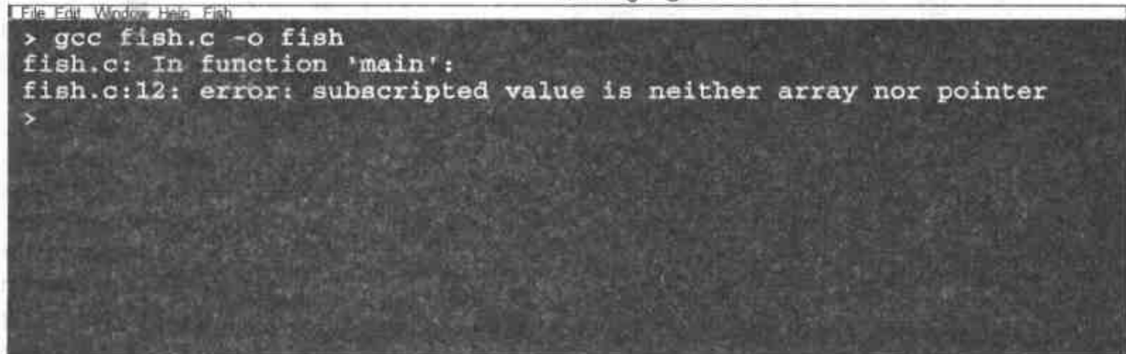
Вы можете считывать поля структуры с помощью оператора «точка»

Так как структуры в чем-то похожи на массивы, вы можете подумать, что доступ к их полям осуществляется схожим образом:

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
printf("Кличка = %s\n", snappy[0]);
```

Вы могли бы получить доступ к первому полю переменной snappy, если бы она была указателем на массив.

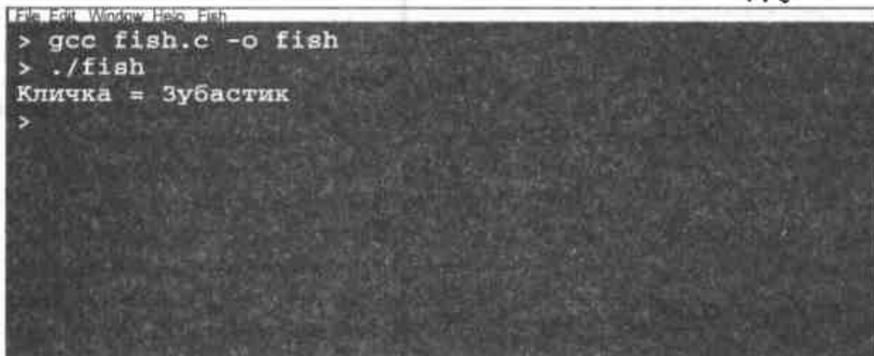
Попытавшись считать поля структуры так, будто это массив, вы получите ошибку.



Но это не так. Даже несмотря на то, что структуры хранят свои поля подобно массивам, получить к ним доступ можно только ***по имени*** с помощью оператора «точка». Если вы уже использовали языки вроде JavaScript или Ruby, вам это покажется знакомым:

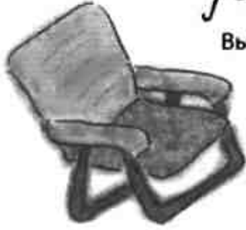
```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
printf("Кличка = %s\n", snappy.name);
```

Это атрибут name внутри snappy.



Он вернет строку "Зубастик".

Хорошо, теперь мы знаем кое-что об использовании структур. Давайте вернемся назад и посмотрим, можно ли обновить тот код...

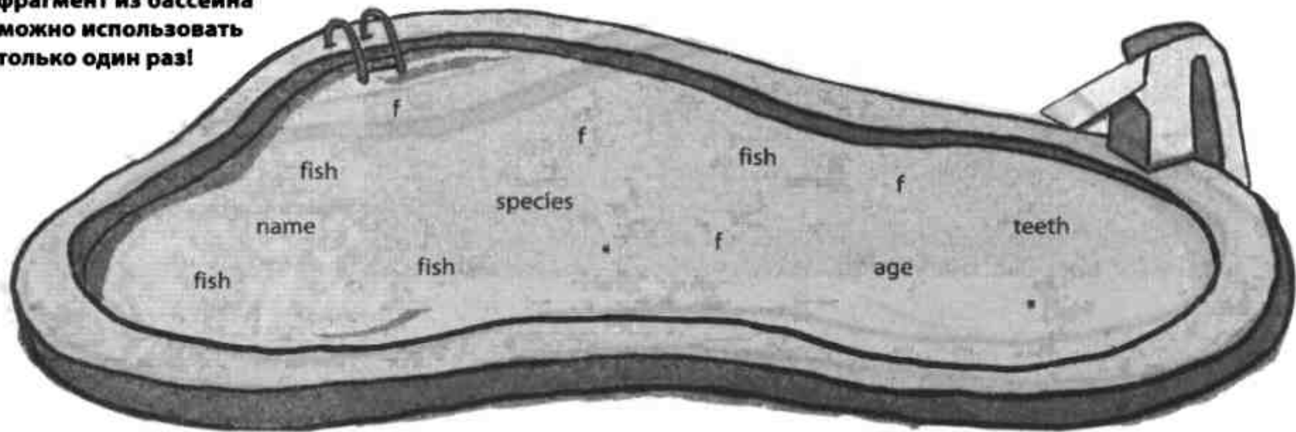


Головоломка ~~у бассейна~~ с пираньей

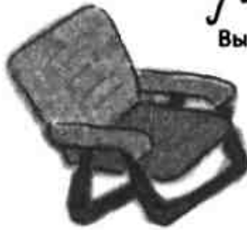
Вы должны написать новую версию функции `catalog()`, используя структуру `fish`. Заполните пустые места фрагментами кода из бассейна. Каждый из них может быть использован не более одного раза. Учтите, что вам понадобятся не все фрагменты.

```
void catalog(struct fish f)
{
  printf(("%s - это %s с %i зубами. Ему %i года\n",
    ......·.........., ......·.........., ......·.........., ......·.........);
}

int main()
{
  struct fish snappy = {"Зубастик", "пиранья", 69, 4};
  catalog(snappy);
  /* Пока что мы пропускаем вывод ярлыка */
  return 0;
}
```

Примечание: каждый фрагмент из бассейна можно использовать только один раз!

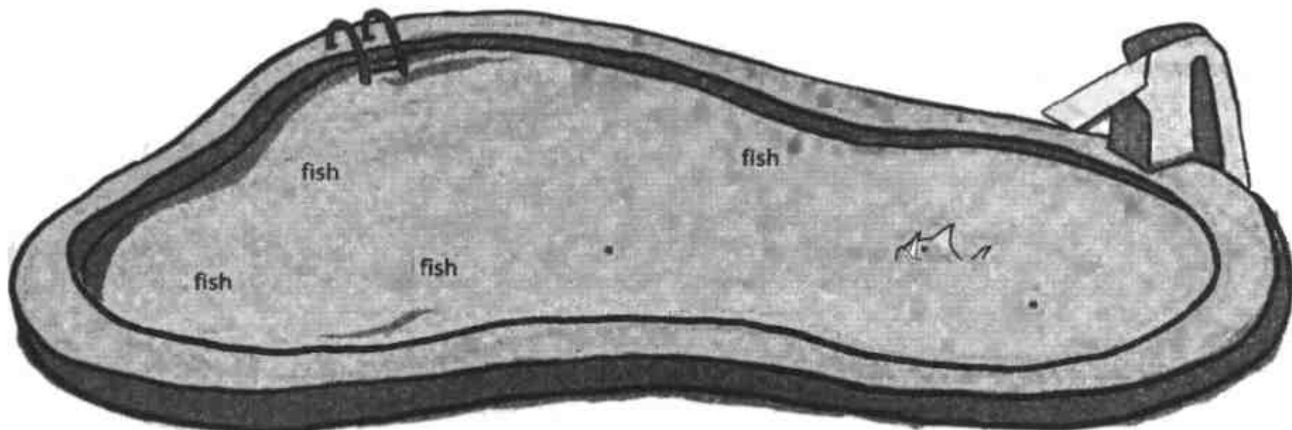


Головоломка ~~у бассейна~~ с пираньей. Решение



Вы должны были написать новую версию функции `catalog()`, используя структуру `fish` и заполняя пустые места фрагментами кода из бассейна.

```
void catalog(struct fish f)
{
  printf("%s - это %s с %i зубами. Ему %i года\n",
    f.name , f.species , f.teeth , f.age );
}

int main()
{
  struct fish snappy = {"Зубастик", "пиранья", 69, 4};
  catalog(snappy);
  /* Пока что мы пропускаем вывод ярлыка */
  return 0;
}
```



Тест-драйв

Переписав функцию `catalog()`, вы легко можете сделать то же самое с функцией `label()`. После чего скомпилировать программу и убедиться, что она по-прежнему работает.

Эй, смотрите, кто-то использует утилиту make...

```
> make pool_puzzle && ./pool_puzzle
gcc pool_puzzle.c -o pool_puzzle
Зубастик — это пиранья с 69 зубами. Ему 4 года
Кличка:Зубастик
Разновидность:пиранья
4 года, 69 зубов
>
```

Эта строчка напечатана функцией catalog().

Эти строчки напечатаны функцией label().

Отлично. Код работает, как и прежде, но теперь вызовы наших функций занимают две простые строчки:

```
catalog(snappy);

label(snappy);
```

Но дело не только в том, что код стало легче читать. Если мы когда-нибудь решим дополнить структуру данными, нам не понадобится изменять функции, в которых эта структура используется.

не бывает Глупых Вопросов

В: Структура — это просто массив?

О: Нет. Но, как и массив, она объединяет набор данных.

В: Переменная массива — это просто указатель на массив. Можно ли то же самое сказать о структуре?

О: Нет. Переменная структуры — это имя самой структуры.

В: Я знаю, что не стоит этого делать, но могу ли я использовать [0], [1], ... для доступа к полям структуры?

О: Нет, получать доступ к полям можно только через их имена.

В: Структуры — это как классы в других языках?

О: Они похожи, но добавление методов в структуру — процесс не из легких.

Подробнее о структурах в памяти

Во время присваивания копируется не сама строка, а указатель на нее.

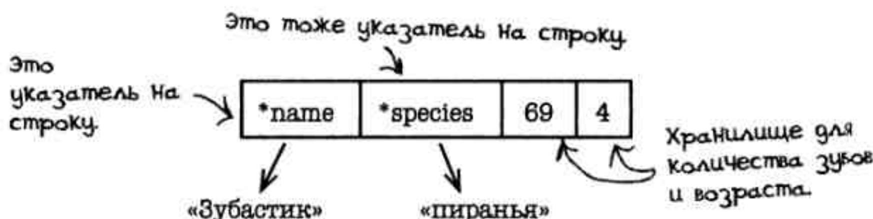
Когда вы присваиваете одну структуру другой, ее содержимое копируется. Но если структура включает в себя указатели, то в результате присваивания будут копироваться только значения этих указателей. Таким образом, поля `name` *и* `species`*, принадлежащие переменным* `gnasher` *и* `snappy`*, будут указывать на одни и те же строки.*

Когда вы определяете структуру, в памяти компьютера ничего не создается. Вы просто даете компьютеру **шаблон**, показывающий, как должен выглядеть новый тип данных.

```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
};
```

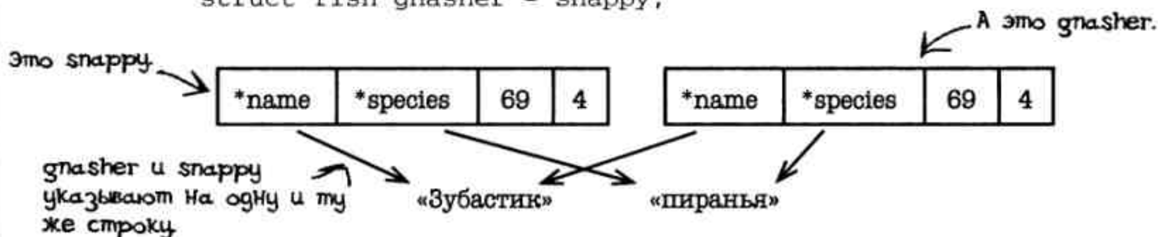
Но при объявлении новой переменной компьютеру необходимо выделить в памяти место для **экземпляра** структуры. Этого места должно быть достаточно для того, чтобы вместить все поля внутри структуры:

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
```



Что, по-вашему, происходит, когда вы присваиваете структуру другой переменной? Компьютер создаст **совершенно новую копию структуры**. Ему нужно будет выделить еще один участок памяти того же размера, после чего скопировать туда каждое поле.

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4};
struct fish gnasher = snappy;
```



Запомните: когда вы присваиваете переменные с типом структуры, вы заставляете компьютер *копировать* данные.

Можно ли поместить одну структуру внутрь другой?

Не забывайте, что при описании структуры мы на самом деле создаем *новый тип данных*. Си предоставляет множество встроенных типов, таких как `int` и `short`, но с помощью структур мы можем их объединять, чтобы описывать компьютеру *более сложные объекты*.

Если структура создается на основе существующих типов данных, это значит, что вы можете **создать ее и из других структур**. Чтобы увидеть, как это работает, давайте рассмотрим пример.

```
struct preferences {      <- Это то, что любит наша рыба.
  const char *food;
  float exercise_hours;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

Это структура внутри другой структуры.

Это новое поле.

Это называется вложенностью.

Наше новое поле называется care, но оно будет содержать поля, описанные в структуре preferences.

> **Зачем нужны вложенные структуры?**
>
> С их помощью можно уменьшить **сложность** кода. Структуры предоставляют более крупные *блоки данных*. Объединяя их вместе, вы получаете возможность создавать структуры все больших и больших размеров. Вполне вероятно, поначалу вы будете использовать только простые типы вроде `int` и `short`, структуры же позволят вам описать очень сложные вещи, такие как **сетевые потоки** или **видеоизображения**.

Этот код говорит компьютеру о том, что одна структура содержит другую. Итак, вы можете создавать переменные так же, как вы это делали в случае с массивами, но теперь у вас появилась возможность добавлять данные для структуры, которая находится внутри другой структуры:

Это данные поля care, которое является структурой.

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4, {"Мясо ", 7.5}};
```

Это значение для care.food

Это значение для care.exercise_hours

Соединив структуры вместе, вы можете получать доступ к их полям с помощью операторов «точка»:

```
printf("Зубастик любит кушать %s", snappy.care.food);
printf("Зубастик любит заниматься %f часов", snappy.care.exercise_hours);
```

Давайте проверим ваши навыки работы со структурами...

Большое упражнение

Парни, обслуживающие аквариум Head First, начали записывать множество данных о каждой рыбе, которая в нем обитает. Вот их структуры:

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

Это данные, которые будут записаны для одной из рыб.

```
Кличка: Зубастик
Разновидность: пиранья
Рацион: мясо
Количество пищи: 0.09 кг
Описание упражнения: купаться в джакузи
Продолжительность упражнения: 7.5 часов
```

Вопрос № 0: Как вы изобразите эти данные на языке Си?

```
struct fish snappy = ..........................................................................................
```

Вопрос № 1: Допишите код функции `label()`, чтобы она выводила следующее:

```
Кличка: Зубастик
Разновидность: пиранья
4 года, 69 зубов
Давать 0.09 кг мяса и разрешать купаться в джакузи на протяжении 7.50 часов
```

```
void label(struct fish a)
{
  printf("Кличка:%s\nРазновидность:%s\n%i года, %i зубов\n",
         a.name, a.species, a.teeth, a.age);
  printf("Давать %2.2f кг %s и разрешать %s на протяжении %2.2f часов\n",
         .................................................. , .................................................. ,
         .................................................. , .................................................. );
}
```

Большое упражнение
Решение

Парни, обслуживающие аквариум Head First, начали записывать множество данных о каждой рыбе, которая в нем обитает. Вот их структуры:

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

Это данные, которые будут записаны для одной из рыб.

```
Кличка: Зубастик
Разновидность: пиранья
Рацион: мясо
Количество пищи: 0.09 кг
Описание упражнения: купаться в джакузи
Продолжительность упражнения: 7.5 часов
```

Вопрос № 0: Как вы изобразите эти данные на языке Си?

```
struct fish snappy = {"Зубастик", "пиранья", 69, 4, {{"мясо", 0.2}, {" купаться в джакузи", 7.5}}};
```

Вопрос № 1: Допишите код функции `label()`, чтобы она выводила следующее:

```
Кличка: Зубастик
Разновидность: пиранья
4 года, 69 зубов
Давать 0.09 кг мяса и разрешать купаться в джакузи на протяжении 7.50 часов
```

```
void label(struct fish a)
{
  printf("Кличка:%s\nРазновидность:%s\n%i года, %i зубов\n",
         a.name, a.species, a.teeth, a.age);
  printf("Давать %2.2f кг %s и разрешать %s на протяжении %2.2f часов\n",
         a.care.food.weight, a.care.food.ingredients,
         a.care.exercise.description, a.care.exercise.duration);
}
```

Хм... Все эти конструкции для работы со структурами кажутся немного громоздкими. Я должен использовать ключевое слово struct и при объявлении структуры, и при объявлении переменной. Интересно, можно ли это как-то упростить?

Вы можете назначить своей структуре подходящее имя с помощью спецификатора typedef.

Создавая переменные для встроенных типов данных, вы можете использовать простые и короткие имена, такие как `int` или `double`. Но при создании переменных для структур мы должны указывать ключевое слово `struct`.

```
struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
};
...
struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

Однако Си позволяет создавать для любой структуры **псевдоним**. Если добавить спецификатор **`typedef`** *перед* ключевым словом `struct` и указать **имя типа** *после* закрывающей скобки, то новый тип можно будет вызывать так, как вам нравится:

typedef означает, что мы собираемся присвоить структуре новое имя.

```
typedef struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
...
phone p = {5557879, "sinatra.png", 1.35};
```

phone станет псевдонимом для структуры cell_phone.

Теперь, увидев слово phone, компилятор воспримет его как struct cell_phone.

Спецификаторы `typedef` могут сделать ваш код более коротким и простым для чтения. Давайте посмотрим, как будет выглядеть наша программа, если вы начнете добавлять в нее спецификаторы `typedef`...

Как мне называть мой новый тип?

Если вы используете `typedef` при создании псевдонима для структуры, то необходимо решить, каким будет этот ***псевдоним***. Поскольку он представляет собой всего лишь имя вашего типа, то получается, что мы имеем дело с *двумя именами*: с именем структуры (`struct` **`cell_phone`**) и именем ***типа*** (**`phone`**). Зачем нам два имени? Как правило, вам достаточно будет указывать только имя типа — компилятор не будет возражать, если вы пропустите имя структуры:

```
typedef struct {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
phone p = {5557879, "s.png", 1.35};
```

Это псевдоним.

Упражнение

Пришло время аквалангисту провести ежедневный осмотр резервуаров, и ему нужна новая бирка для гидрокостюма. Но вот беда — кажется, часть кода куда-то пропала. Можете ли вы выяснить, какие слова были утеряны?

```
#include <stdio.h>

 ............................struct {
  float tank_capacity;
  int tank_psi;
  const char *suit_material;
} ............................;

 ............................struct scuba {
  const char *name;
  equipment kit;
} diver;

void badge(............................ d)
{
  printf("Кличка: %s Резервуар: %2.2f(%i) Костюм: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
 ............................ randy = {"Рэнди", {5.5, 3500, " неопрен"}};
  badge(randy);
  return 0;
}
```

Упражнение Решение

Пришло время аквалангисту провести ежедневный осмотр резервуаров, и ему нужна новая бирка для гидрокостюма. Но вот беда — кажется, часть кода куда-то пропала. Вы выяснили, какие слова были утеряны?

```
#include <stdio.h>

 typedef struct {
  float tank_capacity;
  int tank_psi;
  const char *suit_material;
} equipment;

 typedef struct scuba {
  const char *name;
  equipment kit;
} diver;

void badge(diver d)
{
  printf("Кличка: %s Резервуар: %2.2f(%i) Костюм: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
  diver randy = {"Рэнди", {5.5, 3500, " неопрен"}};
  badge(randy);
  return 0;
}
```

Здесь программист решил назвать структуру scuba. Но мы просто используем имя типа diver.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Структура — это тип данных, собранный из набора других типов данных.
- Структуры имеют фиксированную длину.
- Доступ к *полям* структур осуществляется по их имени с использованием *точки*.
- Поля структур хранятся в памяти в том же порядке, в котором они указаны в коде.
- Вы можете соединять структуры.
- Спецификатор `typedef` создает *псевдоним* для типа данных.
- Используя `typedef` в сочетании с ключевым словом `struct`, вы можете не указывать имя структуры.

не бывает Глупых Вопросов

В: Поля структур размещаются в памяти одно за другим?

О: Иногда они разделены небольшими промежутками.

В: Зачем?

О: Компьютеру нравится, когда данные не выходят за рамки так называемых слов. Если компьютер использует 32-битные слова, он не захочет, чтобы переменная типа `short`, к примеру, была разделена 32-битной границей.

В: Значит, он оставит промежуток и поместит начало переменной в следующем 32-битном слове?

О: Да.

В: Означает ли это, что каждое поле занимает целое слово?

О: Нет. Компьютер оставляет промежутки только для того, чтобы поля не выходили за границы слова. Если он сможет уместить несколько полей в одном слове, то он так и сделает.

В: Почему компьютер беспокоится о границах слов?

О: Он считывает из памяти целые слова. Если поле разделено между несколькими словами, процессору придется считывать несколько адресов и каким-то образом склеивать значение.

В: И это будет медленно?

О: Да, это будет медленно.

В: В таких языках, как Java, объекты не копируются в результате присваивания их переменным. Копируются только ссылки. Почему в Си все не так?

О: В Си *любое* присваивание приводит к копированию данных. Если вы хотите скопировать ссылку на фрагмент данных, вы должны присвоить указатель.

В: Я совсем запутался с именами структур. Что это такое? И что такое псевдоним?

О: Имя структуры — это название, которое следует за ключевым словом `struct`. Если вы напишете `struct fred { ... }`, то `fred` будет именем. И при создании переменных вы будете писать `struct fred x`.

В: А псевдоним?

О: Иногда хочется избежать использования ключевого слова `struct` при объявлении переменных, поэтому `typedef` позволяет создавать псевдонимы, состоящие из одного слова. В `typedef struct fred { ... } james` псевдонимом является `james`.

В: Так что такое анонимная структура?

О: Структура, у которой нет имени. У структуры `typedef struct { ... } james` есть псевдоним `james`, но нет имени. В большинстве случаев, если у вас есть псевдоним, имя вам не понадобится.

Как обновить структуру

На самом деле, структура — это всего лишь набор переменных, которые собраны вместе и ведут себя, словно единый фрагмент данных. Вы уже видели, как создаются объекты из структур и как осуществляется доступ к их значениям посредством оператора «точка». А как изменить значение уже существующей структуры? Вы можете изменять поля так же, как и любые другие переменные:

Здесь создается структура. →

Здесь устанавливается значение для поля teeth →

```
fish snappy = {"Зубастик", "пиранья", 69, 4};
printf("Привет,  %s\n", snappy.name);
snappy.teeth = 68;
```

← Здесь считывается значение поля name.

← Ого! Похоже, Зубастик врезался во что-то твердое.

Взглянув на данный фрагмент кода, вы ведь сможете понять, что он делает?

```
#include <stdio.h>

typedef struct {
  const char *name;
  const char *species;
  int age;
} turtle;

void happy_birthday(turtle t)
{
  t.age = t.age + 1;
  printf("С днем рождения, %s! Теперь тебе %i лет!\n",
    t.name, t.age);
}

int main()
{
  turtle myrtle = {"Тортилла", "Кожистая черепаха", 99};
  happy_birthday(myrtle);
  printf("%s прожила %i лет\n", myrtle.name, myrtle.age);
  return 0;
}
```

Но с этим кодом что-то не так...

Тест-драйв

Вот что произойдет, когда вы скомпилируете и запустите код:

Что за...

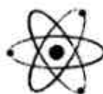
Имелось ввиду «Что за напасть».

```
File Edit Window Help ILikeTurtles
> gcc turtle.c -o turtle && ./turtle
С днем рождения Тортилла! Теперь тебе 100 лет!
Тортилла прожила 99 лет
>
```

Случилось что-то странное.

Код создает новую структуру и передает ее в функцию, которая *должна* увеличить значение одного из полей на 1. *Именно это код и сделал*... по крайней мере на какое-то время.

Поле `age` было обновлено внутри функции `happy_birthday()` — мы знаем, что так оно и было, потому что функция `printf()` вывела новое увеличенное значение. Но когда управление программой вернулось в функцию `main()`, поле `age` восстановило свое предыдущее значение.



Сила мозга

С кодом происходит что-то странное. Но у вас уже достаточно информации, чтобы сказать, ***что*** же именно произошло. Можете определить, в чем дело?

Код клонирует черепаху

Давайте внимательно присмотримся к коду, в котором вызывается функция happy_birthday():

```
void happy_birthday(turtle t)
{
  ...
}
...
happy_birthday(myrtle);
```

Это черепаха, которую мы передаем в функцию.

Структура myrtle будет скопирована в этот параметр.

Когда вы присваиваете структуру, ее значения копируются в новую структуру.

В языке Си параметры передаются в функцию **по значению**. Иначе говоря, значения, переданные функции при вызове, *присваиваются* ее параметрам. Таким образом, в нашем коде происходит что-то похожее на это:

```
turtle t = myrtle;
```

Однако *вспомните*, что присваивание структур в Си приводит к копированию значений. Когда вы вызовете функцию, параметр t будет содержать *копию* структуры myrtle. Это аналогично тому, как если бы функция ***клонировала исходную черепаху***. Таким образом, код внутри функции действительно обновляет возраст черепахи, **но не той.**

Что происходит, когда функция завершает работу? Параметр t исчезает, а остальной код внутри функции main() использует структуру myrtle, значение которой никогда не менялось. Это всегда был совершенно отдельный участок кода.

Так что же делать, если нужно обновить структуру внутри другой функции?

Вам нужен указатель на структуру

Когда мы используем функцию `scanf()`, мы можем передать ей только **указатель**, но не саму переменную:

```
scanf("%f", &length_of_run);
```

Зачем так было сделано? Если функция `scanf()` будет знать, по какому адресу в памяти находится переменная, она сможет обновить данные по этому адресу, то есть обновить переменную.

То же самое касается и структур. Если вы хотите, чтобы функция обновила переменную структуры, не стоит передавать ей в качестве параметра саму структуру, поскольку в результате в функцию будет отправлена *копия* данных. Вместо этого нужно передать адрес структуры:

```
void happy_birthday(turtle *t)
{
   ...
}
...
happy_birthday(&myrtle);
```

Это значит: «Кто-то собирается дать мне указатель на структуру».

Не забывайте: адрес -- указатель

Это значит, что мы передадим в функцию адрес переменной myrtle.

Наточите свой карандаш

Посмотрим, догадаетесь ли вы, какие *выражения* необходимо подставить на место пропусков в этой новой версии функции `happy_birthday()`.

Будьте осторожны. Не забывайте, что `t` теперь является **переменной-указателем**.

```
void happy_birthday(turtle *t)
{
  .............age = .............age + 1;
  printf("С днем рождения, %s! Теперь тебе %i лет!\n",
         .............name, .............age);
}
```

Наточите свой карандаш
Решение

Вам нужно было догадаться, какие *выражения* необходимо подставить на место пропусков в новой версии функции happy_birthday().

```
void happy_birthday(turtle *t)
{
  ....(*t).....age = ....(*t).....age + 1;
  printf("С днем рождения, %s! Теперь тебе %i лет!\n",
        ....(*t).....name, ....(*t).....age);
}
```

Мы должны поставить * перед именем переменной, потому что нам нужно значение, на которое она указывает.

Круглые скобки действительно важны. Без них код не будет работать.

(*t).age или *t.age

Так для чего же необходимо помещать `*t` в круглые скобки? Дело в том, что выражения `(*t).age` и `*t.age` имеют совершенно разный смысл.

Таким образом, выражение **`*t.age`** — то же самое, что **`*(t.age)`**. Задумайтесь на минутку. Это выражение означает: «Содержимое ячейки памяти по адресу, хранящемуся в `t.age`». Но `t.age` — это не адрес в памяти.

Так что будьте осторожны со скобками, когда используете структуры, — это действительно важно.

Тест-драйв

Давайте проверим, избавились ли мы от ошибки:

```
File Edit Window Help ILikeTurtles
> gcc happy_birthday_turtle_works.c -o happy_birthday_turtle_works
С днем рождения, Тортилла! Теперь тебе 100 лет!
Тортилла прожила 100 лет
>
```

Отлично. Теперь функция работает.
Передавая указатель на структуру, мы позволяем функции обновлять *исходные данные*, вместо того чтобы создавать их локальную копию.

t->age означает *(*t).age*

Да, существует более удобочитаемый способ обозначения указателей.
Так как при работе с указателями приходится внимательно следить за правильным использованием круглых скобок, создатели языка Си придумали более простой и легкий для восприятия синтаксис. Эти два выражения обозначают одно и то же:

```
(*t).age
t->age
```

Это одно и то же.

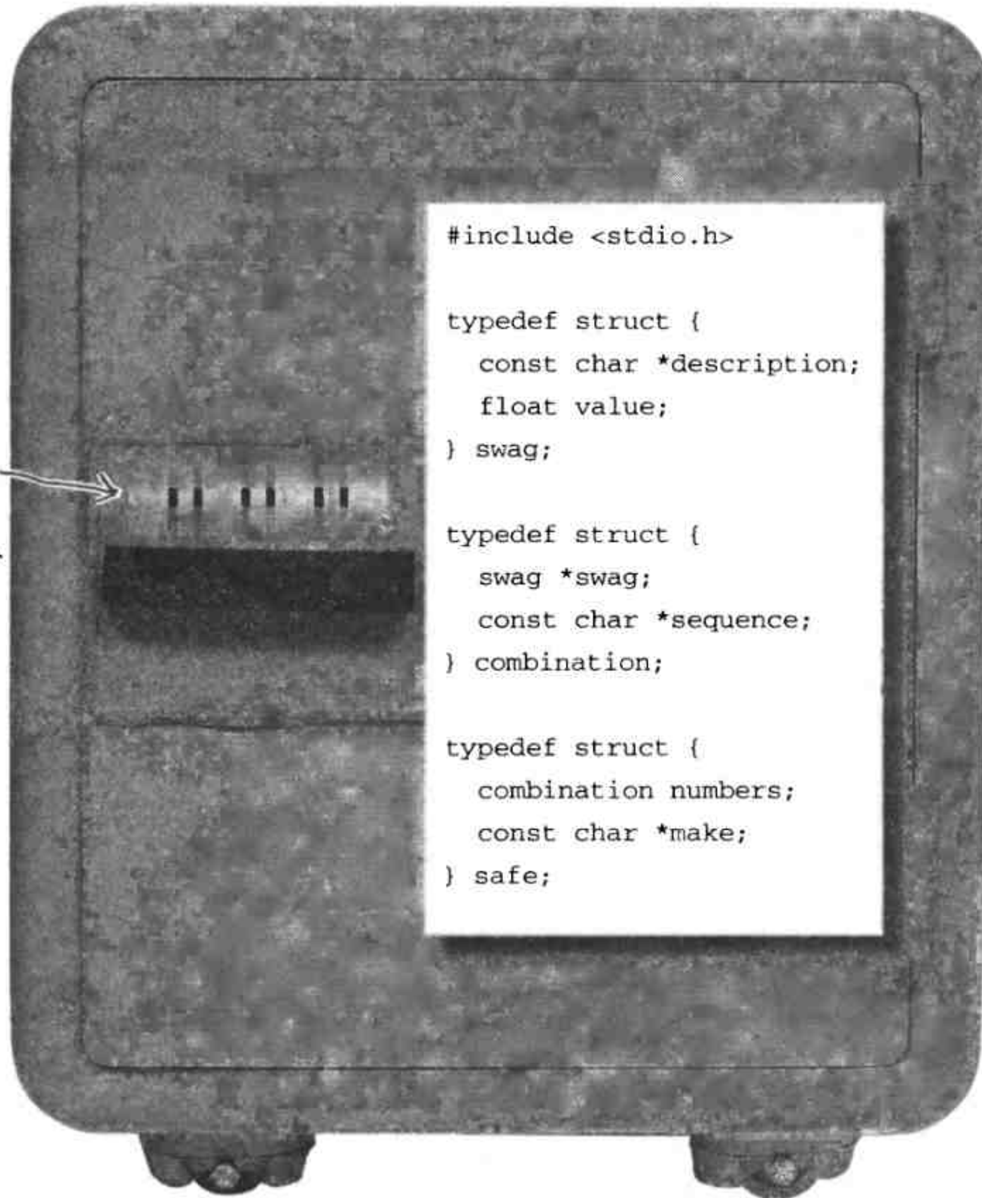
Итак, `t->age` означает: «Поле `age` в структуре, на которую указывает `t`». Следовательно, мы можем записать функцию таким образом:

```
void happy_birthday(turtle *a)
{
  a->age = a->age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
         a->name, a->age);
}
```

Взломщик сейфов

Тшш... В банковском хранилище поздняя ночь. Вы можете подобрать правильный ключ, чтобы взломать сейф? Изучите эти фрагменты кода, затем попробуйте найти подходящую комбинацию, которая позволит добраться до золота.

Вам нужно подобрать эту комбинацию.

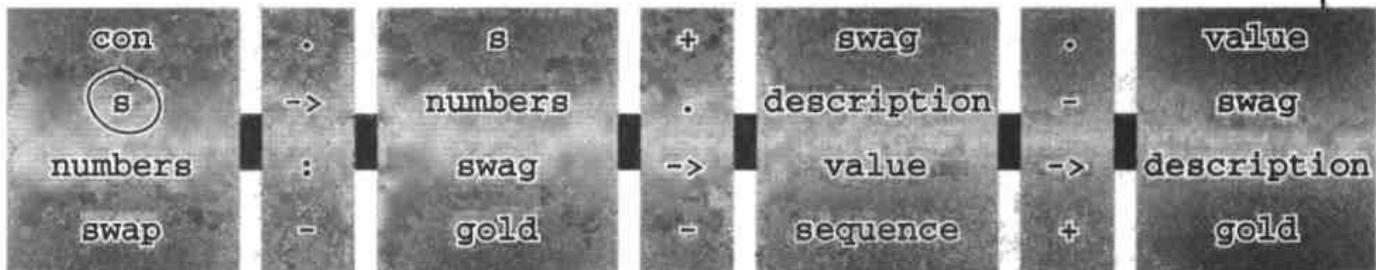


```
#include <stdio.h>

typedef struct {
  const char *description;
  float value;
} swag;

typedef struct {
  swag *swag;
  const char *sequence;
} combination;

typedef struct {
  combination numbers;
  const char *make;
} safe;
```

Банковский сейф был создан следующим образом:

```
swag gold = {"ЗОЛОТО!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

Какая комбинация приведет вас к строке «ЗОЛОТО!»? Чтобы собрать выражение, выберите по одному символу или слову из каждой колонки.



не бывает Глупых Вопросов

В: Почему значения при передаче через параметры копируются?

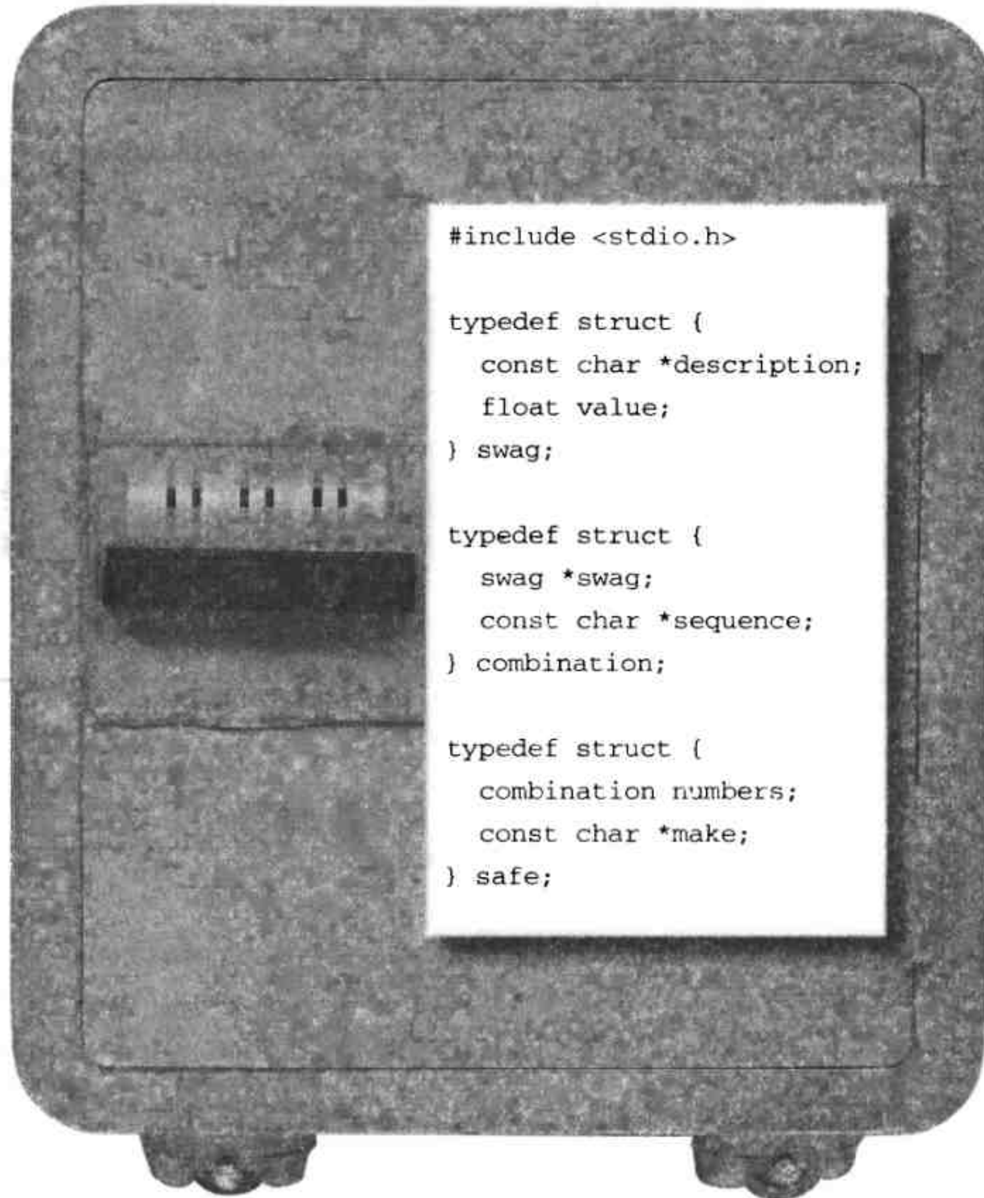
О: Компьютер присвоит значение каждому параметру так же, как если бы вы написали `parameter=value`. А присваивание всегда приводит к копированию значений.

В: Почему `*t.age` воспринимается не так, как `(*t).age`?

О: Потому что компьютер в первую очередь рассматривает оператор «точка», и только затем `*`.

Взломщик сейфов Решение

Тшш... В банковском хранилище поздняя ночь. Вам следовало подобрать правильный ключ, чтобы взломать сейф. Для этого вы должны были изучить фрагменты кода и найти подходящую комбинацию, которая позволила бы добраться до золота.

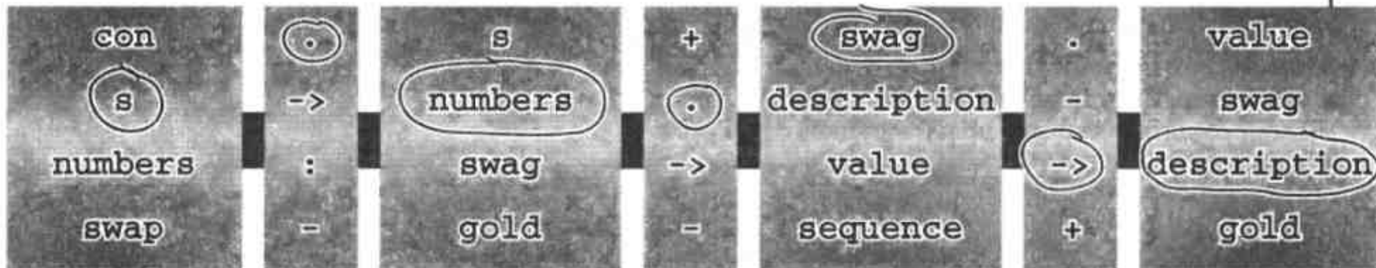


```
#include <stdio.h>

typedef struct {
  const char *description;
  float value;
} swag;

typedef struct {
  swag *swag;
  const char *sequence;
} combination;

typedef struct {
  combination numbers;
  const char *make;
} safe;
```

Банковский сейф был создан следующим образом:

```
swag gold = {"ЗОЛОТО!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

Какая комбинация приведет вас к строке «ЗОЛОТО!»? Чтобы собрать выражение, вам нужно было выбрать по одному символу или слову из каждой колонки.



Итак, вы можете обнаружить золото в сейфе, написав:

```
printf("Содержимое - %s\n", s.numbers.swag->description);
```

КЛЮЧЕВЫЕ МОМЕНТЫ

- При вызове функции значения, передаваемые ей через параметры, ***копируются***.
- Вы можете создавать указатели на структуры так же, как и на любые другие типы данных.
- `pointer->field` то же самое, что `(*pointer).field`.
- Запись `->` избавляет от использования круглых скобок и делает код более легким для восприятия.

Иногда для одного и того же типа сущностей нужны разные типы данных

Структуры позволяют вам моделировать сложные вещи из реального мира. Но иногда встречаются данные, у которых нет единого типа.

Итак, вы хотите записать какую-то *величину*, например **количество, вес** или **объем**. Как это сделать? Что ж, вы *можете* создать несколько полей в рамках структуры. Например:

```
typedef struct {
  ...
  short count;
  float weight;
  float volume;
  ...
} fruit;
```

Но есть несколько причин, почему так поступать не стоит.

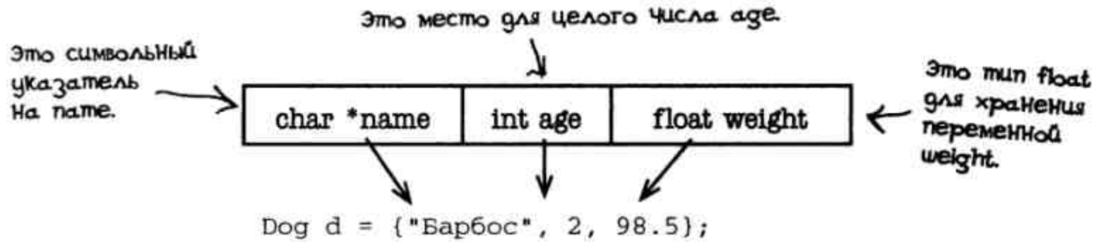
- **Это будет занимать больше места в памяти.**
- **Кто-то может задать более одного значения.**
- **Не существует объекта под названием «величина».**

Было бы *действительно полезно* указать в виде типа данных нечто вроде величины и затем решать, что именно записывать для каждого конкретного фрагмента — количество, вес или объем.

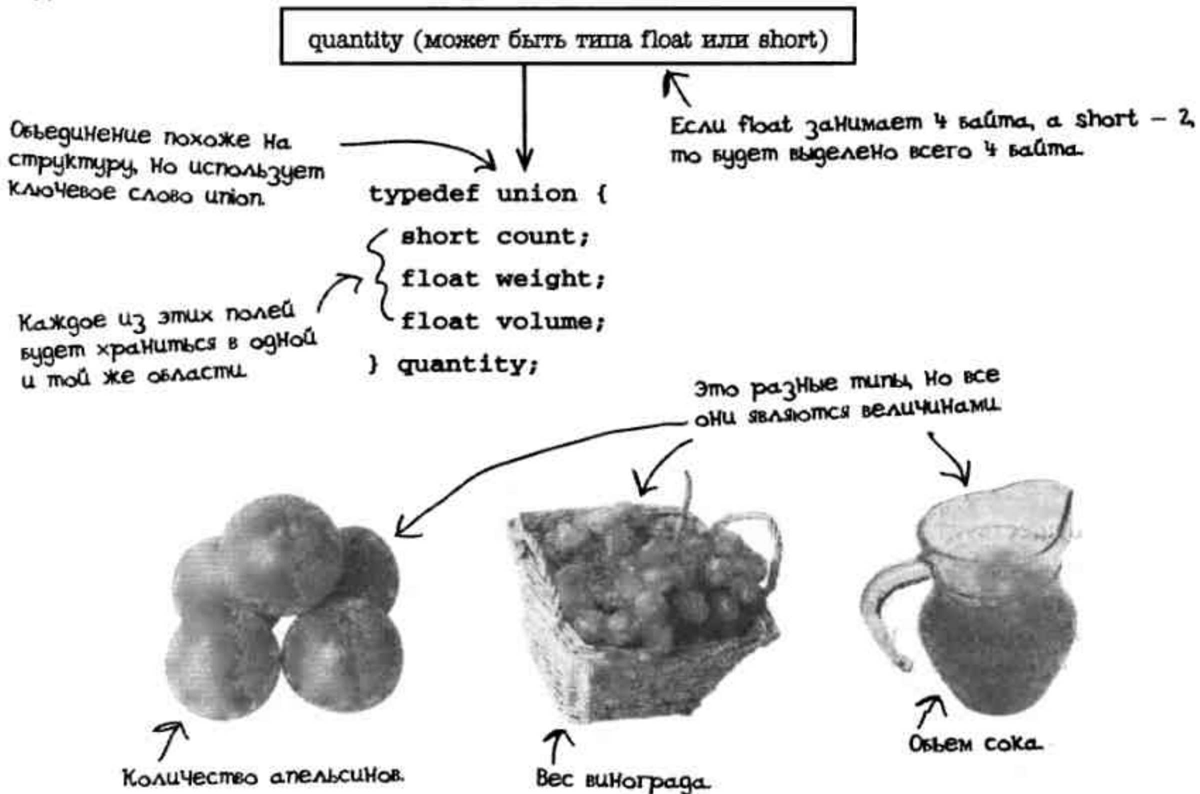
Вы можете делать это в языке Си, используя объединения.

Объединения позволяют повторно использовать место в памяти

Каждый раз, когда вы создаете экземпляр структуры, компьютер размещает в памяти ее поля одно за другим.



С **объединениями** все не так. Они используют место только для одного из полей, содержащихся в их определении. Таким образом, если у вас есть объединение `quantity` с полями `count`, `weight` и `volume`, компьютер выделит место для самого большого из них, а вы уже будете решать, какое значение там хранить. Какое бы поле вы ни выбрали — `count`, `weight` или `volume`, — данные будут размещены в одной и той же области памяти:



Как пользоваться объединениями

Есть несколько способов присвоить значение переменной, объявленной как объединение.

Стиль C89 для первого поля

Если объединение будет хранить значение для **первого поля**, вы можете использовать запись, принятую в стандарте C89. Чтобы присвоить значение первому полю объединения, просто заключите его в фигурные скобки:

```
quantity q = {4};
```

← Это значит, что в качестве величины выступает количество, которое равняется 4.

Назначаемые инициализаторы для других значений

Назначаемый инициализатор присваивает значения полям по их **именам** — вот так:

```
quantity q = {.weight=1.5};
```

← Здесь объединению присваивается значение веса — число с плавающей точкой.

Присваивание значения с помощью «точка»

Третий способ инициализации объединений состоит из двух строчек: в первой создается переменная, а во второй ее полю присваивается значение.

```
quantity q;
q.volume = 3.7;
```

Запомните: каким бы способом вы ни присвоили значение объединению, **сохранен будет только один фрагмент данных**. Объединения всего лишь позволяют создавать переменные, которые поддерживают ***несколько типов данных***.

не бывает Глупых Вопросов

В: Почему размер объединения всегда равен размеру самого большого поля?

О: Компьютер должен быть уверен, что объединение всегда будет иметь один и тот же размер. Единственный способ это сделать — убедиться, что объединение будет достаточно вместительным, чтобы содержать любое из полей.

В: Почему стиль C89 предусматривает присвоение значения только первому полю? Почему бы не пройтись по полям и не присвоить значение тому из них, у которого первым окажется соответствующий тип?

О: Чтобы избежать неопределенности. ЕСЛИ у вас есть два поля — `float` и `double`, какому из них компьютер должен присвоить значение {2.1}? Благодаря тому что значение всегда присваивается первому полю, вы будете точно знать, где находятся данные.

Деликатный путеводитель по стандартам

Назначаемые инициализаторы являются частью стандарта C99 и позволяют присваивать значения полям структур и объединений по их именам. Они поддерживаются большинством компиляторов, но будьте осторожны, если вы используете какую-то *вариацию* языка Си. Например, *Objective C* поддерживает назначаемые инициализаторы, а *Си++* — **нет**.

Кажется, эти назначаемые инициализаторы могли бы пригодиться и для структур. Интересно, можно ли их так использовать?..

Да, назначаемые инициализаторы можно использовать также и для инициализации полей структур.

Они могут быть очень полезны в случае, если у вас есть структура с большим количеством полей, а вы для начала хотите инициализировать только некоторые из них. Это хороший способ упростить код:

```
typedef struct {
  const char *color;
  int gears;
  int height;
} bike;
bike b = {.height=17, .gears=21};
```

Здесь инициализируются поля gears и height, а поле color остается нетронутым.

Объединения часто используются совместно со структурами

Создавая объединения, вы создаете *новые типы данных.* Следовательно, вы можете использовать их там же, где и другие типы данных, такие как, например, `int` или `struct`. Иначе говоря, вы можете использовать их в структурах:

```
typedef struct {
  const char *name;
  const char *country;
  quantity amount;
} fruit_order;
```

Вы, как и прежде, можете получать доступ к значениям структур внутри объединений (и наоборот), используя точку или запись вида `->`:

Здесь используется .amount, потому что это название переменной, которая является объединением и находится внутри структуры.

Здесь мы используем двойной назначаемый идентификатор: .amount для структуры и .weight для .amount.t.

```
fruit_order apples = {"яблок", "Англия", .amount.weight=4.2};
printf("Этот заказ содержит %2.2f кг %s\n", apples.amount.weight, apples.name);
```

Здесь будет напечатано: «Этот заказ содержит 2 кг яблок».

Взболтать, но не смешивать

Сегодня в баре Head First ночь «Маргариты». Но, похоже, ребята слишком увлеклись дегустацией и перемешали рецепты приготовления коктейлей. Попробуйте найти фрагменты кода, соответствующие разным видам коктейля «Маргарита».

Вот основные ингредиенты:

```
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

Вот разные виды коктейлей:

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```
margarita m = {2.0, 1.0, {1}};
```

```
margarita m = {2.0, 1.0, {2}};
```

И наконец, вот несколько разных рецептов для коктейлей и напитков, которые они приготовили. Какие из видов «Маргариты» должны быть добавлены в эти фрагменты кода, чтобы в итоге получились правильные рецепты?

....

```
printf("%2.1f порции текилы\n%2.1f порции куантро\n%2.1f
    порции сока\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 порции текилы
1.0 порци куантро
2.0 порции сока
```

....

```
printf("%2.1f порции текилы \n%2.1f порции куантро\n%2.1f
    порции сока\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 порции текилы
1.0 порции куантро
0.5 порции сока
```

....

```
printf("%2.1f порции текилы \n%2.1f порции куантро\n%i кусочек
    лайма\n", m.tequila, m.cointreau, m.citrus.lime_pieces);
```

```
2.0 порции текилы
1.0 порции куантро
1 кусочек лайма
```

Поработайте Компилятором

Один из этих фрагментов кода компилируется, а другой — нет. Вы должны сыграть роль компилятора и выяснить, какой фрагмент будет работать, какой нет и почему.

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

Взболтать, но не смешивать

Сегодня в баре Head First ночь «Маргариты». Но, похоже, ребята слишком увлеклись дегустацией и перемешали рецепты приготовления коктейлей. Вам нужно было найти фрагменты кода, соответствующие разным видам коктейля «Маргарита».

Вот основные ингредиенты:

```
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

Вот разные виды коктейлей:

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

Ни одна из этих строчек не была использована

```
margarita m = {2.0, 1.0, {1}};
```

И наконец, вот несколько разных рецептов для коктейлей и напитков, которые они приготовили. Какие из видов «Маргариты» должны были быть добавлены в эти фрагменты кода, чтобы в итоге получились правильные рецепты?

```
margarita m = {2.0, 1.0, {2}};

printf("%2.1f порции текилы\n%2.1f порции куантро\n%2.1f
    порции сока\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 порции текилы
1.0 порции куантро
2.0 порции сока
```

```
margarita m = {2.0, 1.0, {0.5}};

printf("%2.1f порции текилы \n%2.1f порции куантро\n%2.1f
    порции сока\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 порции текилы
1.0 порции куантро
0.5 порции сока
```

```
margarita m = {2.0, 1.0, {.lime_pieces=1}};

printf("%2.1f порции текилы \n%2.1f порции куантро\n%i кусочек
    лайма\n", m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 порции текилы
1.0 порции куантро
1 кусочек лайма
```

Поработайте компилятором. Решение

Один из этих фрагментов кода компилируется, а другой — нет. Вы должны были сыграть роль компилятора и выяснить, какой фрагмент будет работать, какой нет и почему.

```
margarita m = {2.0, 1.0, {0.5}};
```

Этот фрагмент скомпилируется без проблем. Вообще-то это один из напитков, о которых мы говорили выше!

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

Этот фрагмент не скомпилируется, потому что компилятор распознает запись {2.0, 1.0, {0.5}} только в том случае, если она находится в одной строчке с объявлением структуры. Если запись будет идти отдельной строкой, компилятор воспримет ее как массив.

Минуточку... Вы присваиваете все эти значения с разными типами, а хранятся они в **одном и том же участке памяти**... Как я потом узнаю, что сохранила туда float? Что мне помешает прочитать это значение как short или что-то еще? **Эй?..**

Это действительно хорошее замечание. Вы можете хранить внутри объединения множество разных значений, но у вас *нет никакой возможности узнать*, данные какого типа там находятся, если вы их туда уже поместили.

Компилятор не сможет следить за полями, которые принадлежат объединению, поэтому нам ничего не стоит сделать запись в одном поле, а чтение произвести из другого. Разве это проблема? Иногда это может быть **БОЛЬШОЙ ПРОБЛЕМОЙ...**

```
#include <stdio.h>
typedef union {
  float weight;
  int count;
} cupcake;

int main()
{
  cupcake order = {2};
  printf("Количество кексов: %i\n", order.count);
  return 0;
}
```

Программист по ошибке инициализировал вес, а не количество.

Инициализировался вес, а считывается количество.

Вот что сделала программа.

```
> gcc badunion.c -o badunion && ./badunion
Количество кексов: 1073741824
```

Много кексов получилось...

Нам нужно каким-то образом отслеживать значения, хранящиеся в объединении. Для этого некоторые программисты на Си используют один прием — **перечисление**.

Переменная типа enum хранит обозначения

Иногда нужно хранить не число или текст, а список обозначений. Если вы хотите записать день недели, вам достаточно сохранить ПОНЕДЕЛЬНИК, ВТОРНИК, СРЕДА и т. д. Для этого не нужно сохранять текст, ведь количество возможных значений для выбора равно 7.

Вот для чего были придуманы перечисления.

Перечисление позволяет создавать список обозначений. Например:

Возможные цвета в нашем перечислении.

Значения разделяются запятыми.

```
enum colors {RED, GREEN, PUCE};
```

С помощью typedef мы можем присвоить нашему типу подходящее имя.

Любой объявленной переменной с типом **enum colors** может быть присвоено одно из ключевых слов, находящихся в списке. Таким образом, вы можете объявить переменную типа enum colors следующим образом:

```
enum colors favorite = PUCE;
```

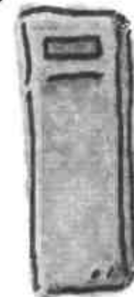
Будьте осторожны!

В структурах и объединениях элементы разделяются точкой с запятой (;), но в перечислениях используются обычные запятые.

В действительности каждому обозначению из списка компьютер присвоит число, и именно числа будут храниться внутри перечисления. Знать эти числа вам не нужно, в своем коде вы можете ссылаться просто на обозначения. Благодаря этому код станет более простым для чтения, а вам не нужно будет использовать значения вроде REB или PUSE:

Компьютер догадается, что это недопустимое значение, и прервет компиляцию.

```
enum colors favorite = PUSE;
```



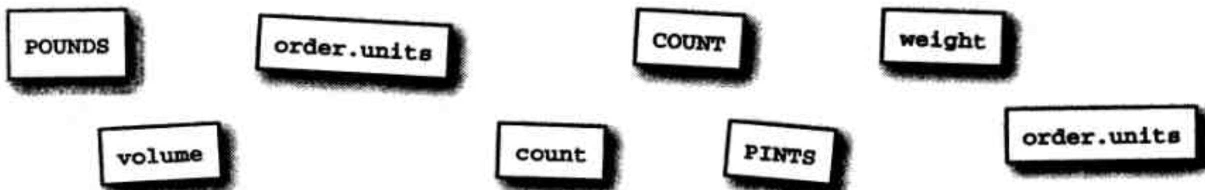
Вот так работают перечисления. Но как они помогут нам в отслеживании объединений? Давайте рассмотрим пример...

Магнитики с кодом

Созданный вами на основе перечисления новый тип данных можно помещать внутрь структур и объединений. В этой программе перечисление используется для определения того, какая именно величина была сохранена. Сможете ли вы догадаться, куда делись пропавшие фрагменты кода?

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("Этот заказ содержит ");

  if (............................ == PINTS)

    printf("%2.2f пинт %s\n", order.amount. ............................ , order.name);
```

```
  else if (.......................... == ..........................)
    printf("%2.2f фунтов %s\n", order.amount.weight, order.name);
  else

    printf("%i %s\n", order.amount. .......................... , order.name);
}

int main()
{

  fruit_order apples = {"яблок", "Англия", .amount.count=144,  .......................... };

  fruit_order strawberries = {"клубники", "Испания", .amount...........=17.6, POUNDS};

  fruit_order oj = {"апельсинового сока", "США", .amount.volume=10.5,....................};
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```



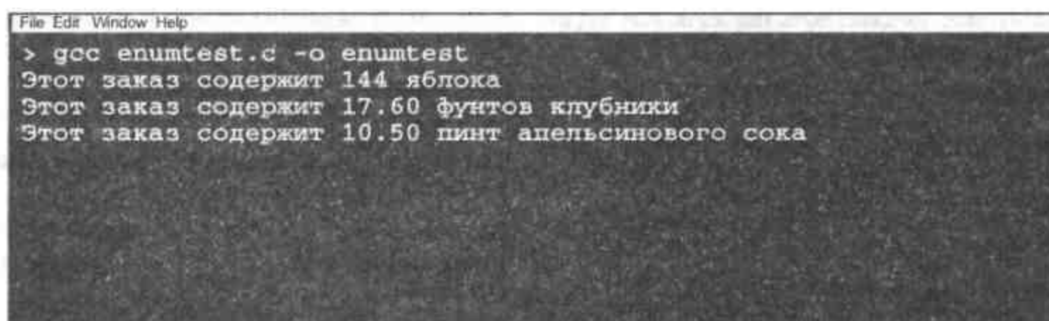
Магнитики с кодом Решение

Созданный вами на основе перечисления новый тип данных можно помещать внутрь структур и объединений. В этой программе перечисление используется для определения того, какая именно величина была сохранена. Вы смогли догадаться, куда делись пропавшие фрагменты кода?

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("Этот заказ содержит ");

  if (order.units == PINTS)

    printf("%2.2f пинт %s\n", order.amount.volume, order.name);
```

```
  else if (..order.units.. == .......POUNDS....)
    printf("%2.2f фунтов %s\n", order.amount.weight, order.name);
  else
    printf("%i %s\n", order.amount.count.........., order.name);
}

int main()
{
  fruit_order apples = {"яблок", "Англия", .amount.count=144, .....COUNT............};
  fruit_order strawberries = {"клубники", "Испания", .amount.weight=17.6, POUNDS};
  fruit_order oj = {"апельсинового сока", "США", .amount.volume=10.5,..PINTS....};
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```

Запустив программу, вы получите следующее:



```
> gcc enumtest.c -o enumtest
Этот заказ содержит 144 яблока
Этот заказ содержит 17.60 фунтов клубники
Этот заказ содержит 10.50 пинт апельсинового сока
```

Объединение: ...В общем, я сказал коду: «Эй, смотри. Мне все равно, давал ты мне `float` или нет. Ты попросил `int`. Вот и держи `int`».

Структура: Старина, это было совершенно незаслуженное оскорбление.

Объединение: Вот и я говорю, что незаслуженное.

Структура: Все же знают, что у тебя есть только одно место для хранения.

Объединение: Вот именно. Все едино. Я в чем-то похож на дзен-буддиста...

Перечисление: Что случилось, приятель?

Структура: Не вмешивайся, Перечисление. Я считаю, что парень совсем обнаглел.

Объединение: Я и говорю, если бы он хотя бы оставил какую-то запись. Обвинил, что я сохранил это как `int`. Ему нужно было использовать перечисление или что-то еще.

Перечисление: Ты что-то хотел?

Структура: Помолчи, Перечисление.

Объединение: Если он хотел сохранить за раз несколько вещей, он ведь должен был позвать тебя, правильно?

Структура: Во всем должен быть порядок. Как люди не могут этого понять.

Перечисление: Чистота и порядок?

Структура: Разделение и последовательность — вот что нужно. Я храню рядом друг с другом несколько вещей. Причем одновременно, старина.

Объединение: Это именно то, о чем я и говорю.

Структура: Все О Д Н О В Р Е М Е Н Н О.

Перечисление: (Пауза) Так, а в чем проблема?

Объединение: Перечисление, я тебя прошу... Я говорю, что люди просто должны сделать выбор. Хотите хранить несколько вещей — используйте структуру. Но записывать одну вещь с разными возможными типами? Это ни в какие ворота.

Структура: Сейчас я с ним поговорю.

Объединение: Эй, подожди...

Перечисление: Кому это он звонит, приятель?

Структура/Объединение: Заткнись, Перечисление.

Объединение: Слушай, давай не будем усугублять.

Структура: Алло? Могу я поговорить с сервисом Bluetooth?

Объединение: Эй, давай просто все обдумаем.

Структура: Что значит «он Вам перезвонит»?

Объединение: Я просто... Мне кажется, это не очень удачная идея.

Структура: Нет, приятель, давай я оставлю ему сообщение.

Объединение: Пожалуйста, просто повесь трубку.

Перечисление: Кто на связи, старина?

Структура: Тише, Перечисление. Не видишь, я разговариваю по телефону? Слушай, просто передай ему, что, если он хочет сохранить `float` и `int`, пусть приходит ко мне. Или я сам к нему приду. Понятно? Алло? Алло?

Объединение: Полегче, приятель. Просто попробуй успокоиться.

Структура: «Ожидайте»? Они перевели меня в ^*&^ый режим ожидания!

Объединение: Они что? А ну дай мне трубку... О... У них вместо гудков играет The Eagles! Ненавижу The Eagles...

Перечисление: Так вот почему ты такой толстый — вмещаешь в себя несколько полей одновременно?

Структура: Ну все, дружище, ты нарвался!

Иногда нужен контроль на уровне отдельных битов

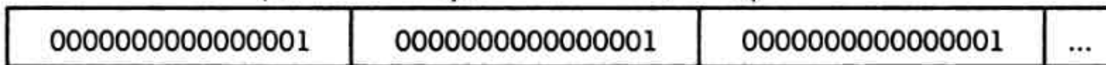
Представьте, что вам понадобилась структура, которая содержит много значений вида да/нет. Вы *могли* бы создать ее с помощью набора значений типа short или int:

```
typedef struct {
  short low_pass_vcf;
  short filter_coupler;
  short reverb;
  short sequential;
  ...
} synth;
```

Каждое из этих полей будет содержать 1 для true или 0 для false.

После этих полей следует еще множество других.

В каждом поле будет использоваться много битов.



Это будет работать. Но есть одна проблема. Поля типа short будут занимать значительно больше места, чем *один бит* (а именно столько требуется для значений **true/false**). Это неэкономно. Было бы намного лучше, если бы наша структура могла хранить значения в виде последовательности одиночных битов.

Поэтому и были придуманы ***битовые поля***.

Двоичная система счисления для ботанов

При работе с двоичными значениями было бы полезно каким-то образом указывать ноли и единички в виде литерала. Например:

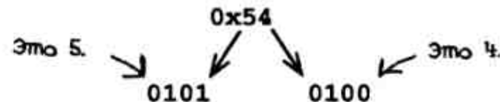
```
int x = 01010100;
```

К сожалению, язык Си не поддерживает **двоичные литералы**, однако в нем *есть* поддержка **шестнадцатеричной системы**. Любое число, начинающееся с 0x, Си воспринимает как **шестнадцатеричное**:

```
int x = 0x54;
```

Это не десятичное 54.

Но как преобразовать шестнадцатеричное значение в двоичное и наоборот? И проще ли это, чем переход между двоичными и **десятичными** числами? Хорошая новость — вы можете конвертировать шестнадцатеричные числа в двоичные, преобразуя **каждую цифру отдельно**:



Каждая шестнадцатеричная цифра соответствует четырехзначному двоичному числу. Как только вы выучите принцип, по которому цифры от 0 до 15 переводятся в двоичный вид, то сможете делать преобразования между шестнадцатеричными и двоичными значениями за считаные секунды.

Битовые поля хранят определенное количество битов

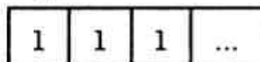
Битовые поля позволяют указать, *сколько битов* будет храниться в отдельном поле. Например, мы могли бы переписать нашу структуру таким образом:

```
typedef struct {
  unsigned int low_pass_vcf:1;
  unsigned int filter_coupler:1;
  unsigned int reverb:1;
  unsigned int sequential:1;
  ...
} synth;
```

Каждое поле должно иметь тип unsigned int.

Это значит, что для каждого поля будет использоваться по одному биту.

С помощью битовых полей мы можем сделать так, чтобы каждая переменная принимала только 1 бит.



Будьте осторожны!

Если битовые поля собраны внутри структуры, они могут сэкономить место в памяти.

Но если компилятор найдет всего одно битовое поле, он выделит для него целое слово. Вот почему битовые поля обычно находятся рядом друг с другом.

Если у вас есть последовательность битовых полей, компьютер может **упаковать их вместе**, чтобы сэкономить память. Таким образом, восемь однобитных полей могут храниться в одном единственном байте.

Давайте посмотрим, как вам удастся применить битовые поля на практике.

Сколько битов мне нужно?

Битовые поля можно использовать не только для хранения последовательности значений true/false, они могут быть полезны и для других значений, имеющих небольшой диапазон, например месяцы в году. Сохраняя в структуре номер месяца, вы знаете, что это будет значение, скажем, от 0 до 11. Вы можете уместить эти числа в 4 бита. Почему? Потому что 4 бита позволяют хранить значения от 0 до 15, а 3 бита — только от 0 до 7.

```
...
unsigned int month_no:4;
...
```

Упражнение

Вернемся к аквариуму Head First. Наши ребята решили узнать, насколько посетители довольны их услугами. Давайте проверим, сможете ли вы применить битовые поля при создании соответствующей структуры.

Анкета для посетителей аквариума

Это Ваш первый визит?	
Придете ли Вы снова?	
Количество пальцев, потерянных Вами в резервуаре с пираньями	
Потеряли ли Вы своего ребенка в секции с акулами?	
Сколько дней в неделю Вы бы нас посещали, если бы у Вас была такая возможность?	

```
typedef struct {
  unsigned int first_visit: ..........;
  unsigned int come_again: ..........;
  unsigned int fingers_lost: ..........;
  unsigned int shark_attack: ..........;
  unsigned int days_a_week: ..........;
} survey;
```

Вам нужно решить, сколько битов использовать.

Упражнение Решение

Вернемся к аквариуму Head First. Наши ребята решили узнать, насколько посетители довольны их услугами. Вам нужно было применить битовые поля при создании соответствующей структуры.

Анкета для посетителей аквариума

Это Ваш первый визит?	
Придете ли Вы снова?	
Количество пальцев, потерянных Вами в резервуаре с пираньями	
Потеряли ли Вы своего ребенка в секции с акулами?	
Сколько дней в неделю Вы бы нас посещали, если бы у Вас была такая возможность?	

```
typedef struct {
  unsigned int first_visit: 1;
  unsigned int come_again: 1;
  unsigned int fingers_lost: 4;
  unsigned int shark_attack: 1;
  unsigned int days_a_week: 3;
} survey;
```

1 бит может хранить два значения: true/false.

4 бита нужны для хранения чисел не больше 10.

3 бита могут вместить число не больше 7.

не бывает Глупых Вопросов

В: Почему Си не поддерживает двоичные литералы?

О: Потому что они занимают много места. Как правило, использование шестнадцатеричных значений более эффективно.

В: Почему для хранения чисел до 10 нужно 4 бита?

О: В 4 битах умещаются значения от 0 до двоичного 1111, что равняется 15. Но 3 бита могут вместить значения не больше двоичного 111, что равняется 7.

В: Что произойдет, если попробовать поместить число в слишком большое битовое поле?

О: Компьютер переместит только те биты, которые ему нужны.

В: А что если я попробую поместить значение 9 внутрь 3-битного поля?

О: Компьютер сохранит 1 (001 в двоичном коде), так как 9 == 1001.

В: Битовые поля используются только для экономии места в памяти?

О: Нет. Они могут пригодиться, если вам нужно прочитать низкоуровневую двоичную информацию.

В: Например?

О: Если вы считываете или записываете какой-то нестандартный бинарный файл.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Объединение позволяет хранить разные типы данных в одном и том же месте в памяти.
- Назначаемые инициализаторы присваивают значения полям по их именам.
- Назначаемые инициализаторы являются частью стандарта C99. Они не поддерживаются в Си++.
- Если объявить объединение со значением внутри {фигурных скобок}, оно будет сохранено с типом первого поля.
- Компилятор не помешает вам сохранить в объединении одно поле, а прочитать совершенно другое.
- Перечисления хранят обозначения.
- Битовые поля позволяют хранить значения с нестандартным количеством битов.
- Битовые поля всегда объявляются как `unsigned int`.

Ваш инструментарий языка Си

Вы изучили главу 5, пополнив свои знания информацией о структурах, объединениях и битовых полях. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

typedef позволяет создавать псевдонимы для типов данных.

Вы можете считывать поля структуры с помощью оператора «точка».

Структура сочетает в себе разные типы данных.

С помощью записи -> вы можете легко обновлять поля, используя указатель на структуру.

Вы можете инициализировать структуры так же, как вы это {делали, с, массивами}.

Назначаемые инициализаторы позволяют присваивать значения полям структур и объединений по их именам.

Объединения могут хранить разные типы данных в одном и том же месте.

Битовые поля позволяют выбирать, сколько именно бит будет сохранено внутри структуры.

Перечисления позволяют создавать набор обозначений.

6 Структуры данных и динамическая память

Наводим мосты

Иногда одной структуры недостаточно.

Для моделирования сложных требований к данным нужно **соединить несколько структур**. В этой главе вы узнаете, как с помощью **указателей** объединять нестандартные типы данных в **большие сложные структуры**. В процессе создания **связных списков** вы изучите *ключевые принципы* этого подхода, а также научитесь создавать структуры, совместимые с данными разных объемов, **динамически выделяя память в куче** и освобождая ее в момент, когда работа с ней уже закончена. И если организация всего этого процесса слишком усложнится, вам поможет `valgrind`, о котором мы также поговорим.

Нужно ли вам гибкое хранилище?

Си-самолет Кокосовых авиалиний летает между островами.

Мы уже рассмотрели различные виды сохраняемых данных в Си и научились хранить несколько фрагментов данных в одном массиве. Но иногда может понадобиться более гибкий подход.

Представьте, что вы работаете в туристической компании, которая организует авиатуры на тропические острова. Любой тур состоит из последовательности коротких перелетов с одного острова на другой. Вам нужно знать название каждого острова и время работы местного аэропорта. Как вы запишете эту информацию?

Вы можете создать структуру для каждого отдельного острова:

```
typedef struct {
  char *name;
  char *opens;
  char *closes;
} island;
```

Если тур будет включать в себя *несколько* островов, нам понадобится их список — массив:

```
island tour[4];
```

Но есть проблема. Массивы имеют фиксированную длину, это значит, что они не очень-то *гибкие*. Мы можем использовать их, если будем **точно** знать *длину* маршрута. Но что если мы изменим его? Что если посреди маршрута мы добавим еще один пункт назначения?

Чтобы хранить данные, количество которых может меняться, нам нужно что-то более растяжимое, чем массив. Нам нужен *связной список*.

Связные списки похожи на цепочки данных

Связной список — это пример **абстрактной структуры данных**. *Абстрактной* ее называют потому, что она является *общей* и ее можно использовать для хранения множества разных видов данных.

Чтобы понять принцип работы связных списков, вернемся к нашей туристической компании. Связной список хранит как сами данные, так и ссылки на *другие* узлы списка.

Наточите свой карандаш

Если вы знаете, где начинается связной список, то можете перемещаться по нему от одного фрагмента данных к другому с помощью ссылок, пока не достигнете конца списка. Возьмите карандаш и измените список таким образом, чтобы тур включал в себя путешествие на остров Черепа (между Скалистым и Туманным островами).

Мы сохраняем некоторый набор данных для каждого острова.

Это ссылка на следующий фрагмент данных.

Наточите свой карандаш
Решение

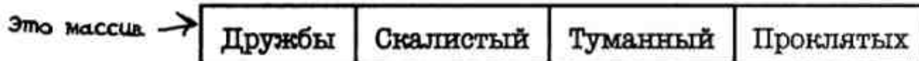
Если вы знаете, где начинается связной список, то можете перемещаться по нему от одного фрагмента данных к другому с помощью ссылок, пока не достигнете конца списка. Используя карандаш, вам нужно было изменить список таким образом, чтобы тур включал в себя путешествие на остров Черепа (между Скалистым и Туманным островами).

Связные списки поддерживают вставку данных

Сделав всего несколько изменений, можно добавить к нашему маршруту дополнительный этап. В этом заключается еще одно преимущество связных списков перед массивами — **данные вставляются очень быстро**. Если вы захотите вставить данные посреди *массива*, вам придется сдвигать на одну позицию все, что находится за ними:

Если вы захотите добавить значение после Скалистого острова, вам придется сдвинуть все остальные на одну позицию.

Это массив.



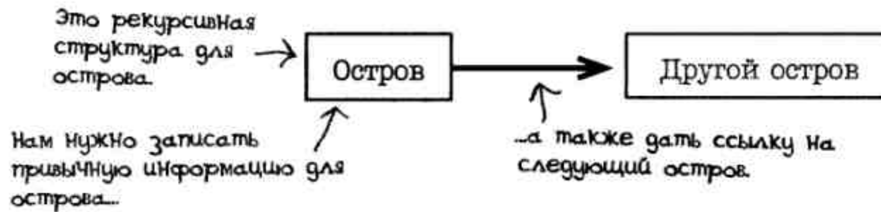
И поскольку массив имеет фиксированную длину, вы потеряете остров Проклятых.

Таким образом, связные списки позволяют хранить **переменное количество** данных и упрощают процесс **добавления новых элементов**.

Но как создать связной список на языке Си?

Создаем рекурсивную структуру

Каждый элемент в списке должен быть связан со следующим. Структура, содержащая ссылку на другую структуру того же типа, называется **рекурсивной**.



Рекурсивные структуры содержат указатели на другие структуры того же типа. Если у нас есть расписание перелетов между островами, которые мы собираемся посетить, то мы можем использовать рекурсивную структуру для каждого острова. Давайте рассмотрим, как это работает.

Мы запишем эти подробности для каждого острова.

Аэропорт на острове	
Название:	Дружба
Открывается:	9:00
Закрывается:	17:00
Следующий остров:	Скалистый

Мы также запишем для каждого из них следующий остров.

Мы должны дать имя структуре.

```
typedef struct island {
  char *name;
  char *opens;
  char *closes;
  struct island  *next;
} island;
```

Для названия и времени открытия/закрытия мы будем использовать строки.

Мы будем хранить в структуре указатель на следующий остров.

Как мы сохраним ссылку с одной структуры на другую? Воспользуемся указателем. Информация об острове будет содержать *адрес* следующего острова, на который мы собираемся. Таким образом, имея данные об одном острове, мы всегда сможем узнать следующий пункт маршрута.

Давайте напишем немного кода и начнем путешествие по островам.

Будьте осторожны!

У рекурсивной структуры должно быть имя.

При использовании команды `typedef` ***вы можете пропустить собственное имя структуры. Однако в случае с*** ***рекурсивными*** ***структурами необходимо добавить указатель на тот же тип. Синтаксис Си не позволит вам использовать псевдоним, созданный с помощью*** `typedef`, ***поэтому структуре необходимо присвоить настоящее имя. Вот почему в этом примере структура называется*** `struct island`.

Создадим острова с помощью Си...

Объявив тип данных `island`, вы можете создать первый набор островов. Например:

В этом коде будут созданы структуры для каждого из островов.

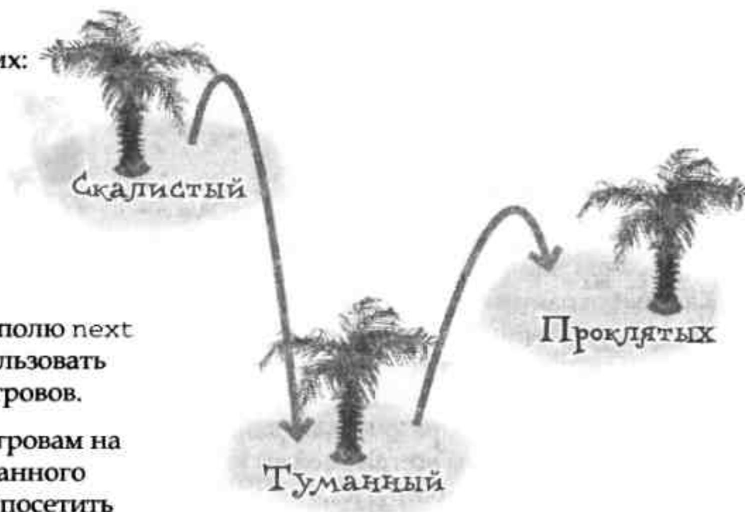
```
island amity = {"Дружбы", "09:00", "17:00", NULL};
island craggy = {"Скалистый", "09:00", "17:00", NULL};
island isla_nublar = {"Туманный", "09:00", "17:00", NULL};
island shutter = {"Проклятых", "09:00", "17:00", NULL};
```

Заметили, что изначально всем полям, указывающим на следующий остров, было присвоено значение NULL? Оно используется специально для обнуления *указателей* и на самом деле в Си обозначает 0.

...и свяжем их вместе в единый тур

Создав все острова, мы можем соединить их:

```
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
```



Будьте внимательны, когда присваиваете полю `next` *адрес* следующего острова. Вы будете использовать структуру переменных для каждого из островов.

Итак, мы создали полноценный тур по островам на языке Си. Но что если при перелете с Туманного острова на остров Проклятых мы захотим посетить островЧерепа?

Вставляем значения в список

Вы можете вставлять острова так же, как делали это ранее, — меняя значения указателей между островами:

В этой строке создается остров Черепа. →

```
island skull = {"остров Черепа", "09:00", "17:00", NULL};
isla_nublar.next = &skull;
skull.next = &shutter;
```

← Здесь Туманный остров соединяется с островом Черепа.

← Здесь остров Черепа соединяется с островом Проклятых.

С помощью всего двух строчек вы вставили в список новое значение. В случае с массивом нам бы пришлось написать куда больше кода, чтобы сдвинуть его элементы.

Теперь, когда вы узнали, как создавать и использовать связные списки, давайте применим приобретенные навыки на практике...

Магнитики с кодом

О нет, на дверях холодильника был код для функции `display()`**, но кто-то перемешал магнитики. Сможете ли вы собрать код заново?**

```
void display(island *start)
{
  island *i = start;

  for (; i ........... ...........; i ........... ........... ) {

    printf("Название: %s открыт: %s-%s\n", ........... , ..........., ...........);
  }
}
```

Магнитики с кодом Решение

О нет, на дверях холодильника был код для функции display(), но кто-то перемешал магнитики. Удалось ли вам собрать код?

```
void display(island *start)
{
  island *i = start;
  for (; i != NULL ; i = i->next ) {
    printf("Name: %s open: %s-%s\n", i->name , i->opens , i->closes );
  }
}
```

В начале цикла нам не нужен никакой дополнительный код.

Мы должны выполнять цикл, пока у текущего элемента есть ссылка на следующий остров.

В конце каждого повторения мы переходим к следующему острову.

не бывает Глупых Вопросов

В: Связные списки изначально встроены в некоторые языки, такие как Java. Есть ли в Си какие-нибудь структуры данных?

О: В языке Си нет никаких встроенных структур, вы сами должны создавать их.

В: Если я захочу использовать 700-й элемент в очень длинном списке, мне придется начать с самого первого и перебирать все элементы по очереди?

О: Да, именно так.

В: Это не очень хорошо. Я думал, что связной список окажется лучше массива.

О: Вы должны понимать, что структуры данных не бывают *хорошими* или *плохими*. Они либо *подходят* для вашей задачи, либо *не подходят*.

В: Значит, если мне нужна структура данных, в которую можно быстро вставлять элементы, то мне подойдет связной список, а если я хочу получить прямой доступ к элементам, то мне лучше использовать массив?

О: Именно.

В: Вы привели пример структуры, которая содержит указатель на другую структуру. Может ли такая структура рекурсивно хранить указатель на саму себя?

О: Нет.

В: Почему нет?

О: Компилятор должен знать, сколько именно места в памяти займет структура. Если она начнет рекурсивно копировать саму себя, каждый следующий фрагмент данных будет иметь другой размер.

Тест-драйв

Давайте опробуем функцию `display()` на связном списке островов и скомпилируем код в программу под названием `tour`.

```
island amity = {"остров Дружбы", "09:00", "17:00", NULL};
island craggy = {"Скалистый", "09:00", "17:00", NULL};
island isla_nublar = {"Туманный", "09:00", "17:00", NULL};
island shutter = {"остров Проклятых", "09:00", "17:00", NULL};
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
island skull = {"остров Черепа", "09:00", "17:00", NULL};
isla_nublar.next = &skull;
skull.next = &shutter;
display(&amity);
```

```
> gcc tour.c -o tour && ./tour
Название: остров Дружбы
Открыт: 09:00-17:00
Название: Скалистый
Открыт: 09:00-17:00
Название: Туманный
Открыт: 09:00-17:00
Название: остров Черепа
Открыт: 09:00-17:00
Название: остров Проклятых
Открыт: 09:00-17:00
>
```

Замечательно. В коде создается связной список островов, и мы можем легко вставлять в него новые элементы.

Теперь, когда мы овладели основами работы с рекурсивными структурами и списками, можно перейти к главной программе. Она будет считывать данные из файла, который выглядит следующим образом:

```
остров Дельфинов
Ангельский остров
остров Дикой Кошки
остров Нери
остров Большой Надежды
```

Дальше будет еще больше строчек.

Ребята, работающие на авиалиниях, все еще дописывают свой файл, поэтому его длину вы узнаете уже после запуска программы. Каждая строчка содержит название острова. Преобразование этого файла в связной список не должно вызывать затруднений, не так ли?

Деликатный путеводитель по стандартам

Новая переменная `skull` была объявлена прямо посреди кода, представленного на этой странице. Это допускается только в стандартах C99 и C11. В ANSI C все локальные переменные должны быть объявлены в начале функции.

Хм... Раньше мы для каждого элемента списка использовали отдельную переменную. Как мы узнаем, сколько переменных нам нужно, если мы не знаем длины файла? Может, мы должны генерировать новые переменные по мере необходимости?

Да, вам нужно создать *динамическое хранилище*.

Во всех программах, которые мы до сих пор писали, использовалось статическое хранилище. Каждый раз, когда нужно было что-то сохранить, вы добавляли в код переменную. Эти переменные, как правило, размещались в стеке. Напомним, стек — это область памяти для хранения локальных переменных.

Поэтому первые четыре острова мы создавали следующим образом:

```
island amity = {"остров Дружбы", "09:00", "17:00", NULL};
island craggy = {"Скалистый остров", "09:00", "17:00", NULL};
island isla_nublar = {"Туманный остров", "09:00", "17:00", NULL};
island shutter = {"остров Проклятых", "09:00", "17:00", NULL};
```

Каждому острову нужна собственная переменная. Но этот фрагмент кода всегда будет создавать только четыре острова. Для каждого нового потребуется новая локальная переменная. Это приемлемо, если на момент компиляции вам уже известно, сколько данных нужно будет хранить. Однако довольно часто программа не знает заранее, сколько места в хранилище ей понадобится для работы. При написании веб-браузера вы не будете знать, какое количество данных придется сохранить, пока... вы не прочтете веб-страницу. Поэтому у программ на языке Си должен быть какой-то способ сообщать операционной системе о том, что им нужно еще немного места в памяти.

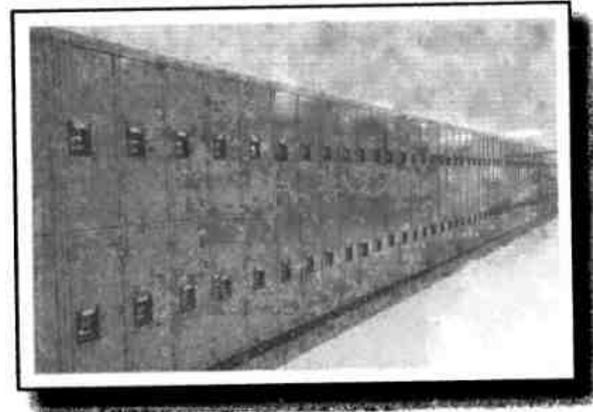
Программы нуждаются в динамическом хранилище.

Было бы здорово, если бы во время работы программы можно было выделять столько места, сколько мне нужно. Но это всего лишь мечты...

Используйте кучу в качестве динамического хранилища

Большая часть памяти, которую вы использовали до этого момента, находилась в **стеке**. Стек — это область памяти для хранения локальных переменных. Любой фрагмент данных хранится внутри переменной, и все переменные исчезают сразу после завершения функции.

Проблема в том, что во время выполнения программы сложно получить дополнительное место в стеке. Поэтому и была придумана **куча** — место, куда приложение складывает данные на долгосрочное хранение. Куча не очищается автоматически, следовательно, она идеально подходит для таких структур данных, как связной список. Процесс использования кучи можно представить в виде бронирования ячейки в камере хранения.



Работа с кучей похожа на хранение переменных в таких ячейках.

Сначала нужно получить свою память с помощью malloc()

Представьте, что во время работы ваша программа внезапно обнаружила большой объем данных, который она должна сохранить. Это все равно что попросить ключи от большой ячейки в хранилище. В языке Си это делается с помощью функции **`malloc()`**. Вы сообщаете функции, сколько памяти вам нужно, и она просит операционную систему выделить соответствующее место в куче. Затем функция `malloc()` возвращает **указатель** на выделенное место, что подобно получению ключей от ячейки. Данный указатель позволяет получать доступ к памяти, но его также можно использовать для наблюдения за выделенным местом в хранилище.

Куча.

Функция malloc() вернет вам указатель на место в куче.

Возвращайте обратно память, которая вам больше не нужна

Весомое преимущество кучи заключается в том, что вы можете хранить в ней данные по-настоящему долго. Но... это также является ее недостатком...

При использовании стека вам не нужно беспокоиться о возвращении памяти — это происходит автоматически. Каждый раз, когда функция завершает работу, локальное хранилище покидает стек.

С кучей все не так. Место, которое вы запросили в куче, остается занятым до тех пор, пока вы не скажете стандартной библиотеке Си, что оно больше не нужно. Объем хранилища ограничен, и, если ваш код будет просить все больше и больше места в куче, это очень быстро приведет к утечкам памяти.

Утечки памяти происходят из-за того, что программа все чаще запрашивает новое место, не освобождая память, которая ей больше не нужна. Это одна из наиболее часто встречающихся ошибок, свойственных программам на Си, которые иногда довольно сложно отследить.

Куча имеет фиксированный объем, поэтому старайтесь использовать ее разумно.

Вы можете освобождать память, вызывая функцию free()

Функция `malloc()` выделяет место в памяти и дает вам указатель на него. Указатель нужен для получения доступа к данным. Когда вы закончите работать с этим хранилищем, необходимо освободить память с помощью функции **`free()`**. Это все равно что вернуть ключ от ячейки смотрителю, чтобы ею мог воспользоваться кто-то другой.

Если какой-то участок вашего кода запрашивает место в куче с помощью функции `malloc()`, то в какой-то другой части программы должна вызываться функция `free()`, возвращающая это место. Когда ваше приложение завершает работу, все занятое им место в куче освобождается автоматически, но лучше делать это вручную, передавая функции `free()` каждый участок динамической памяти, который был создан.

Давайте посмотрим, как работают функции malloc() и free().

Запрашиваем память с помощью malloc()...

Функция, которая запрашивает память, называется `malloc()` (от *memory allocation* — «выделение памяти»). Она принимает один-единственный параметр — количество байтов, которое вам нужно. В большинстве случаев, скорее всего, точное количество необходимой памяти будет неизвестно, поэтому в сочетании с `malloc()` почти всегда используется функция `sizeof()`. Например:

```
#include <stdlib.h>
...
malloc(sizeof(island));
```

Вам нужно подключить заголовочный файл stdlib.h, чтобы получить доступ к функциям malloc() и free().

Это означает: «Дай мне столько места, чтобы хватило для хранения структуры island».

Функция `sizeof()` сообщает, сколько байтов занимает в вашей системе конкретный тип данных. Это может быть *структура* или один из базовых типов, таких как `int` или `double`.

Функция `malloc()` выделяет для вас отрезок памяти, после чего возвращает указатель с начальным адресом. Это будет *указатель общего назначения*, который вы должны привести к правильному типу, чтобы угодить компилятору:

```
island *p = malloc(sizeof(island));
```

Это означает: «Создать достаточно места для типа island и сохранить адрес в переменной p».

...и освобождаем память с помощью free()

Вы можете использовать память, созданную в куче, так долго, как пожелаете. Но, закончив с ней работать, вы должны освободить ее с помощью функции `free()`.

Функции `free()` нужно передать адрес в памяти, который был создан функцией `malloc()`. Зная, где начинается выделенная область памяти, стандартная библиотека может проверить свои записи и определить, какой объем памяти необходимо освободить. Таким образом, если вы хотите освободить память, которая была выделена в примере выше, вам нужно сделать следующее:

```
free(p);
```

Это означает: «Освободить память, которая была выделена в куче по адресу p».

Итак, узнав больше о динамической памяти, мы можем приступить к написанию кода.

Запомните: если в одной части программы вы выделили память с помощью функции malloc(), то в другой ее части вы должны освободить эту память, используя функцию free().

О нет! Это же безработные актеры...

Сейчас наши амбициозные актеры находятся в творческом отпуске. Они нашли время в своем плотном графике, чтобы помочь вам с программированием, и написали одну полезную функцию для создания островов на основе переданного названия. Эта функция выглядит следующим образом:

Это новая функция.

Название острова передается в виде символьного указателя.

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

Здесь в куче создается новая структура island.

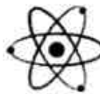
Для выделения места в куче используется функция malloc().

sizeof определяет, сколько байтов нам потребуется.

В этих трех строчках задаются поля новой структуры.

Функция возвращает адрес новой структуры.

Функция выглядит довольно круто. Актеры заметили, что все аэропорты открываются и закрываются в одно и то же время, поэтому задали полям `opens` и `closes` значения по умолчанию. Функция возвращает указатель на вновь созданную структуру.



Сила мозга

Взгляните внимательно на код функции `create()`. Как вы считаете, могут ли с ней возникнуть какие-то проблемы? Подумайте хорошенько, затем переверните страницу и посмотрите, как эта функция ведет себя на практике.

Дело об исчезнувшем острове

Дневник капитана. 11:00, пятница. Погода ясная. Написана функция `create()`, использующая динамическое выделение памяти. Команда программистов уверяет, что готова к воздушным испытаниям.

Загадка на пять минут

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

14:15. Погода облачная. Встречный северо-западный ветер возле Бермудских островов достигает 15 узлов. Совершаем первую посадку. Бортовая команда программистов предоставляет базовый код. Название острова вводится в командной строке.

Создаем массив для хранения названия острова. → `char name[80];`

Запрашиваем у пользователя название острова. → `fgets(name, 80, stdin);`

```
char name[80];
fgets(name, 80, stdin);
island *p_island0 = create(name);
```

```
File Edit Window Help
> ./test_flight
Атлантида
```

14:45. Тяжелый взлет из-за подземных толчков. Команда программистов все еще на борту. Тряска уменьшилась.

15:35. Прилет на второй остров. Погода хорошая. Ветра нет. Ввод подробной информации в новую программу.

Запрашиваем у пользователя название второго острова.

```
fgets(name, 80, stdin);
island *p_island1 = create(name);
p_island0->next = p_island1;
```

Здесь создается второй остров.

Здесь первый остров связывается со вторым.

```
остров Титчмарша
```

17:50. Вернулся в главный офис, привожу документы в порядок. Странное дело. Похоже, что в дневнике полета, выданном тестовой программой, содержится ошибка. В записях о сегодняшнем рейсе перелет к первому острову был загадочным образом переименован. Попросил команду программистов провести расследование.

Эта функция, которую мы создали ранее, выведет информацию о списке островов.

```
display(p_island0);
```

```
Название: остров Титчмарша
 открыт: 09:00-17:00
Название: остров Титчмарша
 открыт: 09:00-17:00
```

Что случилось с Атлантидой?

Теперь первый остров называется так же, как и второй!

Что случилось с названием первого острова? Неужели в функцию `create()` закралась ошибка? Можно ли об этом судить по способу вызова функции?

Дело об исчезнувшем острове

Что случилось с названием первого острова?

Еще раз взгляните на код функции `create()`:

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

Когда код записывает название острова, он не копирует всю строку, а использует только адрес в памяти, по которому она находится. Насколько важно, где находится строка с названием? Мы можем это выяснить, взглянув на код вызова функции:

```
char name[80];
 fgets(name, 80, stdin);
 island *p_island0 = create(name);
 fgets(name, 80, stdin);
 island *p_island1 = create(name);
```

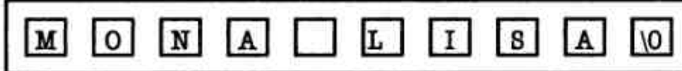
Программа запрашивает у пользователя название каждого острова, но в *обоих случаях* используется локальный символьный массив `name`. Это значит, что **оба острова делят одну строку с названием**. Присвоив название второго острова локальной переменной `name`, мы заодно переименовали и первый остров.

Подробнее о копировании строк

В языке Си вам часто придется копировать строки. Вы *могли бы* делать это с помощью функции `malloc()`, выделяя место в куче и копируя туда отдельно каждый символ из строки. Но, верите или нет, разработчики стандартной библиотеки Си додумались до этого раньше вас. Они создали в заголовочном файле *string.h* функцию под названием **`strdup()`**.

Представьте, что у вас есть указатель на символьный массив, который вам хотелось бы скопировать:

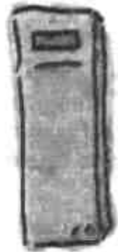
```
char *s = "MONA LISA";
```



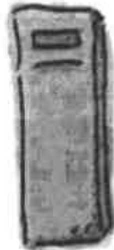
Функция **`strdup()`** может воспроизводить полную копию строки, находящуюся где-то в куче:

```
char *copy = strdup(s);
```

1. **Внутри функции strdup() стандартная библиотека определяет длину строки, а затем вызывает malloc(), чтобы выделить в куче подходящее количество символов.**



2. **Затем каждый символ копируется на новое место в куче.**



Из этого следует, что функция `strdup()` всегда выделяет место **в куче**. Она не может получить место в стеке, так как оно предназначено для *локальных переменных*, которые слишком часто очищаются.

Но поскольку `strdup()` помещает новую строку в кучу, вы **всегда должны помнить о том, что место для нее нужно освобождать с помощью функции `free()`.**

Давайте исправим наш код, применив функцию strdup()

Мы можем исправить исходную функцию `create()` с помощью функции `strdup()` следующим образом:

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = strdup(name);
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

Как видите, нужно всего лишь подставить `strdup()` в поле `name`. Вы можете объяснить, почему так?

Дело в том, что мы присваиваем полям `opens` и `closes` *строковые литералы*. Помните, мы говорили о том, где хранятся данные в памяти? Строковые литералы размещаются в той ее области, которая предназначена **только для чтения**, то есть специально для **констант**. Мы всегда присваиваем полям `opens` и `closes` константные значения, поэтому они никогда не меняются и нет необходимости копировать их на всякий случай. Но мы должны копировать массив `name`, так как позже он может измениться.

не бывает
Глупых Вопросов

В: Если бы поле `name` в структуре `island` являлось массивом, а не указателем типа `char`, то мне бы все равно нужно было использовать `strdup()`?

О: Нет. Каждая структура `island` хранила бы свою собственную копию `name`, поэтому вам не пришлось бы производить копирование самостоятельно.

В: Тогда зачем мне использовать в своих структурах указатель вместо массива символов?

О: Указатели типа `char` не ограничивают вас в объеме памяти, выделяемом для строк. Используя же массив символов, вы должны изначально решить, какую длину будет иметь строка.

Исправили ли мы код?

Чтобы увидеть, был ли исправлен код благодаря нашим изменениям, давайте еще раз его запустим:

Теперь код работает. Каждый раз, когда пользователь вводит название острова, функция `create()` сохраняет его в абсолютно новой строке.

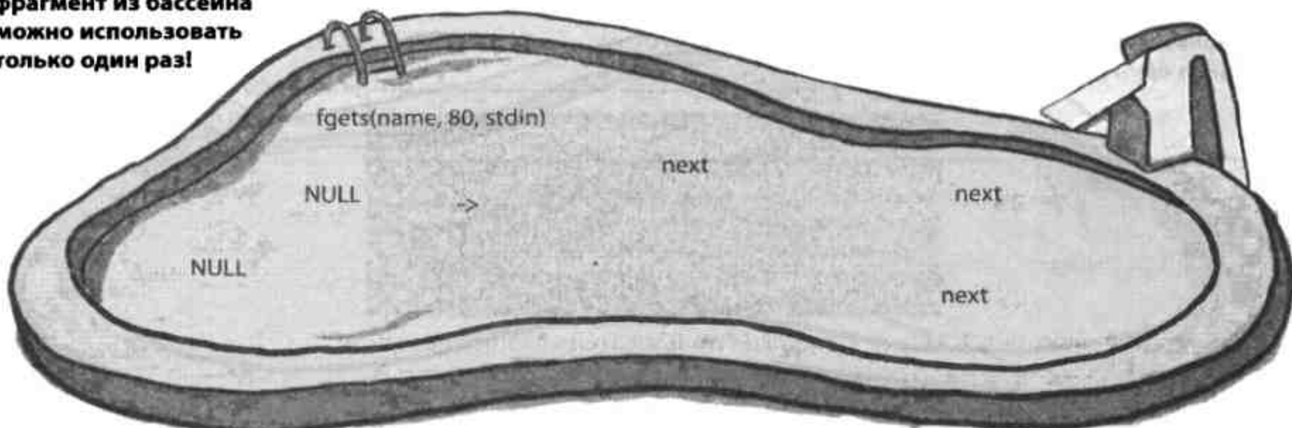
Теперь, когда у нас есть функция для заполнения информации об островах, давайте используем ее для создания связного списка из файла.

Головоломка у бассейна

Катастрофа! Код для создания туров по островам упал в бассейн! Вы **должны** выловить из бассейна фрагменты кода и заполнить пропуски. Ваша **цель** — воссоздать программу, чтобы она могла считывать перечень имен из стандартного ввода, объединяя их вместе в виде связного списка. Вы **не** можете использовать фрагмент кода более одного раза, не все фрагменты вам пригодятся.

```
island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; ............... != ............... ; i = ............... ) {
  next = create(name);
  if (start == NULL)
    start = ...............;
  if (i != NULL)
    i ............... ............... = next;
}
display(start);
```

Примечание: каждый фрагмент из бассейна можно использовать только один раз!



Головоломка у бассейна. Решение

Катастрофа! Код для создания туров по островам упал в бассейн! Вы **должны были** выловить из бассейна фрагменты кода и заполнить пропуски. Вашей **целью** было воссоздать программу, чтобы она могла считывать перечень имен из стандартного ввода, объединяя их вместе в виде связного списка.

```
island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; fgets(name, 80, stdin) != NULL ; i = next ) {
  next = create(name);
  if (start == NULL)
    start = next ;
  if (i != NULL)
    i -> next = next;
}
display(start);
```

Мы прочитаем строку из стандартного ввода.

В конце каждого повторения мы будем присваивать переменной i значение следующего созданного нами острова.

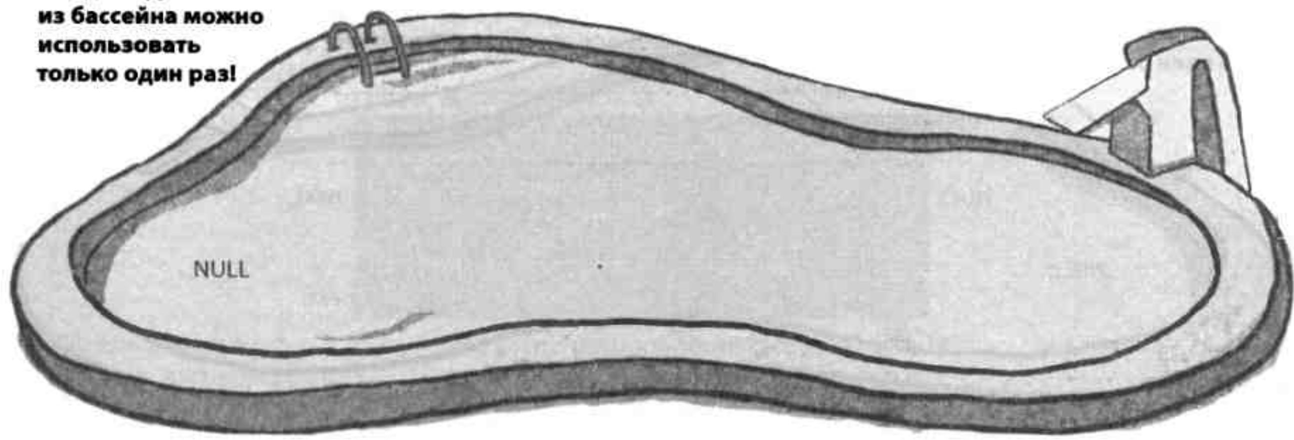
Здесь создается остров.

Мы будем продолжать выполнение цикла, пока не закончатся строки.

Изначально переменной start присваивается NULL, поэтому мы устанавливаем ей значение первого острова.

Не забывайте, i является указателем, поэтому мы будем использовать запись вида ->.

Примечание: каждый фрагмент из бассейна можно использовать только один раз!



Наточите свой карандаш

Но не спешите! Мы еще не закончили. Не забывайте, что однажды **выделив место** с помощью функции `malloc()`, мы должны **освободить его**, используя `free()`. Программа, которую мы перед этим написали, создает в куче связной список островов, используя функцию `malloc()`. Теперь пришло время написать код, чтобы освободить память, с которой мы закончили работать.

Мы начали с функции `release()`, которая, принимая указатель на первый остров, освобождает всю память, использованную в связном списке:

```
void release(island *start)
{
  island *i = start;
  island *next = NULL;
  for (; i != NULL; i = next) {
    next = ......................;
     ......................;
     ......................;
  }
}
```

Тщательно все обдумайте. Что именно вы будете удалять из памяти — только остров или что-то еще? В какой последовательности вы будете это делать?

Наточите свой карандаш
Решение

Но не спешите! Мы еще не закончили. Не забывайте, что однажды **выделив место** с помощью функции `malloc()`, мы должны **освободить его**, используя `free()`. Программа, которую мы перед этим написали, создает в куче связной список островов, используя функцию `malloc()`. Теперь пришло время написать код, чтобы освободить память, с которой мы закончили работать.

Мы начали с функции `release()`, которая, принимая указатель на первый остров, освобождает всю память, использованную в связном списке:

```
void release(island *start)
{
  island *i = start;
  island *next = NULL;
  for (; i != NULL; i = next) {
    next = i->next;
    free(i->name);
    free(i);
  }
}
```

Присвоим переменной next указатель на следующий остров.

Прежде всего мы должны освободить строку name, которую создали с помощью strdup().

Структуру island мы можем освободить только после удаления строки name.

Если бы мы сначала освободили island, мы не смогли бы получить и удалить поле name.

Вам нужно было тщательно обдумать, что именно следовало удалить из памяти — только остров или что-то еще — и в какой последовательности.

Освободите память после того, как закончите с ней работать

Теперь у вас есть функция для освобождения связного списка, которую нужно вызвать после того, как работа с этим списком будет закончена. Наша программа всего лишь выводит информацию об островах, поэтому по завершении вывода список можно сразу освободить:

```
display(start);
release(start);
```

Выполнив это, мы можем протестировать код.

Тест-драйв

Так что же произойдет после того, как мы скомпилируем код и пропустим через него файл?

```
File Edit Window Help FreeSpaceYouDon'tNeed
> ./tour < trip1.txt
Название: остров Дельфинов
 Открыт: 09:00-17:00
Название: Ангельский остров
 Открыт: 09:00-17:00
Название: остров Дикой Кошки
 Открыт: 09:00-17:00
Название: остров Нери
 Открыт: 09:00-17:00
Название: остров Большой Надежды
 Открыт: 09:00-17:00
Название: остров Рамита де ла Вайя
 Открыт: 09:00-17:00
Название: остров Голубых Дельфинов
 Открыт: 09:00-17:00
Название: остров Фантазий
 Открыт: 09:00-17:00
Название: остров Фарне
 Открыт: 09:00-17:00
Название: остров Исле де Муэрте
 Открыт: 09:00-17:00
Название: остров Марии-Терезы
 Открыт: 09:00-17:00
Название: остров Привидений
 Открыт: 09:00-17:00
Название: остров Шина
 Открыт: 09:00-17:00
```

Программа работает. Не забывайте, что мы не знаем длину файла заранее. В данном случае мы *могли бы* не размещать его содержимое в памяти, потому что единственная наша задача — вывести данные на экран. Но, поскольку файл был *сохранен*, у нас появляется возможность его изменять. Мы могли бы внести в наш тур дополнительные пункты назначения или удалить их, изменить или дополнить маршрут.

Благодаря динамическому выделению памяти вы можете сохранять ВО ВРЕМЯ ВЫПОЛНЕНИЯ ПРОГРАММЫ столько данных, сколько вам нужно. Доступ к куче (динамической области памяти) осуществляется с помощью функций `malloc()` и `free()`.

Задушевные беседы

В сегодняшнем выпуске: **Стек и Куча пытаются выяснить, чем они отличаются друг от друга**

Стек:

Куча, ты дома? Я уже пришел.

Куча:

Не часто тебя увидишь в это время суток. Что-то случилось?

Стек:

Произвожу глубокую регрессию. Ой… извини… Просто навожу порядок…

Куча:

Что ты делаешь?

Стек:

Код только что завершил выполнять функцию. Нужно освободить хранилище от локальных переменных.

Куча:

Ты должен относиться к жизни проще. Расслабься немного…

Стек:

Наверное, ты права. Не возражаешь, если я присяду?

Куча:

Пиво будешь? Не беспокойся о кепке, просто брось ее куда-нибудь.

Стек:

Кажется, это твое?..

Куча:

О, ты нашел пиццу. Отлично, я ищу ее уже целую неделю.

Стек:

Тебе действительно стоит поискать кого-нибудь, кто привел бы в порядок это место.

Куча:

Не волнуйся. Это осталось от Программы заказов по Интернету. Она еще, наверное, вернется и все заберет.

Стек:

Откуда ты знаешь? Как ты можешь быть уверена в том, что она просто не забыла об этом?

Куча:

Она вернется чуть позже. Она должна вызвать `free()`.

Стек:

Хм… Это точно? А кто ее автор? Случайно, не та женщина, которая написала безобразную игру «Ударь зайку»? Утечки памяти на каждом шагу. От этих структур с зайками я еле передвигался. Постоянные падения. Это было ужасно.

Стек:

Куча:

Эй, чистить память — это не моя работа. У меня просят место — я его выделяю и держу до тех пор, пока мне не скажут его освободить.

Это безответственно.

Ага, наверное. Но меня легко использовать. Не то что тебя с твоей... суетливостью.

Суетливостью? Я не суечусь! Может, тебе дать платок?..

<сморкается> Что? Я просто говорю, что за тобой сложно уследить.

А я считаю, что с памятью нужно обращаться должным образом.

Как бы то ни было, я живу и даю жить другим. Если программа хочет создать бардак — это не мое дело.

Ты неряха.

Я отношусь к жизни легко.

Почему бы тебе не выполнить сборку мусора?!

Ох, ну вот опять...

Ну хоть бы немного... прибралась. Ты же ничем не занята!

Полегче.

<всхлипывая> Извини. Я просто не могу смотреть на эту вопиющую безалаберность.

Эй, у тебя слезы потекли... Вот, возьми...

<сморкается> Спасибо. Стоп, это еще что такое?

Это таблица результатов игры «Ударь зайку». Не волнуйся, вряд ли она еще понадобится.

не бывает Глупых Вопросов

В: Почему у кучи такое название?

О: Потому что компьютер не систематизирует ее автоматически. Это буквально большая куча данных.

В: Что такое сборка мусора?

О: Некоторые языки следят за тем, как выделяется место в куче, освобождая его, когда данные больше не используются.

В: Почему в языке Си нет сборки мусора?

О: Си — довольно старый язык. Во времена его создания большинство языков программирования не выполняли автоматическую сборку мусора.

В: Я понимаю, зачем мы копировали название острова в предыдущем примере. Но почему мы не делали то же самое со значениями `opens` и `closes`?

О: : Значениям `opens` и `closes` присвоены строковые литералы. Строковой литерал не может быть изменен, поэтому нет ничего особенного в том, что несколько элементов данных ссылаются на одну и ту же строку.

В: Вызывается ли функция `malloc()` внутри `strdup()`?

О: Это зависит от реализации стандартной библиотеки. Но в большинстве случаев да.

В: Нужно ли мне освобождать все свои данные перед закрытием программы?

О: Нет, не нужно. Операционная система сама освободит всю память, которая использовалась программой. Но это хорошая привычка — очищать вручную все, что было вами создано.

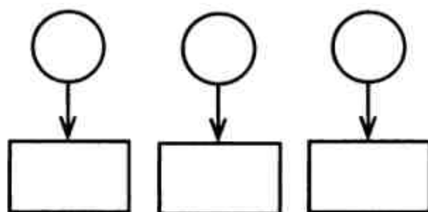
КЛЮЧЕВЫЕ МОМЕНТЫ

- Динамические структуры позволяют хранить переменное количество фрагментов данных.
- Связной список — это структура данных, которая позволяет с легкостью вставлять элементы.
- В языке Си динамические структуры данных обычно объявляются в виде рекурсивных структур.
- Рекурсивная структура содержит один или несколько указателей на такие же структуры.
- Стек используется для локальных переменных и управляется компьютером.
- Куча служит для длительного хранения. Выделение места осуществляется с помощью `malloc()`.
- Функция `sizeof()` сообщает о том, сколько места требуется для структуры.
- Данные остаются в куче до тех пор, пока вы сами не удалите их с помощью функции `free()`.

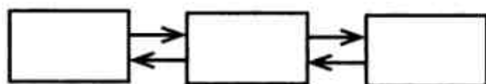
Попробуйте догадаться

Мы рассмотрели процесс создания связных списков в языке Си. Но в будущем вам могут понадобиться и другие структуры данных. Несколько примеров таких структур представлены ниже. Попробуйте подобрать для каждой из них соответствующее описание.

Структура данных | **Описание**

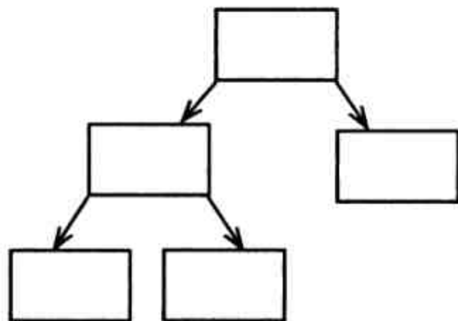


Меня можно использовать для хранения последовательности элементов. Я облегчаю процесс вставки новых данных. Но перебирать меня можно только в одном направлении.



Каждый элемент, который я храню, может быть связан с двумя другими элементами. Я могу пригодиться для хранения иерархической информации.

Меня можно использовать для объединения двух разных типов данных. Например, вы можете связать с моей помощью имя и соответствующий телефонный номер.



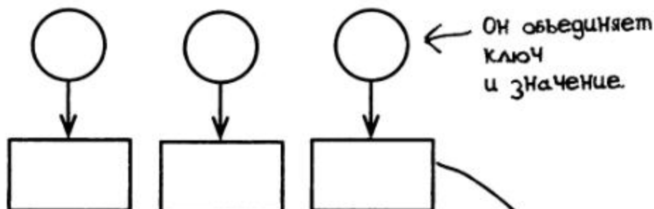
Каждый элемент, который я храню, может быть связан с двумя другими элементами. Вы можете перебирать меня в обоих направлениях.

Попробуйте догадаться

Решение

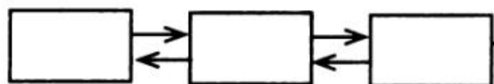
Мы рассмотрели процесс создания связных списков в языке Си. Но в будущем вам могут понадобиться и другие структуры данных. Несколько примеров таких структур представлены ниже. Вам нужно было подобрать для каждой из них соответствующее описание.

Ассоциативный массив или словарь



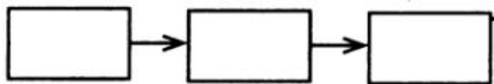
Он объединяет ключ и значение.

Двусвязной список

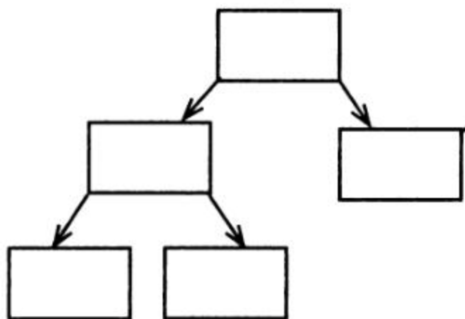


Похож на обычный связной список, но имеет ссылки в обоих направлениях.

Связной список



Двоичное дерево



Описание

Меня можно использовать для хранения последовательности элементов. Я облегчаю процесс вставки новых данных. Но перебирать меня можно только в одном направлении.

Каждый элемент, который я храню, может быть связан с двумя другими элементами. Я могу пригодиться для хранения иерархической информации.

Меня можно использовать для объединения двух разных типов данных. Например, вы можете связать с моей помощью имя и соответствующий телефонный номер.

Каждый элемент, который я храню, может быть связан с двумя другими элементами. Вы можете перебирать меня в обоих направлениях.

Структуры данных могут пригодиться, но будьте осторожны!

Создавая структуры данных на языке Си, вы должны быть осмотрительны. Если не следить тщательно за хранящимися данными, то может случиться так, что в куче останется старая бесполезная информация. Со временем это приведет к повышенному потреблению памяти и может вызвать появление ошибок и соответствующий сбой программы. **Поэтому очень важно научиться отслеживать и устранять утечки памяти в коде...**

СОВЕРШЕННО СЕКРЕТНО

Федеральное бюро расследований, Министерство юстиции США,
Вашингтон

От кого: Дж. Эдгар Гувер, директор

Тема: ПРЕДПОЛОЖИТЕЛЬНАЯ УТЕЧКА В ПРАВИТЕЛЬСТВЕННОЙ ЭКСПЕРТНОЙ СИСТЕМЕ

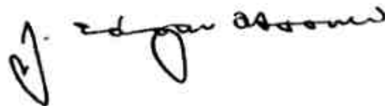
Наш офис в Кембридже докладывает о возможной утечке внутри новой Экспертной системы для идентификации подозрительных личностей (ЭСИПЛ). Источники и информаторы, ознакомленные с программными материалами, предполагают, что утечка могла стать результатом недобросовестной работы программистов или кого-то неизвестного.

Информатор, ранее предоставлявший надежные сведения и заявляющий о своей приближенности к причастным лицам, сообщает, что результатом утечки стало небрежное управление данными в области памяти, известной в хакерских кругах под названием «куча».

По моему поручению Вам выдан полный доступ к исходному коду экспертной системы. В Вашем распоряжении все ресурсы лаборатории программного обеспечения ФБР. Тщательно изучите улики и проанализируйте все обстоятельства этого дела. Данная утечка должна быть найдена и устранена.

Отказаться от дела Вы не можете.

Искренне Ваш,



Вещественное доказательство A: исходный код

Ниже представлен исходный код Экспертной системы для идентификации подозрительных личностей (ЭСИПЛ). Это программное обеспечение может применяться для записи и идентификации подозреваемых лиц. Сейчас вам не требуется детально изучать этот код, сохраните его в своих записях, чтобы в дальнейшем можно было бы сослаться на него в ходе текущего расследования.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node {
  char *question;
  struct node  *no;
  struct node  *yes;
} node;

int yes_no(char *question)
{
  char answer[3];
  printf("%s? (y/n): ", question);
  fgets(answer, 3, stdin);
  return answer[0] == 'y';
}

node* create(char *question)
{
  node *n = malloc(sizeof(node));
  n->question = strdup(question);
  n->no = NULL;
  n->yes = NULL;
  return n;
}

void release(node *n)
{
  if (n) {
    if (n->no)
      release(n->no);
    if (n->yes)
      release(n->yes);
    if (n->question)
      free(n->question);
    free(n);
  }
}
```

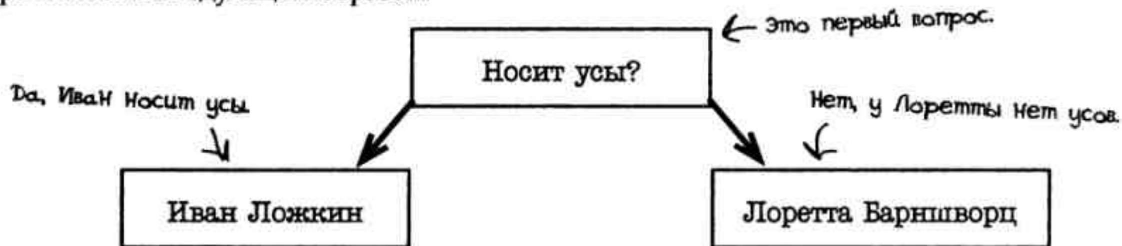
```
int main()
{
  char question[80];
  char suspect[40];
  node *start_node = create("Носит ли подозреваемый усы?");
  start_node->no = create("Лоретта Барнщворц");
  start_node->yes = create("Иван Ложкин");

  node *current;
  do {
    current = start_node;
    while (1) {
      if (yes_no(current->question))
      {
        if (current->yes) {
          current = current->yes;
        } else {
          printf("ПОДОЗРЕВАЕМЫЙ ОПОЗНАН\n");
          break;
        }
      } else if (current->no) {
        current = current->no;
      } else {

         /* Присваиваем yes_node имя нового подозреваемого */
         printf("Кто подозреваемый? ");
         fgets(suspect, 40, stdin);
         node *yes_node = create(suspect);
         current->yes = yes_node;

         /* Присваиваем no_node копию этого вопроса */
         node *no_node = create(current->question);
         current->no = no_node;

         /* Затем заменяем этот вопрос на новый */
         printf("Задайте мне вопрос, который подходит для %s, но не для %s. ", suspect,
            current->question);
         fgets(question, 80, stdin);
         current->question = strdup(question);

         break;
      }
    }
  } while(yes_no("Выполнить еще раз"));
  release(start_node);
  return 0;
}
```

Обзор системы ЭСИПЛ

Программа ЭСИПЛ — это экспертная система, которая учится идентифицировать людей по их отличительным признакам. Чем больше данных вы в нее введете, тем больше она узнает и тем умнее станет.

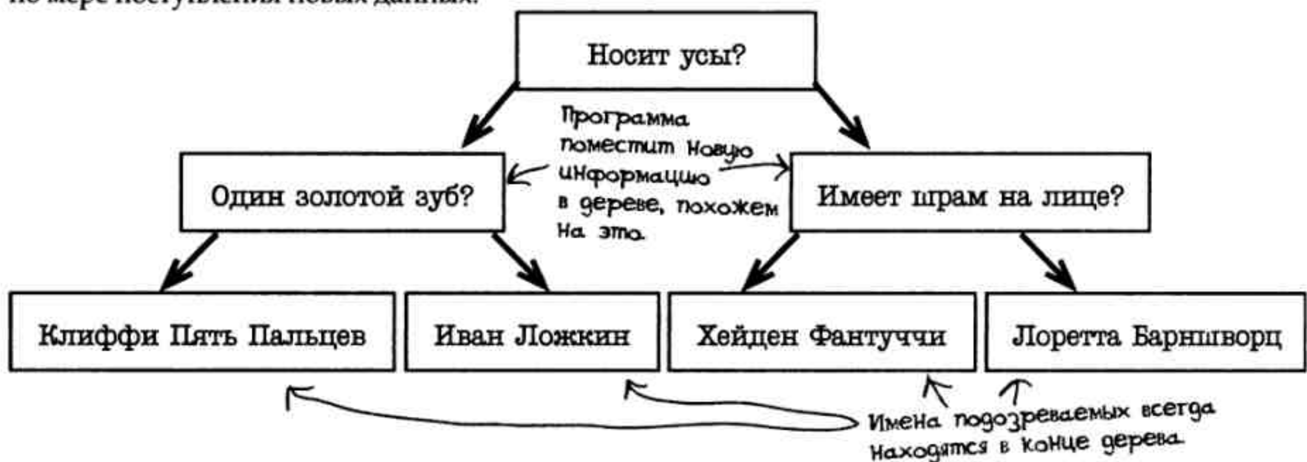
Программа формирует иерархию подозреваемых

Программа записывает данные с помощью **двоичного дерева**. *Двоичное дерево* позволяет соединить каждый фрагмент данных с двумя другими фрагментами следующим образом:



Так выглядят данные на момент запуска программы. Первый элемент (или **узел**) дерева содержит вопрос: «Носит ли подозреваемый усы?» Он соединен с двумя другими узлами: один для ***утвердительного*** ответа, а другой — для ***отрицательного***. Узлы *yes* и *no* хранят имена подозреваемых.

Программа будет использовать это дерево для идентификации подозреваемого, когда будет задавать пользователю ряд вопросов. Если она не сумеет обнаружить подозреваемого, то запросит у пользователя имя нового фигуранта дела и некоторые подробности, которые помогут его опознать. Программа поместит эту информацию в дереве, растущем по мере поступления новых данных.



Давайте посмотрим, как выглядит эта программа в действии.

Тест-драйв

Вот что произойдет, когда агент скомпилирует программу ЭСИПЛ и сделает пробный запуск:

```
File Edit Window Help TrustNoone
> gcc spies.c -o spies && ./spies
Носит ли подозреваемый усы? (y/n): n
Лоретта Барншворц? (y/n): n
Кто подозреваемый? Хейден Фантуччи
Задайте мне вопрос, который подходит для Хейден Фантуччи,
 но не для Лоретта Барншворц. Имеет шрам на лице
Выполнить еще раз? (y/n): y
Носит ли подозреваемый усы? (y/n): n
Имеет шрам на лице
? (y/n): y
Хейден Фантуччи
? (y/n): y
ПОДОЗРЕВАЕМЫЙ ОПОЗНАН
Выполнить еще раз? (y/n): n

>
```

При первой попытке программа не смогла опознать Хейдена Фантуччи. Но как только были введены подробности о подозреваемом, этого хватило, чтобы господин Фантуччи был идентифицирован со второй попытки.

Сообразительная программа. Так в чем же проблема?

Сотрудники лаборатории, использовавшие систему на протяжении нескольких часов, отмечали, что, несмотря на корректную работу, она потребляла почти **в два раза больше памяти**, чем ей на самом деле нужно.

Вот почему вызвали **вас**. Где-то в глубине исходного кода происходит выделение памяти в куче, и эта память потом *никогда больше не освобождается*. Вы можете прямо сейчас сесть и прочитать весь этот код в надежде, что на глаза попадется строчка, вызывающая проблему. Однако утечки памяти бывает очень сложно отследить.

Может быть, вам стоит заглянуть в лабораторию программного обеспечения?..

Экспертиза программного обеспечения: использование valgrind

На выявление ошибок в таких крупных и сложных проектах, как ЭСИПЛ, может уходить слишком много времени. Поэтому хакеры, использующие язык Си, написали несколько инструментов, которые могут помочь вам в работе. Один из таких инструментов применяется в **Linux** и называется **valgrind**.

valgrind может отслеживать фрагменты данных, для которых выделяется место в куче. Во время работы valgrind создает свою собственную **поддельную версию malloc()**. Когда ваша программа захочет выделить память в куче, valgrind перехватит вызовы функций malloc() и free() и заменит их своими версиями. malloc() от valgrind будет записывать вызовы на всех участках кода, отмечая, какие фрагменты памяти были выделены. По завершении работы программы valgrind выведет отчет обо всех данных, оставшихся в куче, и сообщит место кода, где они были созданы.

Подготовьте свой код: добавьте отладочную информацию

Вам *не нужно* менять свой код для использования valgrind. Вам даже не нужно его перекомпилировать. Но чтобы максимально использовать возможности этого инструмента, необходимо убедиться, что исполняемый файл содержит **отладочную информацию**. Отладочная информация добавляется в исполняемый файл при компиляции. Это данные вроде номера строки в исходнике, из которой скомпилировался конкретный фрагмент итогового кода. При наличии этой информации valgrind сможет выдать значительно больше подробностей о происхождении утечек памяти.

Чтобы добавить отладочную информацию в исполняемый файл, нужно перекомпилировать исходный код с ключом -g.

```
gcc -g spies.c -o spies
```

Ключ -g говорит компилятору, что помимо кода ему нужно записывать номера строчек.

Только факты: допросите свой код

Чтобы увидеть, как работает valgrind, давайте запустим его на компьютере под управлением Linux и используем несколько раз для допроса шпионов в программе ЭСИПЛ.

На странице http://valgrind.org вы можете узнать о доступности valgrind для вашей операционной системы и его установке.

Для начала попробуем опознать одного из имеющихся подозреваемых — Ивана Ложкина. Запустим valgrind в командной строке с параметром --leak-check=full, а затем передадим ему программу, работу которой хотим проверить:

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==1754== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Носит ли подозреваемый усы? (y/n): y
Иван Ложкин? (y/n): y
ПОДОЗРЕВАЕМЫЙ ОПОЗНАН
Выполнить еще раз? (y/n): n
==1754== All heap blocks were freed -- no leaks are possible
```

Используйте valgrind несколько раз, чтобы собрать больше улик

После завершения программы ЭСИПЛ в куче ничего не осталось. Но что если мы запустим программу еще раз и дадим ей информацию о новом подозреваемом — Хейдене Фантуччи?

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==2750== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Носит ли подозреваемый усы? (y/n): n
Лоретта Барншворц? (y/n): n
Кто подозреваемый? Хейден Фантуччи
Задайте мне вопрос, который подходит для Хейден Фантуччи,
 но не для Лоретта Барншворц. Имеет шрам на лице
Выполнить еще раз? (y/n): n
==2750== HEAP SUMMARY:
==2750==      in use at exit: 34 bytes in 1 blocks
==2750==    total heap usage: 11 allocs, 10 frees, 242 bytes allocated
==2750== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2750==    at 0x4026864: malloc (vg_replace_malloc.c:236)
==2750==    by 0x40B3A9F: strdup (strdup.c:43)
==2750==    by 0x8048587: create (spies.c:22)
==2750==    by 0x804863D: main (spies.c:46)
==2750== LEAK SUMMARY:
==2750==    definitely lost: 34 bytes in 1 blocks
>
```

В куче осталось 34 байта

Мы выделили 11 новых фрагментов памяти, но освободили только 10 из них.

Наталкивают ли нас эти строчки на какие-то мысли?

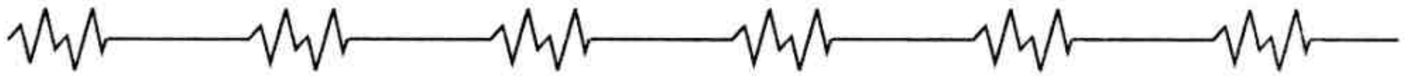
Почему 34 байта? Это улика?

На этот раз valgrind нашел утечку памяти

Похоже, после завершения программы в куче осталось 34 байта информации. Вот что нам говорит `valgrind`:

- 34 байта памяти было выделено, но не освобождено.
- Похоже, мы выделили 11 отрезков памяти, но освободили только 10 из них.
- Наталкивают ли нас эти строчки на какие-то мысли?
- Почему 34 байта? На что это указывает?

Давайте проанализируем эти несколько фактов.



Рассмотрим улики

Итак, мы запустили `valgrind` и собрали несколько улик. Самое время провести их анализ и попытаться сделать какие-то умозаключения.

1. Местоположение

Вы запускали код *два раза*. В первом случае проблем не возникло. Утечка памяти произошла только после того, как вы ввели имя нового подозреваемого. О чем это говорит? О том, что утечка не могла присутствовать в коде, который выполнялся в первый раз. Если еще раз вернуться к исходному коду, становится ясно, что проблема находится на этом участке:

```
} else if (current->no) {
  current = current->no;
} else {

   /* Присваиваем yes_node имя нового подозреваемого */
   printf("Кто подозреваемый? ");
   fgets(suspect, 40, stdin);
   node *yes_node = create(suspect);
   current->yes = yes_node;

   /* Присваиваем no_node копию этого вопроса */
   node *no_node = create(current->question);
   current->no = no_node;

   /* Затем заменяем этот вопрос на новый */
   printf("Задайте мне вопрос, который подходит для %s, но не для %s. ",
           suspect, current->question);
   fgets(question, 80, stdin);
   current->question = strdup(question);

   break;
}
```

2. Улики, собранные с помощью valgrind

Когда мы еще раз прогнали код через `valgrind`, программа выделила 11 фрагментов памяти, но освободила только 10. О чем это говорит?

`valgrind` сообщил нам, что после завершения программы в куче осталось 34 байта. Посмотрите на исходный код и подумайте, какой фрагмент данных мог бы занимать 34 байта?

И наконец, о чем вам говорит этот текст, полученный от `valgrind`?

```
==2750== 34 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2750==    at 0x4026864: malloc (vg_replace_malloc.c:236)
==2750==    by 0x40B3A9F: strdup (strdup.c:43)
==2750==    by 0x8048587: create (spies.c:22)
==2750==    by 0x804863D: main (spies.c:46)
```

СЕРЬЕЗНЫЕ ВОПРОСЫ

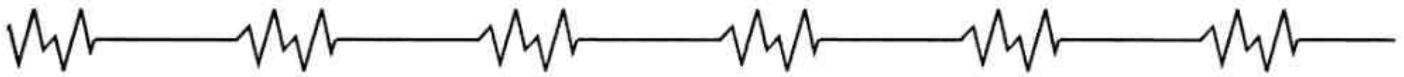
Внимательно рассмотрите все улики и ответьте на следующие вопросы.

1. Сколько фрагментов данных осталось в куче?

2. Какой из фрагментов данных остался в куче?

3. Что вызвало утечку — какая строки или строки кода?

4. Как устранить утечку?



Серьезные ответы

Вам нужно было внимательно рассмотреть все улики и ответить на следующие вопросы.

1. Сколько фрагментов данных осталось в куче?

Остался один фрагмент данных.

2. Какой из фрагментов данных остался в куче?

Строка «Лоретта Барнсворц» состоит из 17 символов — это 34 байта в кодировке UTF-8 .

3. Что вызвало утечку — какая строки или строки кода?

Мы знаем, что функция create() как таковая не вызывает утечек памяти, потому что она не делала этого при первом запуске. Следовательно, это должна быть строчка с strdup():

```
current->question = strdup(question);
```

4. Как устранить утечку?

Если current->question уже указывает на что-то, находящееся в куче, нам нужно освободить это место до того, как выделять новое под новый вопрос:

```
free(current->question);
current->question = strdup(question);
```

Проверяем на практике наше решение

Теперь, когда код исправлен, мы можем снова прогнать его через valgrind.

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==1800== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Носит ли подозреваемый усы? (y/n): n
Лоретта Барншворц? (y/n): n
Кто подозреваемый? Хейден Фантуччи
Задайте мне вопрос, который подходит для Хейден Фантуччи,
 но не для Лоретта Барншворц. Имеет шрам на лице.
Выполнить еще раз? (y/n): n
==1800== All heap blocks were freed -- no leaks are possible
>
```

Утечка устранена

Мы ввели в программу абсолютно те же тестовые данные, и на этот раз куча полностью очистилась. А как ваши успехи? Смогли ли вы раскрыть это дело? Не переживайте, если на этот раз вам не удалось обнаружить и устранить проблему. Утечка памяти — это одна из тех ошибок, свойственных программам на языке Си, которую труднее всего найти. На самом деле многие приложения, написанные на Си, с большой долей вероятности содержат в глубине себя ошибки работы с памятью. Вот почему так важны инструменты вроде valgrind.

- Определите, когда происходит утечка.
- Найдите место, где она происходит.
- Убедитесь, что утечка устранена.

не бывает Глупых Вопросов

В: `valgrind` сообщил, что утечка памяти образовалась в 46-й строчке, но мы исправляли совсем другой участок кода. Как же так?

О: Данные "Лоретта..." были размещены в куче в 46-й строке, но утечка произошла, когда указатель на нее (`current->question`) получил новое значение, не освободив память. Утечки памяти случаются не при создании данных, а в момент, когда программа теряет все ссылки на эти данные.

В: Могу ли я установить `valgrind` на свой компьютер с Mac/Windows/FreeBSD?

О: Подробности о последнем выпуске `valgrind` можно найти по адресу *http://valgrind.org*.

В: Каким образом `valgrind` перехватывает вызовы `malloc()` и `free()`?

О: Функции `malloc()` и `free()` содержатся в стандартной библиотеке Си. Но у `valgrind` есть свои собственные версии этих функций, и именно их будет использовать ваша программа, если ее запустить с помощью этого инструмента.

В: Почему компилятор при сборке программы не всегда добавляет отладочную информацию?

О: Потому что из-за отладочной информации исполняемый файл увеличится в размере. Это также может замедлить выполнение вашей программы.

В: Почему `valgrind` так называется?

О: Так называется главный вход в Валгаллу (в мифологии северных народов это место пребывания павших в сражении). Утилита `valgrind` дает вам доступ к компьютерной куче.

КЛЮЧЕВЫЕ МОМЕНТЫ

- `valgrind` ищет утечку памяти.
- Работа `valgrind` основывается на перехватывании вызовов функций `malloc()` и `free()`.
- Когда программа завершает работу, `valgrind` выводит информацию о том, что осталось в куче.
- `valgrind` может предоставить больше данных, если вы скомпилируете свой код с добавлением отладочной информации.
- Запустив программу несколько раз, вы сможете сузить область поиска утечки.
- `valgrind` может показать, в какой строчке исходного кода данные были помещены в кучу.
- С помощью `valgrind` можно проверить, устранена ли утечка.

Ваш инструментарий языка Си

Вы изучили главу 6, пополнив свои знания информацией о структурах данных и динамической памяти. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

Связной список более гибкий по сравнению с массивом.

В связной список можно легко вставлять данные.

Динамические структуры данных основаны на рекурсивных структурах.

Связной список является динамической структурой данных.

malloc() выделяет память в куче.

Рекурсивные структуры содержат одну или несколько ссылок на похожие данные.

free() освобождает память в куче.

В отличие от стека, память в куче не освобождается автоматически.

strdup() создает копию строки в куче.

Утечка памяти означает, что вы больше не можете получить доступ к данным, место для которых выделили.

Стек используется для локальных переменных.

valgrind поможет вам отследить утечки памяти.

7 Продвинутые функции

Выжмите из своих функций все соки

Базовые функции — это хорошо, но иногда нужно нечто большее. До сих пор мы уделяли внимание основам, но что если для достижения результата вам потребуется что-то более *мощное* и *гибкое*? В этой главе вы узнаете, как **усовершенствовать код, передавая функции в качестве параметров**, научитесь **сортировать элементы с помощью функций сравнения** и в завершение увидите, как можно сделать код *супергибким*, используя **функции с переменным числом аргументов**.

В поисках идеала

В этой книге вы уже имели дело со множеством функций, однако у вас есть возможность сделать их значительно более мощными. При правильном подходе функции могут **выполнять больше задач**, *не используя* дополнительного кода.

Разберемся на примере, как это работает. Представьте, что у вас есть массив строк, который нужно отфильтровать таким образом, чтобы одни строки отображались, а другие нет:

```
int NUM_ADS = 7;
char *ADS[] = {
 "Уильям: одинокий мужчина, афроамериканец, с хорошим чувством юмора, любит спорт,
 телевизор, перекусить",
 "Мэттью: одинокий мужчина, европеец, некурящий, любит живопись, кино, театр",
 "Луис: одинокий мужчина, латиноамериканец, непьющий, любит книги, театр, живопись",
 "Майк: разведенный мужчина, европеец, любит грузовики, спорт и Джастина Бибера",
 "Питер: одинокий мужчина, азиат, любит шахматы, тренироваться в зале и живопись",
 "Джош: одинокий мужчина, еврей, любит спорт, кино и театр",
 "Джед: разведенный мужчина, афроамериканец, любит театр, книги и перекусить"
};
```

Давайте напишем код, который с помощью строковых функций будет отфильтровывать этот массив.

Магнитики с кодом

Допишите функцию `find()` так, чтобы она могла находить в списке спортивных парней, которые **не** увлекаются творчеством Джастина Бибера.

Учтите: чтобы дописать эту функцию, вам могут понадобиться не все фрагменты.

```
void find()
{
  int i;
  puts("Результаты поиска:");
  puts("------------------------------------");

  for (i = 0; i              ........; i++) {

    if (    .......(..  .......,..  .......)..

            .......  .......  .......(..  .......,..  .......)) {

      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

strcmp

strstr

ADS[i]

ADS[i]

"спорт"

NUM_ADS

<

strstr

!

&&

||

"Бибер"

strcmp

Магнитики с кодом Решение

Вам нужно было дописать функцию `find()` таким образом, чтобы она могла находить в списке спортивных парней, которые **не** увлекаются творчеством Джастина Бибера.

```
void find()
{
  int i;
  puts("Search results:");
  puts("------------------------------------");

  for (i = 0; i < NUM_ADS ; i++) {
    if ( strstr ( ADS[i] , "спорт" )
        && ! strstr ( ADS[i] , "Бибер" )) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

strcmp

|| strcmp

Тест-драйв

Теперь, объединив данные и функцию в файле под названием `find.c`, вы можете скомпилировать и запустить программу следующим образом:

```
File Edit Window Help FindersKeepers
> gcc find.c -o find && ./find
Результаты поиска:
------------------------------------
Уильям: одинокий мужчина, афроамериканец,
с хорошим чувством юмора, любит спорт,
телевизор, перекусить
Джош: одинокий мужчина, еврей, любит спорт,
кино и театр
------------------------------------
>
```

Мне нужен тот, кто не курит и любит театр.

Найти того, кто любит спорт или тренировки в зале.

Найти того, кто любит живопись, театр или перекусить.

Как и следовало ожидать, функция `find()` перебирает в цикле элементы массива и находит соответствующие строки. Теперь, когда у вас есть базовый код, вы можете легко создавать *клоны* этой функции, которые будут выполнять другой вид поиска.

Эй, минутку! Клоны? **Клоны функции?** Это глупо. Вся разница между ними будет заключаться примерно в **одной строчке**.

Именно так, клонировав функцию, вы получите много повторяющегося кода.

Программы на Си часто должны выполнять *почти идентичные* задачи, отличающиеся друг от друга какой-то незначительной деталью. Сейчас у нас есть функция `find()`, которая перебирает все элементы массива, пытаясь найти совпадение в каждой строке. Но условие, которое она проверяет, остается **неизменным**. Она всегда проверяет одно и то же.

Вы можете передать функции какой-то параметр, чтобы она искала разные подстроки. Но проблема в том, что нельзя вести поиск сразу по *трем* словам, например «живопись», «театр» или «перекусить». А ведь позже вам может понадобиться что-то совсем другое.

Нам нужно нечто более замысловатое...

Передача кода в функцию

Необходимо каким-то образом упаковать фрагмент кода, отвечающий за проверку, и **передать его в функцию `find()`**. Это все равно что передать в функцию *проверочный агрегат*, через который можно будет пропустить каждый кусок данных.

Это означает, что большая часть функции `find()` останется **без изменений**. Она по-прежнему будет содержать код для проверки каждого элемента массива и выводить примерно такой же текст. Но саму проверку элементов будет производить *код, который вы ей передадите*.

Нам нужно передать в find() имя функции

Представьте, что мы взяли исходное условие для поиска и переписали его в виде функции:

```
int sports_no_bieber(char *s)
{
  return strstr(s, "спорт") && !strstr(s, "Бибер");
}
```

Если бы существовал какой-то способ передать в `find()` **имя этой функции в качестве** *параметра*, вы могли бы вставить проверку:

```
void find( имя-функции match      )
{
  int i;
  puts("Результаты поиска:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if ( вызвать-функцию-match    (ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

Вместо match нужно подставить имя функции, которая содержит код проверки.

Здесь нам нужно каким-то образом вызвать функцию, чье имя было передано в параметре match.

Если бы мы имели возможность передавать в `find()` имя функции, мы бы могли в дальнейшем производить совершенно любые проверки. Чтобы многократно использовать одну и ту же функцию `find()`, вам достаточно было бы написать код, который возвращает *true* или *false* на основании полученной строки.

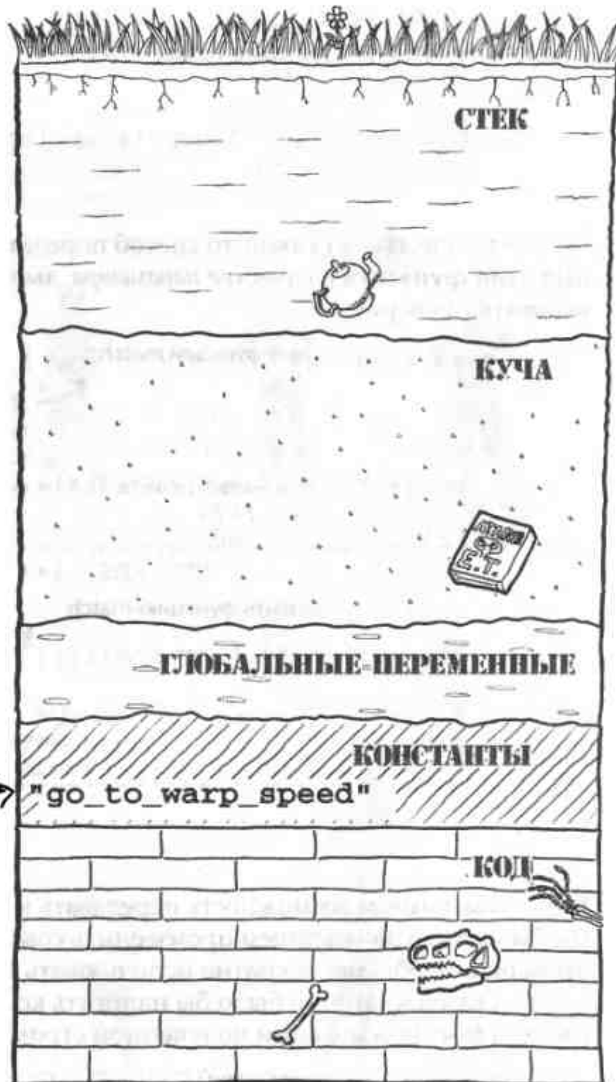
```
find(sports_no_bieber);
find(sports_or_workout);
find(ns_theater);
find(arts_theater_or_dining);
```

Но как указать, что параметр содержит имя функции? И как с помощью этого имени вызвать саму функцию?

Имя функции является указателем на эту функцию...

Вы, наверное, уже догадались, что где-то здесь должны быть замешаны указатели, не так ли? Подумайте, чем *на самом деле* является **имя функции**. Это способ *ссылаться* на участок кода. По сути это и есть указатель — ***способ ссылаться на что-либо в памяти***.

Вот почему в языке Си имена функций тоже являются переменными-указателями. Создавая функцию `go_to_warp_speed(int speed)`, вы одновременно создаете указатель с именем `go_to_warp_speed`, который содержит адрес этой функции. Таким образом, если мы передадим в `find()` параметр типа *указатель на функцию*, у нас появится возможность использовать его для вызова функции, на которую он ссылается.




```
int go_to_warp_speed(int speed)
{
  dilithium_crystals(ENGAGE);
  warp = speed;
  reactor_core(c, 125000 * speed, PI);
  clutch(ENGAGE);
  brake(DISENGAGE);
  return 0;
}
```

При создании любой функции вы также создаете одноименный указатель, который на нее ссылается.

Этот указатель содержит адрес функции.

```
go_to_warp_speed(4);
```

При вызове функции вы используете указатель на нее.

Давайте рассмотрим синтаксис языка Си, который понадобится для работы с указателями на функции.

...но такого типа данных, как функция, не существует

Как правило, объявление указателей в языке Си происходит довольно просто. Если у вас есть тип данных, например int, вам всего лишь нужно добавить к его названию звездочку — и вы получите указатель int *. К сожалению, в Си нет такого типа данных, как функция, поэтому вы не можете делать объявления вида function *.

```
int *a;        <- Здесь объявляется указатель типа int ...

function *f;   <- ...Но указатель на функцию так объявить нельзя.
```

Так почему же в языке Си нет типа function?

В языке Си нет такого типа, потому что не существует какого-то одного *типа* функций. При создании функции вы можете многое изменять — тип возвращаемого значения, список принимаемых ею параметров и т. д. Комбинацией этих признаков и определяется *тип* функции.

```
int go_to_warp_speed(int speed)
{
  ...
}

char** album_names(char *artist, int year)
{
  ...
}
```

Существует множество разных типов функций. Они отличаются друг от друга тем, что имеют разные параметры и возвращаемый тип.

Таким образом, для указателей на функции мы должны использовать чуть более сложную запись...

Как создать указатель на функцию

Если бы вам нужно было создать переменные-указатели, способные хранить адреса каждой функции, упомянутой на предыдущей странице, вы должны были бы написать следующее:

```
int (*warp_fn)(int);
warp_fn = go_to_warp_speed;
warp_fn(4);
```

Здесь создается переменная с именем warp_pointer, которая может хранить адрес функции go_to_warp_speed().

Это эквивалентно обычному вызову go_to_warp_speed(4).

```
char** (*names_fn)(char*,int);
names_fn = album_names;
char** results = names_fn("Sacha Distel", 1972);
```

Здесь создается переменная с именем names_fn, которая может хранить адрес функции album_names().

Выглядит довольно сложно, не так ли?

К сожалению, так оно и должно быть, потому что вам необходимо сообщить компилятору возвращаемый тип и типы параметров, которые функция будет принимать. Но, объявив указатель на функцию, вы можете использовать его так же, как и любую другую переменную: присваивать ему значение, добавлять в массив, передавать в другую функцию…

...что возвращает нас обратно к коду функции `find()`...

не бывает
Глупых Вопросов

В: Что означает `char**`? Это опечатка?

О: `char**` — это указатель, который используется для ссылки на массив строк.

Упражнение

Ниже представлены различные запросы по поиску людей. Подумайте, сможете ли вы создать функции для каждого запроса. Первую функцию мы уже написали:

Тот, кому нравится спорт и кто не любит Бибера.

```
int sports_no_bieber(char *s)
{
  return strstr(s, "спорт") && !strstr(s, "Бибер");
}
```

Найти того, кто любит спорт или тренировки в зале.

```
int sports_or_workout(char *s)
{

}
```

Мне нужен тот, кто не курит и любит театр.

```
int ns_theater(char *s)
{

}
```

Найти того, кто любит живопись, театр или перекусить.

```
int arts_theater_or_dining(char *s)
{

}
```

Затем постарайтесь дописать функцию `find()`:

```
void find( ............................................ )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

В find() нужно передать указатель на функцию с именем match.

Здесь будет вызвана функция match(), которую мы передали.

Упражнение Решение

Вам нужно было изучить различные запросы по поиску людей и создать функции для каждого запроса.

Тот, кому нравится спорт и кто не любит Бибера.

```
int sports_no_bieber(char *s)
{
  return strstr(s, "спорт") && !strstr(s, "Бибер");
}
```

Найти того, кто любит спорт или тренировки в зале.

```
int sports_or_workout(char *s)
{
    return strstr(s, "спорт") || strstr(s, "тренировки в зале");
}
```

Мне нужен тот, кто не курит и любит театр.

```
int ns_theater(char *s)
{
    return strstr(s, "некурящий ") && strstr(s, "театр");
}
```

Найти того, кто любит живопись, театр или перекусить.

```
int arts_theater_or_dining(char *s)
{
    return strstr(s, "живопись") || strstr(s, "театр") || strstr(s, "перекусить");
}
```

Затем постарайтесь дописать функцию `find()`:

```
void find( int (*match)(char*) )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

Тест-драйв

Давайте соберем эти функции вместе и посмотрим, как они работают. Нам нужно создать программу, которая будет вызывать `find()` с каждой функцией по очереди.

```
int main()
{
  find(sports_no_bieber);
  find(sports_or_workout);
  find(ns_theater);
  find(arts_theater_or_dining);
  return 0;
}
```

```
File Edit Window Help FindersKeepers
> ./find
Результаты поиска:
------------------------------------
Уильям: одинокий мужчина, афроамериканец, с хорошим
чувством юмора, любит спорт, телевизор, перекусить
Джош: одинокий мужчина, еврей, любит спорт, кино и театр
------------------------------------
Результаты поиска:
------------------------------------
Уильям: одинокий мужчина, афроамериканец, с хорошим
чувством юмора, любит спорт, телевизор, перекусить
Майк: разведенный мужчина, европеец, любит грузовики, спорт
и Джастина Бибера
Питер: одинокий мужчина, азиат, любит шахматы,
тренироваться в зале и живопись
Джош: одинокий мужчина, еврей, любит спорт, кино и театр
------------------------------------
Результаты поиска:
------------------------------------
Мэттью: одинокий мужчина, европеец, некурящий, любит
живопись, кино, театр
------------------------------------
Результаты поиска:
------------------------------------
Уильям: одинокий мужчина, афроамериканец, с хорошим
чувством юмора, любит спорт, телевизор, перекусить
Мэттью: одинокий мужчина, европеец, некурящий, любит
живопись, кино, театр
Луис: одинокий мужчина, латиноамериканец, непьющий, любит
книги, театр, живопись
Джош: одинокий мужчина, еврей, любит спорт, кино и театр
Джед: разведенный мужчина, афроамериканец, любит театр,
книги и перекусить
------------------------------------
>
```

Это find(sports_no_bieber).

Это find(sports_or_workout).

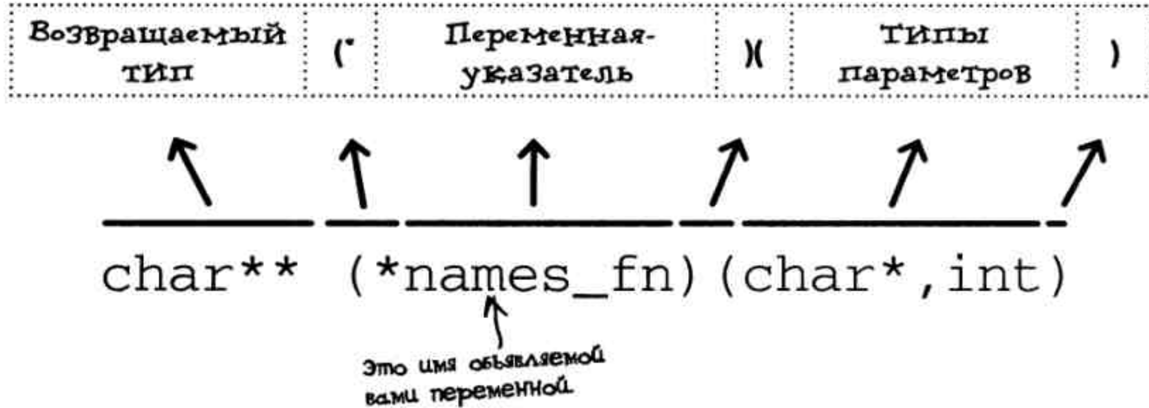
Это find(ns_theater).

Это find(arts_theater_or_dining).

При каждом новом вызове функции `find()` выполняется совершенно другой поиск. Вот почему указатели на функции — одна из наиболее мощных возможностей языка Си: они позволяют смешивать разный код. С помощью указателей на функции вы можете создавать **более мощные программы** с **меньшим количеством кода**.

Охотнику на заметку: указатели на функции

Когда вы прячетесь в камышах, указатели на функции довольно сложно разглядеть. Поэтому мы придумали компактную инструкцию, которая легко поместится в патронташ любого программиста на Си.




```
char** (*names_fn)(char*,int)
```

Это имя объявляемой вами переменной

не бывает Глупых Вопросов

В: Если указатели на функции ничем не отличаются от обычных указателей, почему тогда при вызове не нужно ставить перед ними *?

О: Вы можете это делать. Вместо `match(ADS[i])` можно было бы написать `(*match)(ADS[i])`.

В: А могли бы мы использовать &, чтобы получить адрес функции?

О: Да. Вместо `find(sports_or_workout)` вы могли бы написать `find(&sports_or_workout)`.

В: Почему тогда мы этого не сделали?

О: Потому что так код выглядит изящней. Компилятор все равно поймет, что вы от него хотите, даже если пропустите * и &.

Выполняем сортировку с помощью стандартной библиотеки Си

Многим программам нужно сортировать данные. В случае с простыми сущностями, такими как набор чисел, это делается довольно легко. У чисел есть свой собственный порядок. Но с другими типами данных все обстоит несколько сложнее.

Представьте, что у вас есть группа людей. По какому признаку вы их построите — по росту, интеллекту, привлекательности?

Поэтому, когда разработчики стандартной библиотеки Си решили написать функции для сортировки данных, они столкнулись с проблемой:

разве возможно написать набор основных функций для сортировки данных абсолютно любого типа?

Наводим порядок с помощью указателей на функции

Вы уже, наверное, догадались, каким было решение. В стандартной библиотеке Си есть множество разных сортировочных функций, и каждая из них принимает указатель на **функцию сравнения**, которая и будет определять, как один фрагмент данных соотносится с другим, — **больше**, **меньше** или **равен**.

Вот как выглядит функция `qsort()`:

```
qsort(void *array,
      size_t length,
      size_t item_size,
      int (*compar)(const void *, const void *));
```

Это указатель на массив. (← `void *array`)

Это длина массива. (→ `size_t length`)

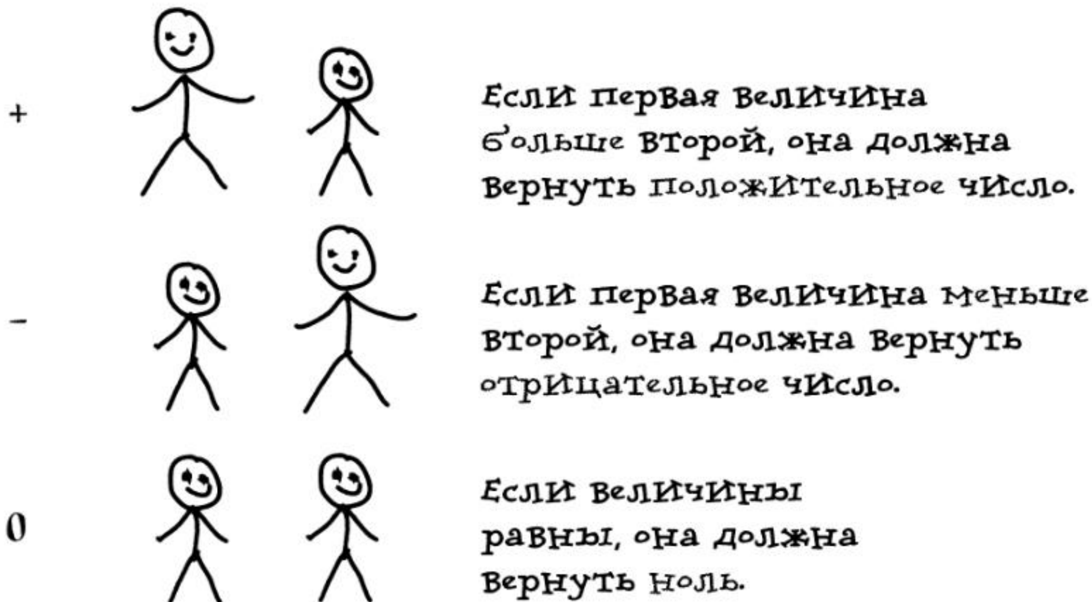
Это размер каждого элемента в массиве. (→ `size_t item_size`)

Не забывайте, что указатель void* может ссылаться на что угодно.

Это указатель на функцию, которая сравнивает два элемента в массиве. (↑ `compar`)

Функция `qsort()` раз за разом попарно сравнивает значения, и, если те находятся в неправильном порядке, компьютер меняет их местами.

Вот для чего нужна функция сравнения. Она сообщает `qsort()`, в каком порядке должны располагаться элементы, возвращая три разных значения:



Чтобы понять, как это работает на практике, давайте рассмотрим пример.

Подробнее о сортировке целых чисел

Представьте, что у вас есть массив целых чисел, и вы хотите отсортировать их в порядке возрастания. Как в таком случае будет выглядеть функция сравнения?

```
int scores[] = {543,323,32,554,11,3,112};
```

Если мы посмотрим на **сигнатуру** функции сравнения, которая требуется для `qsort()`, то увидим, что она принимает **два указателя пустого типа (void*)**. Помните, мы использовали этот тип вместе с `malloc()`? Пустой указатель может хранить адрес **данных любого вида**, но каждый раз перед использованием вам придется *приводить* его к какому-то конкретному типу.

Пустой указатель void* может ссылаться на что угодно.

Функция `qsort()` сравнивает попарно элементы в массиве и затем расставляет их в правильном порядке. Сравнение значений осуществляется с помощью соответствующей функции, которую вы ей предоставляете.

```
int compare_scores(const void* score_a, const void* score_b)
{
  ...
}
```

Значения в функцию всегда передаются в виде указателей, поэтому первым делом необходимо преобразовать их в целые числа:

```
int a = *(int*)score_a;
int b = *(int*)score_b;
```

Нам нужно привести пустой указатель к указателю на целое число.

Эта первая звездочка получает значение типа int, хранящееся по адресу score_b.

Функция сравнения вернула значение -21. Это значит, что число 11 должно находиться перед 32.

Затем вы должны вернуть положительное число, если a больше b, отрицательное — если a меньше b, и ноль — если числа равны. В случае с целыми значениями это делается довольно просто — нужно всего лишь вычесть одно число из другого:

```
return a - b;
```

Если a > b, результат будет положительным, если a < b – отрицательным, если a и b равны, получится ноль.

Вот так нужно вызывать функцию `qsort()`, чтобы она отсортировала массив:

```
qsort(scores, 7, sizeof(int), compare_scores);
```

Большое упражнение

Теперь ваша очередь. Взгляните на описания этих сортировок и попробуйте придумать для них функции сравнения. Чтобы вам было проще начать, первую функцию мы уже написали.

Сортирует целочисленные значения, начиная с наименьшего.

```
int compare_scores(const void* score_a, const void* score_b)
{
  int a = *(int*)score_a;
  int b = *(int*)score_b;
  return a - b;
}
```

Сортирует целочисленные значения, начиная с наибольшего.

```
int compare_scores_desc(const void* score_a, const void* score_b)
{



.........................................................................
}
```

Сортирует прямоугольники по площади, начиная с наименьшего.

```
typedef struct {
  int width;
  int height;
} rectangle;

int compare_areas(const void* a, const void* b)
{



.........................................................................
}
```

Это тип для прямоугольника.

Предупреждаем: эта функция по-настоящему сложная.

Сортирует список имен в алфавитном порядке, учитывая регистр.

```
int compare_names(const void* a, const void* b)
{
..........................................................
..........................................................
}
```

Вот вам подсказка: strcmp("Abc", "Def") < 0

Если строка — это указатель на char, то как будет выглядеть указатель на нее саму?

И наконец, после того как вы справитесь с функциями `compare_areas()` **и** `compare_names()`**, можете подумать над этими двумя:**

Сортирует прямоугольники по площади, начиная с наибольшей.

```
int compare_areas_desc(const void* a, void* b)
{
..........................................................
}
```

Сортирует список имен в обратном алфавитном порядке, учитывая регистр.

```
int compare_names_desc(const void* a, const void* b)
{
..........................................................
}
```

Большое упражнение

Решение

Теперь ваша очередь. Вам нужно было изучить описания предложенных сортировок и попробовать придумать для них функции сравнения.

Сортирует целочисленные значения, начиная с наименьшего.

```
int compare_scores(const void* score_a, const void* score_b)
{
  int a = *(int*)score_a;
  int b = *(int*)score_b;
  return a - b;
}
```

Эту функцию мы уже написали.

Сортирует целочисленные значения, начиная с наибольшего.

```
int compare_scores_desc(const void* score_a, const void* score_b)
{
int a = *(int*)score_a;
int b = *(int*)score_b;
return b - a;
}
```

Если при вычитании вы поменяете числа местами, то измените порядок итоговой сортировки.

Сортирует прямоугольники по площади, начиная с наименьшего.

```
typedef struct {
  int width;
  int height;
} rectangle;
```

Это тип rectangle.

```
int compare_areas(const void* a, const void* b)
{
rectangle* ra = (rectangle*)a;
rectangle* rb = (rectangle*)b;
int area_a = (ra->width * ra->height);
int area_b = (rb->width * rb->height);
return area_a - area_b;
}
```

Сначала мы приводим указатели к правильному типу.

Затем вычисляем площадь.

Теперь можем проделать фокус с вычитанием.

Сортирует список имен в алфавитном порядке, учитывая регистр.

Вот вам подсказка:
strcmp("Abc", "Def") < 0

```
int compare_names(const void* a, const void* b)
{
char** sa = (char**)a;
char** sb = (char**)b;
return strcmp(*sa, *sb);
}
```

Строка — это указатель на char, поэтому мы имеем дело с указателями на указатели.

Чтобы получить саму строку, нужно использовать оператор *.

И наконец после того как вы справились с функциями `compare_areas()` и `compare_names()`, вы должны были написать еще две функции:

Сортирует прямоугольники по площади, начиная с наибольшей.

```
int compare_areas_desc(const void* a, const void* b)
{
return compare_areas(b, a);
}
```

Вы также могли бы написать -compare_areas(a, b).

Сортирует список имен в обратном алфавитном порядке, учитывая регистр.

```
int compare_names_desc(const void* a, const void* b)
{
return compare_names(b, a);
}
```

Вы также могли бы написать -compare_names(a, b).

Расслабьтесь

Не переживайте, если при решении этих упражнений вы столкнулись с трудностями.

Помимо математики и обычных указателей здесь замешаны еще и указатели на функции. Если у вас что-то не получается, сделайте перерыв, выпейте немного водички и через пару часов попробуйте еще раз.

Тест-драйв

Некоторые функции сравнения получились довольно заковыристыми и стоят того, чтобы увидеть их в действии. Для их вызова вам понадобится примерно такой код.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

Здесь объявляются функции сравнения.

```
int main()
{
  int scores[] = {543,323,32,554,11,3,112};
  int i;
  qsort(scores, 7, sizeof(int), compare_scores_desc);
  puts("Это упорядоченные оценки:");
  for (i = 0; i < 7; i++) {
    printf("Оценка = %i\n", scores[i]);
  }
  char *names[] = {"Карен", "Марк", "Бретт", "Молли"};
  qsort(names, 4, sizeof(char*), compare_names);
  puts("Это упорядоченные имена:");
  for (i = 0; i < 4; i++) {
    printf("%s\n", names[i]);
  }
  return 0;
}
```

Это та строчка, где сортируются оценки.

qsort() меняет порядок размещения элементов в массиве.

Здесь после сортировки массива будет выведено на экран его содержимое.

Здесь будут сортироваться имена.

Не забывайте: массив имен — это просто массив указателей на char, поэтому размер каждого элемента равняется sizeof(char*).

Здесь выводятся отсортированные имена.

Скомпилировав и запустив этот код, мы получим следующее:

```
File Edit Window Help Sorted
> ./test_drive
Это упорядоченные оценки:
Оценка = 554
Оценка = 543
Оценка = 323
Оценка = 112
Оценка = 32
Оценка = 11
Оценка = 3
Это упорядоченные имена:
Бретт
Карен
Марк
Молли
>
```

Отлично, программа работает.

А теперь попробуйте написать свой собственный пример. Сортировочные функции могут оказаться чрезвычайно полезными, однако код, который им нужно передавать для сравнения, бывает довольно сложным. Но чем больше вы будете практиковаться, тем проще он будет выглядеть.

не бывает Глупых Вопросов

В: Я не понимаю, как работает функция сравнения для массива строк. Что означает `char**`?

О: Каждый элемент в строковом массиве является указателем на `char` (`char*`). Когда `qsort()` вызывает функцию сравнения, она передает ей указатели на два элемента в массиве. Следовательно, функция получает два указателя-на-указатели-на-char. В языке Си каждое такое значение записывается как `char**`.

В: Хорошо, но почему при вызове функции `strcmp()` мы пишем `strcmp(*a, *b)`, а не `strcmp(a, b)`?

О: `a` и `b` имеют тип `char**`. Функция `strcmp()` принимает значения типа `char*`.

В: Функция `qsort()` создает отсортированную версию массива?

О: Нет, в действительности она не создает копию, а изменяет исходный массив.

В: Теперь у меня разболелась голова...

О: Не волнуйтесь. Иногда указатели могут быть довольно сложными в использовании. Если они *не кажутся* вам слегка запутанными, это означает, что вы уделили им достаточно времени.

Автоматизируем составление уведомительных писем

Представьте, что вы создаете почтовую программу для отправки разного рода писем разным людям. Один из вариантов подготовки текста для каждого получателя заключается в создании структуры наподобие следующей:

Это три типа писем, которые можно отправлять.

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE};
typedef struct {
  char *name;
  enum response_type type;
} response;
```

Вместе с каждым фрагментом данных для писем мы будем записывать их тип.

В перечислении хранятся названия трех возможных типов писем, которые могут быть записаны вместе с самим письмом. Для каждого типа писем вы можете вызывать соответствующую функцию:

```
void dump(response r)
{
  printf("Дорогой %s,\n", r.name);
  puts("К сожалению, Ваш недавний партнер по свиданию связался с нами,");
  puts("чтобы сообщить, что Вы с ним больше не увидитесь");
}

void second_chance(response r)
{
  printf("Дорогой %s,\n", r.name);
  puts("Хорошие новости: Ваш недавний партнер по свиданию попросил нас");
  puts("организовать еще одну встречу. Пожалуйста, перезвоните как можно
  скорее.");
}

void marriage(response r)
{
  printf("Дорогой %s,\n", r.name);
  puts("Поздравляем! Ваш недавний партнер по свиданию");
  puts("связался с нами с предложением о браке.");
}
```

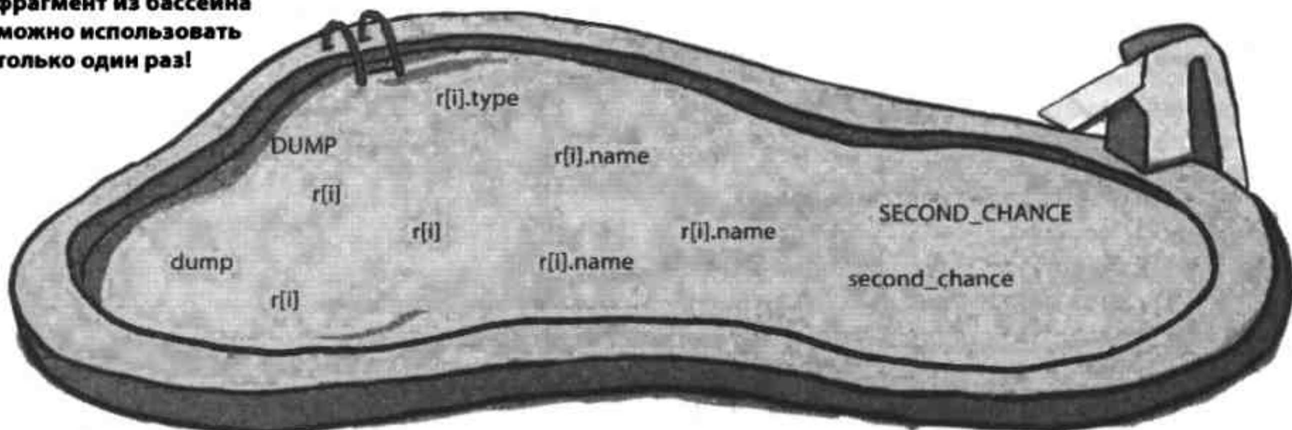
Теперь мы знаем, как выглядят данные и у нас есть функции для генерирования писем. Давайте посмотрим, насколько сложным будет код для создания набора писем из массива этих данных.

Головоломка у бассейна

Достаньте из бассейна фрагменты кода и вставьте их на место пропусков. Ваша задача — собрать воедино функцию `main()`, чтобы она могла генерировать набор писем из массива данных с ответами. Вы не можете использовать один и тот же фрагмент кода более одного раза.

```
int main()
{
  response r[] = {
    {"Майк", DUMP}, {"Луис", SECOND_CHANCE},
    {"Мэттью", SECOND_CHANCE}, {"Уильям", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
    switch(.........................) {
    case .........................:
      dump(.........................);
      break;
    case .........................:
      second_chance(.........................);
      break;
    default:
      marriage(.........................);
    }
  }
  return 0;
}
```

Примечание: каждый фрагмент из бассейна можно использовать только один раз!



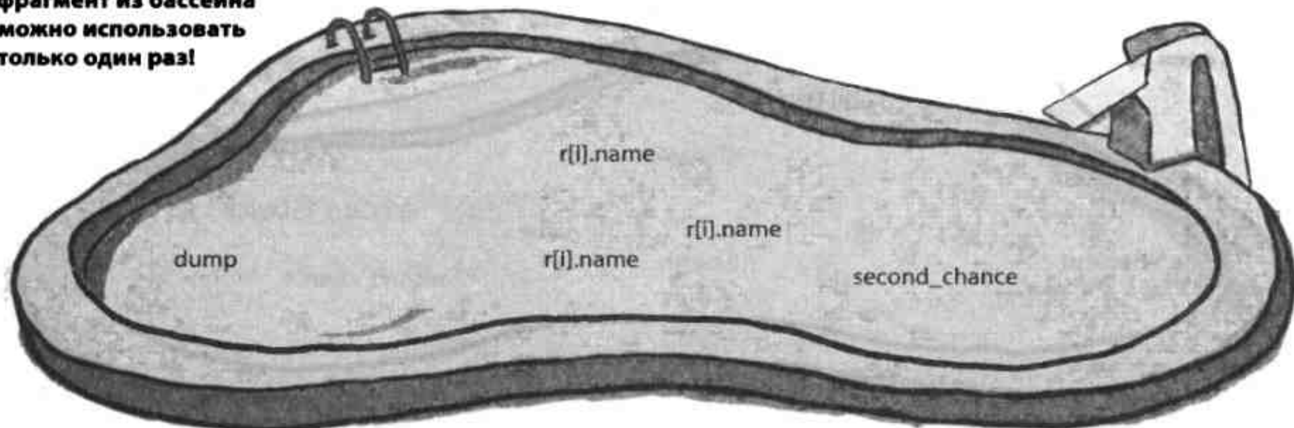
Головоломка у бассейна. Решение

Вам нужно было достать из бассейна фрагменты кода и вставить их на место пропусков, чтобы собрать воедино функцию `main()`, генерирующую набор писем из массива данных с ответами.

```
int main()
{
  response r[] = {
    {"Майк", DUMP}, {"Луис", SECOND_CHANCE},
    {"Мэттью", SECOND_CHANCE}, {"Уильям", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {        ← Перебираем массив.
    switch(r[i].type) {            ← Каждый раз мы проверяем поле type.
    case DUMP:
      dump(r[i]);
      break;
    case SECOND_CHANCE:
      second_chance(r[i]);
      break;
    default:
      marriage(r[i]);
    }
  }
  return 0;
}
```

Для каждого типа мы вызываем соответствующий метод.

Примечание: каждый фрагмент из бассейна можно использовать только один раз!



Тест-драйв

Когда мы запустим программу, можно не сомневаться, что она сгенерирует соответствующее письмо для каждого человека:

```
File Edit Window Help DontForgetToBreak
./send_dear_johns
Дорогой Майк,
К сожалению, Ваш недавний партнер по свиданию связался с нами,
чтобы сообщить, что Вы с ним больше не увидитесь.
Дорогой Луис,
Хорошие новости: Ваш недавний партнер по свиданию попросил нас
организовать еще одну встречу. Пожалуйста, перезвоните как
можно скорее.
Дорогой Мэттью,
Хорошие новости: Ваш недавний партнер по свиданию попросил нас
организовать еще одну встречу. Пожалуйста, перезвоните как
можно скорее.
Дорогой Уильям,
Поздравляем! Ваш недавний партнер по свиданию
связался с нами с предложением о браке
>
```

Что ж, программа работает — и это уже хорошо. Но на вызовы функций для каждого фрагмента данных приходится слишком много кода. Вызов любой функции для соответствующего типа письма будет выглядеть следующим образом:

```
switch(r.type) {
case DUMP:
  dump(r);
  break;
case SECOND_CHANCE:
  second_chance(r);
  break;
default:
  marriage(r);
}
```

А что произойдет, если мы добавим **четвертый** тип письма? Придется привести к единому виду каждый участок нашей программы. Очень скоро у нас может накопиться слишком много кода, который нужно будет поддерживать, и все может выйти из строя.

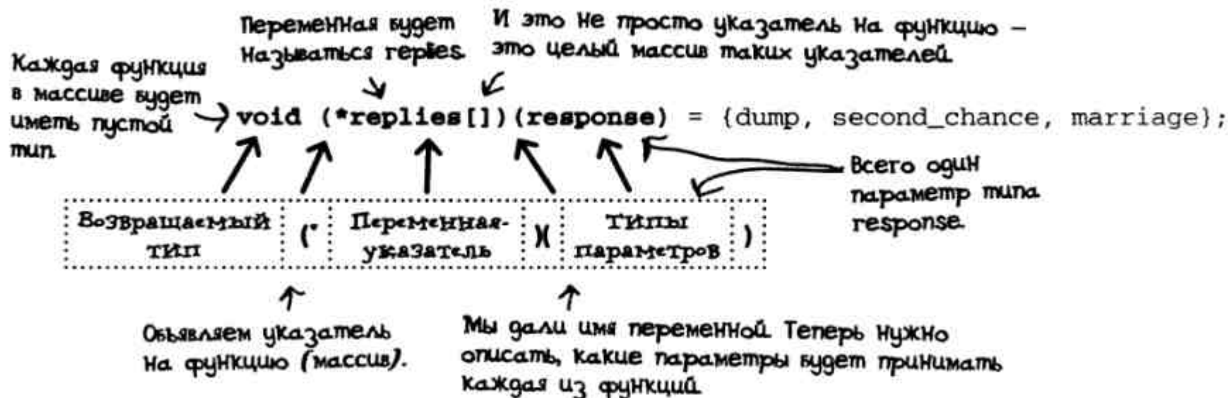
К счастью, язык Си позволяет воспользоваться одним хитрым приемом, связанным с **массивами**...

Создаем массив указателей на функции

Хитрость заключается в создании массива указателей на функции, которые соответствуют типам писем. Прежде чем увидеть, как это работает, давайте подумаем, как создать массив указателей на функции. Имея переменную массива, которая способна хранить имена любых функций, мы могли бы использовать ее вот так:

```
replies[] = {dump, second_chance, marriage};
```

Но такой синтаксис в языке Си не сработает. Мы должны сообщить компилятору, как именно будут выглядеть функции, которые мы собираемся сохранять в массиве, — какой тип они возвращают, какие параметры принимают. Следовательно, нам придется использовать **куда более сложный** синтаксис:



Но какая польза от этого массива?

Посмотрите, что представляет собой этот массив: он содержит набор имен функций, которые расположены **в том же порядке, что и типы внутри перечисления**:

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE};
```

Это *действительно важно*, потому что при создании перечисления компилятор присваивает каждому обозначению порядковый номер, начиная с 0. Поэтому `DUMP == 0`, `SECOND_CHANCE == 1` и `MARRIAGE == 2`. Благодаря этому изящному решению мы можем получить указатель на одну из наших функций с помощью **`response_type`**:

Это наш массив функций replies. → **`replies[SECOND_CHANCE] == second_chance`** ← Это эквивалентно имени функции second_chance.

SECOND_CHANCE равняется 1.

Посмотрим, сможете ли вы переписать нашу старую программу с помощью массива функций.

Наточите свой карандаш

Это упражнение не из легких. Но если вы уделите ему достаточно времени, то сможете с ним справиться. У вас есть вся необходимая информация, чтобы дописать этот код. В новой версии функции `main()` мы заменили всю конструкцию `switch/case` **единственной строчкой кода**. Эта строчка будет находить имя соответствующей функции и **вызывать ее**, используя массив `replies`.

```
void (*replies[])(response) = {dump, second_chance, marriage};

int main()
{
  response r[] = {
    {"Майк", DUMP}, {"Луис", SECOND_CHANCE},
    {"Мэттью", SECOND_CHANCE}, {"Уильям", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {

  }
  return 0;
}
```

Наточите свой карандаш
Решение

Это упражнение было достаточно трудным. В новой версии функции `main()` вся конструкция `switch/case` была удалена, вам нужно было заменить ее **единственной строчкой кода**. Эта строчка должна была находить имя соответствующей функции и **вызывать ее**, используя массив `replies`.

```
void (*replies[])(response) = {dump, second_chance, marriage};

int main()
{
  response r[] = {
    {"Майк", DUMP}, {"Луис", SECOND_CHANCE},
    {"Мэттью", SECOND_CHANCE}, {"Уильям", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
     (replies[r[i].type])(r[i]);
  }
  return 0;
}
```

При желании вы могли бы добавить * после открывающей скобки, но это ничего бы не изменило

Давайте разобьем это на отдельные части.

Все это является функцией наподобие dump или marriage.

(replies[r[i].type])(r[i]);

Это наш массив с именами функций.

Это значение будет равняться 0 для DUMP и 2 для MARRIAGE.

Мы вызываем функцию и передаем ей ссылочные данные r[i].

Тест-драйв

Запустив новую версию программы, мы получим на выходе то же, что и раньше:

```
File Edit Window Help WhoIsJohn
> ./dear_johns
Дорогой Майк,
К сожалению, Ваш недавний партнер по свиданию связался
с нами,чтобы сообщить, что Вы с ним больше не увидитесь
Дорогой Луис,
Хорошие новости: Ваш недавний партнер по свиданию
попросил нас организовать еще одну встречу. Пожалуйста,
перезвоните как можно скорее.
Дорогой Мэттью,
Хорошие новости: Ваш недавний партнер по свиданию
попросил нас организовать еще одну встречу. Пожалуйста,
перезвоните как можно скорее.
Дорогой Уильям,
Поздравляем! Ваш недавний партнер по свиданию связался
с нами с предложением о браке.

>
```

Разница только в том, что вместо целой конструкции `switch` мы имеем всего лишь это:

```
(replies[r[i].type])(r[i]);
```

Если нам нужно будет вызывать функцию для генерации писем в нескольких участках программы, нам не придется копировать весь код. И если мы решим добавить новые тип и функцию, нам достаточно будет просто поместить их в массив:

Таким образом, вы можете добавлять новые типы и функции.

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE, LAW_SUIT};
void (*replies[])(response) = {dump, second_chance, marriage, law_suit};
```

Массив указателей на функции может облегчить работу с кодом и сделать его более коротким, *масштабируемым* и расширяемым. Даже несмотря на то что поначалу такие массивы кажутся довольно сложными для понимания, они могут серьезно улучшить ваши навыки программирования на Си.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Указатели на функции хранят адреса этих функций.
- Имя любой функции на самом деле является указателем на нее.
- Если у вас есть функция `shoot()`, то и `shoot`, и `&shoot` — указатели.
- Новый указатель на функцию объявляется так: `возвращаемый-тип(*имя-переменной)(типы-параметров)`.
- Если `fp` — это указатель на функцию, то саму функцию с параметрами вы можете вызвать так: `fp(params, ...)`...
- ...или вот так: `(*fp)(params,...)` — в Си это будет одно и то же.
- Стандартная библиотека Си содержит сортировочную функцию `qsort()`.
- `qsort()` принимает указатель на *функцию сравнения*, которая может выполнять проверку на (не)равенство.
- В функцию сравнения будут переданы **указатели** на два элемента массива.
- Если у вас есть массив данных, вы можете связать отдельную функцию с каждым его элементом, используя массив указателей на функции.

не бывает Глупых Вопросов

В: Почему синтаксис массивов указателей на функции такой сложный?

О: Потому что при объявлении указателя на функцию вам нужно сообщить типы ее параметров и возвращаемого значения. Именно поэтому нужно столько скобок.

В: Это похоже на какой-то вид объектно ориентированного кода из других языков, правда?

О: Да, похоже. В объектно-ориентированных языках набор функций (*методов*) привязан к данным. Таким же образом вы можете привязать данные к указателям на функции.

В: Эй, неужели это означает, что Си — объектно-ориентированный язык? Ого, это круто.

О: Нет, Си не является объектно-ориентированным языком. Но другие языки, основанные на Си (например, Objective-C и Си++), многими своими объектно ориентированными возможностями обязаны указателям на функции.

Сделайте свои функции эластиииииичными

Иногда возникает потребность в по-настоящему *мощных* функциях, как, например, find(), выполняющей поиск с помощью указателей на другие функции. Но иногда от функций требуется только *простота в использовании*. Возьмем, скажем, printf(). У этой функции есть одна действительно уникальная возможность, которой мы часто пользовались, но о которой мало говорили, — она способна принимать **переменное количество аргументов:**

```
printf("%i бутылок пива на стене, %i бутылок пива\n", 99, 99);
printf("Возьми одну, пусти по кругу,");
printf("%i бутылок пива на стене\n", 98);
```

Вы можете передать в printf() столько аргументов, сколько вам нужно вывести на экран.

Так как же ВАМ САМИМ сделать что-то подобное?

Нам как раз подвернулась подходящая задача. Ребята из бара Head First обнаружили, что им становится сложновато уследить за общим количеством напитков. Один из парней попытался упростить всем жизнь и создал перечисление со списком доступных коктейлей, а также функцию, которая возвращает фрагменты данных для каждого из них:

```
enum drink {
  MUDSLIDE, FUZZY_NAVEL, MONKEY_GLAND, ZOMBIE
};

double price(enum drink d)
{
  switch(d) {
  case MUDSLIDE:
    return 6.79;
  case FUZZY_NAVEL:
    return 5.31;
  case MONKEY_GLAND:
    return 4.82;
  case ZOMBIE:
    return 5.89;
  }
  return 0;
}
```

Это довольно удобно, если нужно получить цену одного напитка. Но что они будут делать, если им потребуется цена всего заказа?

Просто. → `price(ZOMBIE)`

`total(3, ZOMBIE, MONKEY_GLAND, FUZZY_NAVEL)` ← Не так уж просто.

Количество напитков.

Список напитков в заказе.

Им нужна функция total(), которая будет принимать количество напитков и список их названий.

Подробнее о функциях с переменным количеством параметров

Принцип работы **функции с переменным** количеством параметров понятен из ее названия. Стандартная библиотека Си содержит набор **макросов,** которые могут помочь вам в создании собственных функций такого рода.

Вы можете воспринимать макрос как специальный вид функций, которые могут изменять исходный код.

```
print_ints(3, 79, 101, 32);
```

Количество чисел, которые нужно вывести

Числа, которые нужно вывести

Это код:

Здесь будут находиться переменные аргументы.

Переменные аргументы будут следовать после параметра args.

```
#include <stdarg.h>

void print_ints(int args, ...)
{
  va_list ap;
  va_start(ap, args);
  int i;
  for (i = 0; i < args; i++) {
    printf("аргумент: %i\n", va_arg(ap, int));
  }
  va_end(ap);
}
```

Это обычный аргумент, который будет передаваться всегда.

va_start указывает на то, где начинаются переменные аргументы.

Здесь идет перебор всех остальных аргументов.

args содержит информацию о количестве переменных аргументов.

Давайте разобьем его на части и последовательно разберем.

1. **Подключите заголовок stdarg.h.**
Весь код для работы с функциями, имеющими переменное количество параметров, находится в файле *stdarg.h*, поэтому нужно убедиться, что вы его подключили.

2. **Объясните вашей функции, что аргументов может быть сколько угодно...**
Знаете, бывают такие книги, в которых героиня оказывается с парнем наедине, после чего глава заканчивается *многоточием*. Это означает, что дальше будет продолжение. В языке Си многоточие после аргумента функции говорит о том, что дальше следует ожидать еще больше аргументов.

Конечно, мы тоже не читаем такие книги.

3. **Создайте va_list.**
`va_list` будет использоваться для хранения дополнительных аргументов, передаваемых в вашу функцию.

4. **Укажите, где начинаются переменные аргументы.**
Компилятор должен знать имя **последнего постоянного аргумента**. В случае с нашей функцией это будет параметр `args`.

5. **Затем поочередно считайте переменные аргументы.**
Теперь, когда все ваши аргументы хранятся внутри `va_list`, вы можете считать их с помощью функции `va_arg()`. Эта функция принимает два значения: `va_list` и **тип** следующего аргумента. В нашем случае все аргументы имеют тип `int`.

6. **И напоследок... завершите список.**
После того как вы прочитали все аргументы, компилятору необходимо сообщить об этом с помощью макроса `va_end`.

7. **Теперь вы можете вызвать свою функцию.**
Закончив написание функции, вы можете ее вызвать:

```
print_ints(3, 79, 101, 32);
```

Эта строчка выведет значения 79, 101 и 32

Уголок ботана

Функции или макросы?

Макросы используются для изменения кода перед компиляцией. Применяемые здесь `va_start`, `va_arg` и `va_end` могут выглядеть как обычные функции, но на самом деле они содержат внутри скрытые инструкции для *препроцессора*, благодаря которым перед компиляцией генерируется множество дополнительного полезного кода для программы.

не бывает

Глупых Вопросов

В: Погодите, почему `va_end()` и `va_start()` называются *макросами*? Разве это не обычные функции?

О: Нет, они созданы таким образом, чтобы внешне не отличаться от обыкновенных функций, но на самом деле препроцессор замещает их другим кодом.

В: А препроцессор — это...

О: Препроцессор запускается прямо перед этапом компиляции. Помимо прочего, он подключает в код заголовочные файлы.

В: Могу ли я создать функцию, у которой не будет постоянных параметров, *только* переменные?

О: Нет. Нужен как минимум один постоянный параметр, чтобы передать его имя в функцию `va_start()`.

В: А что если я попытаюсь прочитать из `va_arg()` больше аргументов, чем было передано?

О: Будут происходить случайные ошибки.

В: Звучит нехорошо.

О: Да, довольно плохо.

В: Что произойдет, если я попытаюсь прочитать целочисленный аргумент как `double` или нечто в этом роде?

О: Произойдет случайная ошибка.

Упражнение

Теперь дело за вами. Парни из бара Head First хотят создать функцию, которая сможет возвращать общую стоимость напитков. Что-то наподобие этого:

```
printf("Цена равняется %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
```

Здесь будет выведено «Цена равняется 16.9».

Допишите код `total()`, используя функцию `price()`, которая была рассмотрена на предыдущих страницах:

```
double total(int args, ...)
{
  double total = 0;




  ..............................................

  return total;
}
```

Упражнение Решение

Теперь дело за вами. Парни из бара Head First хотят создать функцию, которая сможет возвращать общую стоимость напитков. Что-то наподобие этого:

```
printf("Цена равняется %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
```

Здесь будет выведено «Цена равняется 16.9».

Допишите код `total()`, используя функцию `price()`, которая была рассмотрена на предыдущих страницах:

Не переживайте, если ваш код не выглядит в точности как этот. Его можно написать разными способами.

```
double total(int args, ...)
{
  double total = 0;
  va_list ap;
  va_start(ap, args);
  int i;
  for(i = 0; i < args; i++) {
    enum drink d = va_arg(ap, enum drink);
    total = total + price(d);
  }
  va_end(ap);
  return total;
}
```

Тест-драйв

Если мы напишем немного тестового кода для вызова функции, мы сможем скомпилировать его и посмотреть, что произойдет:

Это тестовый код

```
main(){
  printf("Цена равняется %.2f\n", total(2, MONKEY_GLAND, MUDSLIDE));
  printf("Цена равняется %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE,
  FUZZY_NAVEL));
  printf("Цена равняется %.2f\n", total(1, ZOMBIE));
  return 0;
}
```

```
File Edit Window Help Cheers
> ./price_drinks
Цена равняется 11.61
Цена равняется 16.92
Цена равняется 5.89
>
```

Это программный вывод

Ваш код работает!

Теперь вы знаете, как с помощью переменных аргументов упростить свой код и сделать его интуитивно понятным в использовании.

О да, детка! Я все помню даже после того, как было подано столько коктейлей...

КЛЮЧЕВЫЕ МОМЕНТЫ

- Функции, способные принять неопределенное число аргументов, называются **функциями с переменным количеством параметров**.
- Чтобы создать функцию с переменным количеством параметров, нужно подключить заголовочный файл *stdarg.h*.
- Переменные параметры будут записаны в `va_list`.
- Вы можете работать со списком `va_list` с помощью `va_start()`, `va_arg()` и `va_end()`.
- Вам понадобится как минимум один *обычный параметр*.
- Будьте внимательны: не пытайтесь прочитать параметров больше, чем было передано.
- Вам всегда нужно знать тип каждого параметра, который вы считываете.

Ваш инструментарий языка Си

Вы изучили главу 7, пополнив свои знания информацией о продвинутых функциях. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

Указатели на функции позволяют обмениваться кодом так же, как данными.

Только указателям на функции не требуются операторы * и &...

...Но при желании вы все равно можете использовать операторы * и &.

qsort() отсортирует массив.

Имя любой функции является указателем на нее.

Каждой сортировочной функции нужен указатель на функцию сравнения.

stdarg.h позволяет создавать функции с переменным количеством параметров.

Массивы указателей на функции могут помочь запускать разные функции для разных типов данных.

Функция сравнения определяет порядок, в котором должны располагаться два фрагмента данных.

8 Статические и динамические библиотеки

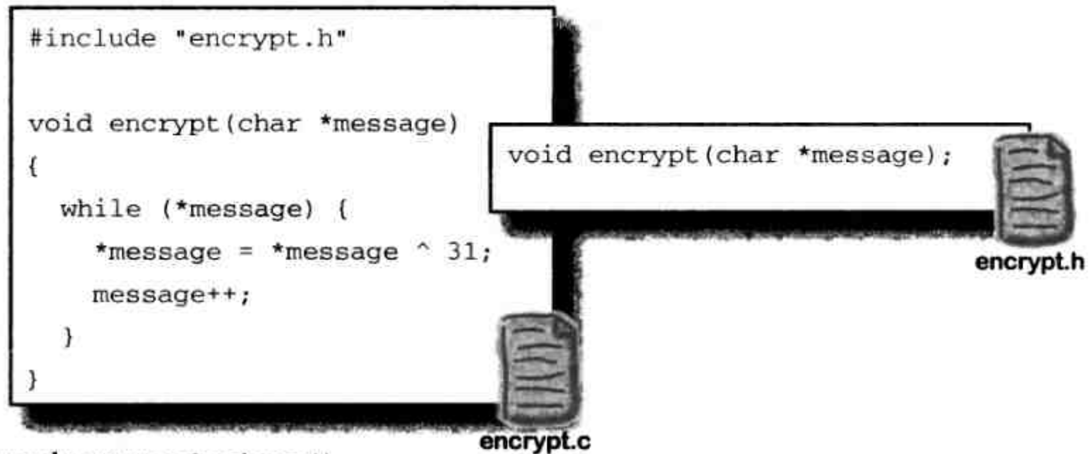
Легко заменяемый код

Вы уже знаете, какой мощью обладают стандартные библиотеки.

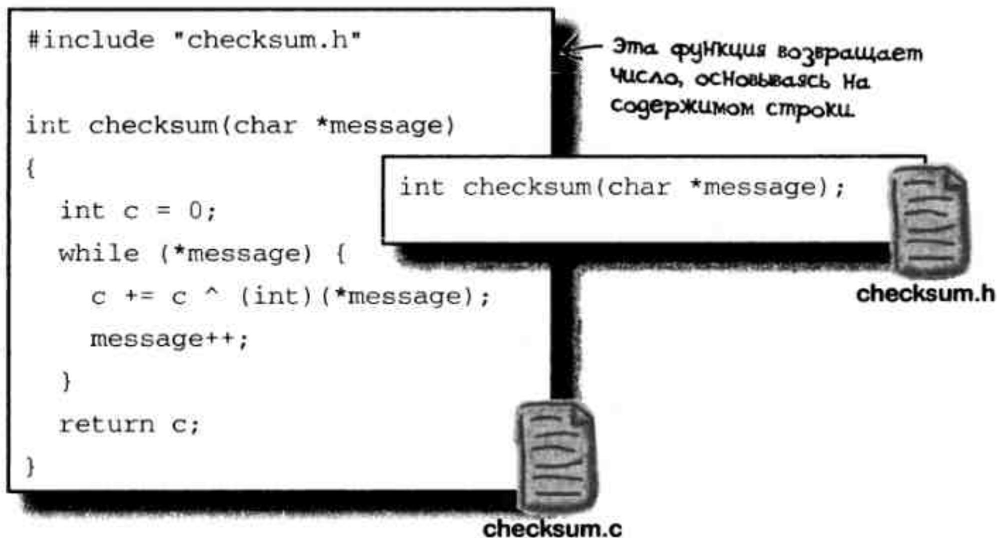
Теперь пришло время применить эту мощь в *собственном* коде. В данной главе вы узнаете, как создавать **собственные библиотеки** и **использовать один и тот же код в нескольких приложениях**. Более того, вы научитесь разделять код во время выполнения программы с помощью **динамических библиотек**. Вам откроются секреты *гуру-программистов*. И дочитав главу, вы сможете писать масштабируемый, простой и эффективный в управлении код.

Код, который вы можете принести с собой в банк

Помните, вы писали функцию encrypt() для шифрования содержимого строки? Она хранилась в отдельном исходном файле, код из которого можно было использовать в нескольких программах:



Кое-кто написал функцию checksum(), которая может проверять, изменилось ли содержимое строки. Шифрование данных и проверка их изменений важны для поддержания **безопасности**. Они полезны и по отдельности, но вместе могли бы составить основу для **библиотеки security** (безопасности).



Библиотека security? О, это именно то, что я искал! Безопасность в нашем банке эээ... слегка хромает.

Глава отдела безопасности в банке Head First. А еще он чистит бассейны

Наточите свой карандаш

Парень из банка написал тестовую программу, чтобы проверить работу этих двух функций. Он поместил весь исходный код в одну директорию на своем компьютере, после чего приступил к компиляции.

Он скомпилировал два исходника в объектные файлы и написал тестовую программу:

```
> gcc -c encrypt.c -o encrypt.o
> gcc -c checksum.c -o checksum.o
>
```

```
#include <stdio.h>
#include <encrypt.h>
#include <checksum.h>

int main()
{
  char s[] = "Скажи друг и проходи";
  encrypt(s);
  printf("Зашифровано в '%s'\n", s);
  printf("Контрольная сумма %i\n", checksum(s));
  encrypt(s);
  printf("Расшифровано обратно в '%s'\n", s);
  printf("Контрольная сумма %i\n", checksum(s));
  return 0;
}
```

Функция encrypt зашифрует ваши данные, а при повторном вызове расшифрует

И вот с этого момента начинаются проблемы. Во время компиляции программы что-то пошло не так...

```
> gcc test_code.c encrypt.o checksum.o -o test_code
test_code.c:2:21: error: encrypt.h: No such file or directory
test_code.c:3:22: error: checksum.h: No such file or directory
>
```

Обведите карандашом команду или код, которые привели к ошибке компиляции.

Наточите свой карандаш

Решение

Проблема кроется в тестовой программе. Все исходные файлы размещены в одной директории, но заголовки *encrypt.h* и *checksum.h* подключаются с помощью **угловых скобок** (< >).

```
#include <stdio.h>
#include <encrypt.h>
#include <checksum.h>

int main()
{
  char s[] = "Скажи друг и проходи";
  encrypt(s);
  printf("Зашифровано в '%s'\n", s);
  printf("Контрольная сумма %i\n", checksum(s));
  encrypt(s);
  printf("Расшифровано обратно в '%s'\n", s);
  printf("Контрольная сумма %i\n", checksum(s));
  return 0;
}
```

Угловые скобки предназначены для стандартных заголовков

Если вы используете угловые скобки в директиве `#include`, компилятор не станет искать заголовочные файлы в *текущем* каталоге. Вместо этого поиск будет производиться в **стандартных** заголовочных директориях.

Чтобы скомпилировать программу с **локальными** заголовочными файлами, необходимо заменить угловые скобки на обычные кавычки (" "):

stdio.h хранится в одной из стандартных заголовочных директорий.

```
#include <stdio.h>
#include "encrypt.h"
#include "checksum.h"
```

encrypt.h и checksum.h находятся в той же директории, что и сама программа.

Теперь код компилируется правильно. Он шифрует тестовую строку во что-то нечитаемое.

```
File Edit Window Help <>
> gcc test_code.c encrypt.o checksum.o -o test_code
> ./test_code
Зашифровано в 'Loz~t?ymvzq{?~q{?zqkzm'
Контрольная сумма -73
Расшифровано обратно в 'Скажи друг и проходи'
Контрольная сумма 56
>
```

Функция checksum возвращает разные значения для разных строк.

Вызвав функцию encrypt() во второй раз, мы получим оригинальную строку.

Где расположены стандартные заголовочные директории?

Так где же компилятор будет искать заголовочные файлы, если вы подключите их с помощью угловых скобок? Загляните в документацию, поставляемую вместе с компилятором. Как правило, на Unix-подобных системах, таких как Mac или Linux, компилятор производит поиск файлов в следующих директориях:

```
/usr/local/include
/usr/include
```

В первую очередь компилятор проверит /usr/local/include.

/usr/local/include часто используется для хранения заголовочных файлов сторонних библиотек.

/usr/include обычно хранит заголовочные файлы операционной системы.

Если же вы используете разновидность компилятора gcc под названием MinGW, поиск будет выполняться так:

```
C:\MinGW\include
```

Но что если вы хотите разделять код?

Возможно, вам захочется написать код, доступный для множества программ, размещенных в разных директориях. Как это реализовать?

Ага, я должен добавить библиотеку security к разным программам. Но я не хочу копировать ее код для каждой из них...

Есть два вида файлов, которые нужно разделять между разными программами: **заголовочные (*.h*)** и **объектные (*.o*)**. Давайте посмотрим, как это делается.

Разделяем заголовочные файлы

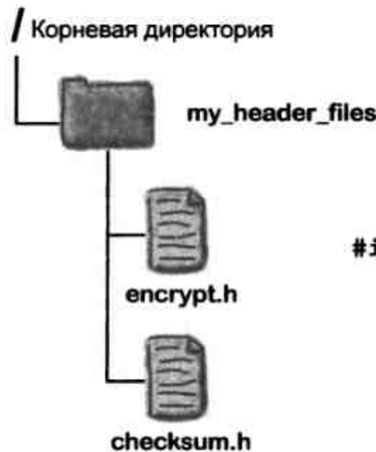
Разделить заголовочные файлы между разными проектами на языке Си можно несколькими способами.

1. **Сохранить их в стандартную директорию.**
   Если вы скопируете свои заголовочные файлы в одну из стандартных директорий, такую как */usr/local/include*, то сможете подключать их в своем коде с помощью угловых скобок.

```
#include <encrypt.h>
```

Если заголовочные файлы находятся в стандартной директории, вы можете использовать угловые скобки.

2. **Указать в директиве include полный путь.**
   Если вы хотите хранить свои файлы в каком-то другом месте, например в ***/my_header_files***, то можете добавить название этой директории в директиву `include`:




```
#include "/my_header_files/encrypt.h"
```

3. **Вы можете сообщить компилятору место поиска.**
   В этом случае вы указываете компилятору, где нужно искать заголовочные файлы. В `gcc` это делается с помощью параметра **`-I`**:

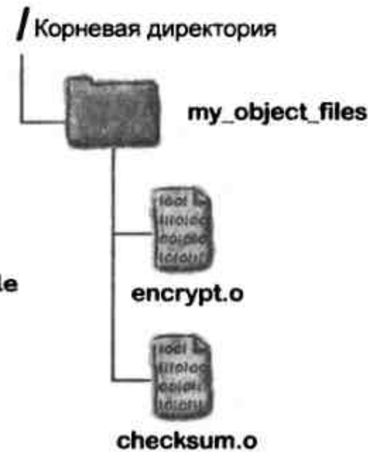
```
gcc -I/my_header_files test_code.c ... -o test_code
```

Это говорит компилятору о том, что он должен искать не только в стандартных директориях, но и внутри /my_header_files.

Параметр `-I` сообщает компилятору `gcc`, что существует еще одно место, где он может найти заголовочные файлы. Он по-прежнему будет выполнять поиск во всех стандартных директориях, но сначала проверит каталоги, указанные параметром `-I`.

Разделяем объектные файлы .o с помощью полного пути

Теперь вы всегда можете размещать свои объектные файлы в некое подобие *общей директории*. После чего останется только указать полный путь к ним при компиляции программы, которая эти файлы использует:




```
gcc -I/my_header_files test_code.c
      /my_object_files/encrypt.o
      /my_object_files/checksum.o -o test_code
```

Используя полный путь к объектным файлам, вы сообщаете компилятору, что вам не нужна отдельная их копия для каждого проекта.

/my_object_files — это центральное хранилище для ваших объектных файлов.

Если вы скомпилируете свой код с указанием *полного пути* к объектным файлам, которые хотите использовать, то *все* ваши программы на Си смогут разделять одни и те же файлы *encrypt.o* и *checksum.o*.

Хм... Если я хочу сделать общими один или два объектных файла, то это нормально. Но что если их уйма? Можно ли как-то сообщить компилятору о множестве объектных файлов?

Да, создав архив объектных файлов, вы сможете сообщить о них компилятору одновременно.

Архив — это всего лишь набор объектного кода, упакованный в один файл. Если вы поместите весь код библиотеки security в один архив, то вам станет значительно проще разделять его между проектами.

Давайте посмотрим, как это делается...

Архив содержит объектные файлы

Вы когда-нибудь сталкивались с файлами типа *.zip* или *.tar*? Если да, то вам должно быть известно, насколько легко создать файл, который содержит *другие* файлы. Это то, чем по сути и является архив *.a* — файл, содержащий другие файлы.

Откройте терминал или командную строку и перейдите в одну из *библиотечных* директорий, которые содержат библиотечный код, например */usr/lib* или *C:\MinGW\lib*. Там вы найдете большой набор архивов *.a*. Чтобы посмотреть их содержимое, воспользуйтесь командой nm:

У вас на компьютере может не быть файлов libl.a, но вы можете опробовать эту команду на любом другом архиве.

Это архив под названием libl.a.

```
File Edit Window Help SilenceInTheLibrary
> nm libl.a

libl.a(libmain.o):
00000000000003a8 s EH_frame0
                 U _exit
0000000000000000 T _main
00000000000003c0 S _main.eh
                 U _yylex

libl.a(libyywrap.o):
0000000000000350 s EH_frame0
0000000000000000 T _yywrap
0000000000000368 S _yywrap.eh
>
```

libmain.o

T _main означает, что libmain.o содержит функцию main().

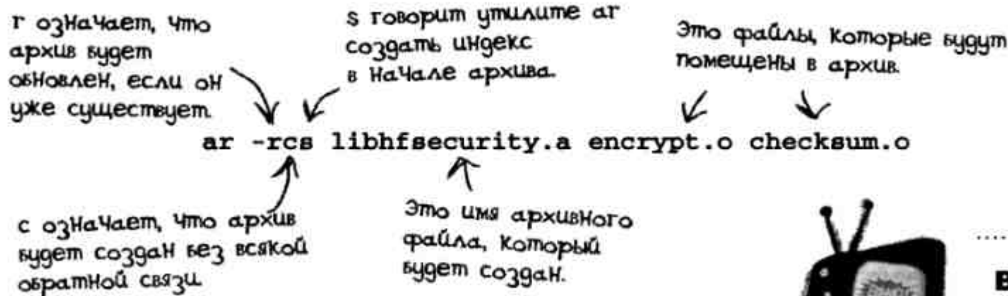
libyywrap.o

Команда nm выводит список **имен**, хранящихся внутри архива. Архив *libl.a*, представленный здесь, содержит два объектных файла: *libmain.o* и *libyywrap.o*. Неважно, какие именно функции они выполняют. Суть в том, что вы можете взять набор объектных файлов и создать единый архив, который можно использовать вместе с gcc.

Но прежде чем вы узнаете, как компилировать программы с помощью архивов, давайте посмотрим, как поместить в архив наши файлы *encrypt.o* и *checksum.o*.

Создайте архив с помощью команды ar...

Команда ar (**archive**) помещает набор объектных файлов внутрь архивного файла:



Будьте осторожны!

Всегда называйте свои архивы по шаблону *lib<что-то>.a*.

Если вы не станете называть их таким образом, компьютеру будет сложно за ними уследить.

Вы заметили, что все архивные файлы имеют названия вида ***lib<что-то>.a***? Это стандартный способ именования архивов. Имя начинается с *lib* (от английского *library* — «библиотека»), потому что это **статическая библиотека**. Позже вы узнаете, что это означает.

...затем сохраните файл .a в библиотечной директории

Получив архив, вы можете поместить его в библиотечную директорию. Но в какую именно? Вариантов несколько — решение за вами.

- **Вы можете поместить свой архив в стандартную директорию, такую как /usr/local/lib.**
  Некоторые программисты, убедившись однажды в работоспособности такого подхода, любят устанавливать архивы в стандартную директорию. Директория */usr/local/lib* является хорошим вариантом при использовании Mac, Linux или Cygwin, потому что специально предназначена для ваших собственных локальных библиотек.

- **Вы можете поместить архивный файл в какую-то другую директорию.**
  Если вы все еще дописываете код или предпочитаете не устанавливать свои файлы в системную директорию, то вы всегда можете создать собственный библиотечный каталог. Например, */my_lib*.

На большинстве компьютеров вам необходимо обладать правами администратора, чтобы размещать файлы внутри /usr/local/lib.

И наконец, скомпилируйте другие свои программы

Архив создавался с одной лишь целью — чтобы вы могли использовать его в других приложениях. Установив архив в стандартную директорию, вы получите возможность компилировать свой код с помощью ключа `-l`:

Не забудьте перечислить исходные файлы до того, как указывать библиотеки с помощью ключа -l

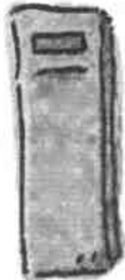
hfsecurity сообщает компилятору, что нужно искать архив с названием libhfsecurity.a

```
gcc test_code.c -lhfsecurity -o test_code
```

Нужен ли вам парметр -I? Все зависит от того, куда вы поместили свои заголовки.

Вы можете использовать несколько параметров -l для нескольких архивов.

Теперь понимаете, почему так важно использовать шаблон *lib<что-то>.a*, когда вы называете библиотеку? Имя, следующее за параметром `-l`, должно совпадать с *частью названия архива*. Таким образом, если ваш архив называется *libawesome.a*, вы можете скомпилировать свою программу с ключом `-lawesome`.



Но что если вы разместите свой архив где-нибудь в другом месте, например в */my-lib*? В таком случае, вам нужно использовать ключ `-L`, чтобы указать директорию для поиска:

```
gcc test_code.c -L/my_lib -lhfsecurity -o test_code
```

Уголок ботана

Содержимое библиотечных директорий может очень сильно отличаться на разных компьютерах. Почему так происходит? Все потому, что у разных операционных систем разные *сервисы*. Каждый файл *.a* является отдельной библиотекой. В этих директориях будут библиотеки для подключения к сети или создания приложений с графическим интерфейсом.

Попробуйте применить команду `nm` к нескольким файлам *.a*. Многие перечисленные имена будут соответствовать функциям, которые вы сможете потом использовать:

```
0000000000000000 T _yywrap
```

T означает Text и указывает на то, что это функция.

Имя функции yywrap().

Команда `nm` выведет названия всех объектных файлов, после чего перечислит имена, которые они содержат. Буква `T` рядом с названием указывает на имя функции в объектном файле.

Магнитики с кодом

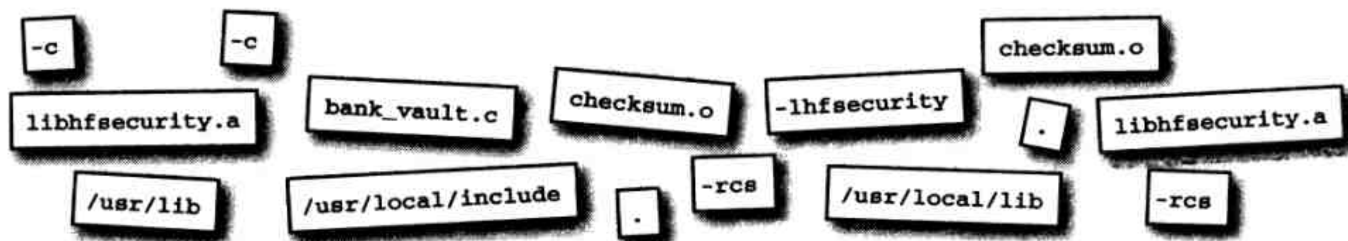
У парня, который отвечает за безопасность, возникли проблемы при компиляции одной из банковских программ с новой библиотекой security. Написанный им код лежит в одной директории вместе с исходниками `encrypt` и `checksum`. Он хочет создать в той же директории архив *libhfsecurity.a*, чтобы использовать его при компиляции своей программы. Поможете ему исправить makefile?

Примечание: в программе `bank_vault` используются следующие директивы `#include`:

```
#include <encrypt.h>
#include <checksum.h>
```

Это makefile:

```
encrypt.o: encrypt.c

      gcc ........................ encrypt.c -o encrypt.o

checksum.o: checksum.c

      gcc ........................ checksum.c -o checksum.o


libhfsecurity.a: encrypt.o ........................

      ar -rcs ........................ encrypt.o ........................


bank_vault: bank_vault.c ........................

      gcc ........................ -I ............ -L ............ ........................ -o bank_vault
```



Магнитики с кодом Решение

У парня, который отвечает за безопасность, возникли проблемы при компиляции одной из банковских программ с новой библиотекой security. Написанный им код лежит в одной директории вместе с исходниками `encrypt` и `checksum`. Он хочет создать в той же директории архив *libhfsecurity.a*, чтобы использовать его при компиляции своей программы. Вы должны были помочь ему исправить makefile.

Примечание: в программе `bank_vault` используются следующие директивы `#include`:

```
#include <encrypt.h>
#include <checksum.h>
```

В директиве #include используются угловые скобки. С помощью параметра -I компилятору нужно сообщить, где находятся заголовочные файлы.

Это makefile:

```
encrypt.o: encrypt.c
      gcc -c encrypt.c -o encrypt.o

checksum.o: checksum.c
      gcc -c checksum.c -o checksum.o

libhfsecurity.a: encrypt.o checksum.o
      ar -rcs libhfsecurity.a encrypt.o checksum.o

bank_vault: bank_vault.c libhfsecurity.a
      gcc bank_vault.c -I . -L . -lhfsecurity -o bank_vault
```

Здесь создается объектный файл из исходного файла encrypt.c.

Здесь — из исходного файла checksum.c.

Вы не можете собрать архив libhfsecurity.a, пока не создадите файлы encrypt.o и checksum.o.

Здесь создается архив libhfsecurity.a.

Вам нужно указать -lhfsecurity, потому что архив называется libhfsecurity.a.

Исходный код программы нужно указывать перед библиотечным кодом.

Здесь нужен параметр -I, потому что заголовочные файлы находятся в текущей директории («.»).

Здесь нужен параметр -L, потому что архив находится в текущей директории.

Неиспользованные магнитики: `/usr/lib` `/usr/local/include` `-rcs` `/usr/local/lib` `-rcs`

КЛЮЧЕВЫЕ МОМЕНТЫ

- Заголовки в угловых скобках (< >) берутся из стандартных директорий.
- Примерами стандартных директорий могут служить */usr/include* и *C:\MinGW\include*.
- Библиотечный архив содержит несколько объектных файлов.
- Вы можете создать архив командой `ar -rcs libarchive.a file0.o file1.o....`
- Имена библиотечных архивов должны начинаться с *lib* и заканчиваться на *.a*.
- Если вам нужно скомпоновать архив с именем *libfred.a*, используйте `-lfred`.
- Ключ `-L` в команде `gcc` должен следовать *за* исходными файлами.

не бывает

Глупых Вопросов

В: Как узнать, где находятся стандартные библиотечные директории на моем компьютере?

О: Вам нужно заглянуть в документацию своего компилятора. На большинстве Unix-подобных систем в число библиотечных директорий входят */usr/lib* и */usr/local/lib*.

В: Почему у меня не получается поместить библиотечный архив в директорию */usr/lib*?

О: Почти наверняка это вопрос безопасности. Многие операционные системы не позволяют записывать файлы в стандартные библиотечные директории, потому что вы случайно можете затереть одну из существующих библиотек.

В: Все операционные системы имеют один и тот же формат архивов?

О: Нет. На разных платформах формат архивов может слегка отличаться, а объектный код, содержащийся в архивах, будет совершенно разным.

В: Могу ли я посмотреть содержимое созданного мной библиотечного архива?

О: Да. Команда `-t <имя_файла>` выведет содержимое архива.

В: Скомпонованы ли объектные файлы внутри архива, как в случае с исполняемым файлом?

О: Нет. Объектные файлы в архиве хранятся отдельно.

В: Могу ли я поместить в архив файл любого типа?

О: Нет. Команда `ar` проверяет тип файла, прежде чем включить его в архив.

В: Могу ли я извлечь из архива какой-то один объектный файл?

О: Да. Чтобы извлечь файл *encrypt.o* из *libhfsecurity.a*, используйте команду `ar -x libhfsecurity.a encrypt.o`.

В: Почему это называется статической компоновкой?

О: Потому что ее нельзя изменить, если она уже была произведена. Скомпоновать два файла статически — это как смешать кофе с молоком: после этого их уже нельзя разделить.

В: Должен ли я использовать библиотеку hfsecurity, чтобы обезопасить данные в моем банке?

О: Это, наверное, не очень хорошая идея.

Откровения компоновщика

Интервью этой недели:
Чем же именно ты занимаешься?

Head First: Спасибо тебе, Компоновщик, что уделил нам сегодня время.

Компоновщик: С радостью.

Head First: Для начала хотелось бы спросить, чувствовал ли ты когда-нибудь недостаток внимания к себе со стороны разработчиков? Возможно, они не до конца понимают, чем именно ты занимаешься?

Компоновщик: Я очень тихий и скромный. Люди редко общаются со мной напрямую с помощью команды `ld`.

Head First: `ld`?

Компоновщик: Да? Вот видите, это я и есть.

Head First: У меня на экране вывелось множество параметров.

Компоновщик: Вот именно. У меня много параметров. Много способов объединять программы. Поэтому некоторые люди просто используют команду `gcc`.

Head First: Выходит, компилятор может компоновать файлы?

Компоновщик: Компилятор описывает действия, которые нужно выполнить для объединения файлов, и затем зовет меня. А я уже все делаю. Тихо и незаметно. Вы никогда не узнаете, что я там находился.

Head First: У меня еще один вопрос…

Компоновщик: Да?

Head First: Не хотелось бы показаться глупым, но что именно ты делаешь?

Компоновщик: Это не глупый вопрос. Я сшиваю вместе скомпилированные фрагменты кода, словно телефонный оператор.

Head First: Не понимаю.

Компоновщик: Когда-то телефонные операторы при звонке соединяли кабелем один телефонный номер с другим, чтобы люди на обоих концах провода могли поговорить. То же самое происходит с объектными файлами.

Head First: Как так?

Компоновщик: Объектному файлу может понадобиться вызвать функцию, хранящуюся в каком-то другом файле. Я создаю связь между тем местом, где функция была вызвана, и тем, где она находится.

Head First: У тебя, наверное, много терпения.

Компоновщик: Мне нравится заниматься такими вещами. В свободное время я плету кружева.

Head First: Правда?

Компоновщик: Нет.

Head First: Спасибо тебе, Компоновщик.

Тренажерный зал Head First выходит на международный уровень

Парни из тренажерного зала Head First хотят выйти на **международный уровень**. Они собираются открывать торговые точки на четырех континентах и продавать спортивные тренажеры под брендом «*Кровь, пот и снаряжение*»™. Сейчас они пишут программное обеспечение для своих беговых дорожек, эллиптических и велотренажеров. Код будет получать данные с датчиков, установленных на тренажерах, и выводить информацию на небольшом экране. Благодаря этому пользователи смогут увидеть, какое расстояние они преодолели и сколько калорий сожгли.

План есть план, но нашим ребятам нужна небольшая помощь. Давайте рассмотрим код более подробно.

Подсчет калорий

Команда все еще работает над программой, но один из *ключевых модулей* уже готов. Библиотека *hfcal* сгенерирует основные данные для вывода на экран. Код будет подавать на стандартный вывод вес пользователя, пройденную им виртуальную дистанцию и специальный сгенерированный *коэффициент*:

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Вес: %3.2f фунта\n", weight);
  printf("Расстояние: %3.2f мили\n", distance);
  printf("Сожжено калорий: %4.2f кал\n", coeff * weight * distance);
}
```

Заголовочный файл hfcal.h всего лишь содержит объявление функции display_calories().

Вес в фунтах.

Расстояние в милях.

hfcal.c

Этот код будет храниться в файле hfcal.c.

Главный код для каждого тренажера еще не написан. Ребята планируют сделать отдельные программы для беговых дорожек, эллиптических и велотренажеров. Но пока что они создали *тестовую программу*, которая будет вызывать код из *hfcal.c* и передавать ему проверочные данные.

```
#include <stdio.h>
#include <hfcal.h>

int main()
{
  display_calories(115.2, 11.3, 0.79);
  return 0;
}
```

Тестовый пользователь имеет вес 115.2 фунта. Он преодолел 11.3 мили на эллиптическом тренажере.

Для этого компьютера коэффициент равен 0.79.

ellipticals.c

Это тестовый код.

Стандартный вывод будет направлен на экран.

```
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
```

Вот что выводит на экран тестовая программа.

Наточите свой карандаш

Теперь, когда вы увидели исходный код тестовой программы и библиотеки *hfcal*, пришло время его собрать.

Посмотрим, насколько хорошо вы запомнили команды.

1. Начните с создания объектного файла под названием *hfcal.o*. Заголовок *hfcal.h* будет сохранен в *./includes*:

2. На следующем этапе вам нужно получить из тестовой программы *elliptical.c* объектный файл с именем *elliptical.o*:

3. Теперь вам необходимо создать архивную библиотеку из файла *hfcal.o* и поместить ее в *./libs*:

4. И наконец, получите исполняемый файл `elliptical`, используя *elliptical.o* и архив *hfcal*:

Наточите свой карандаш
Решение

Теперь, когда вы увидели исходный код тестовой программы и библиотеки *hfcal*, пришло время его собрать.

Посмотрим, насколько хорошо вы запомнили команды.

1. Начните с создания объектного файла под названием *hfcal.o*. Заголовок *hfcal.h* будет сохранен в *./includes*:

Программе hfcalc нужно знать, где находится заголовочный файл.

```
gcc -I./includes -c hfcal.c -o hfcal.o
```

Не забыли добавить ключ -I?

-c означает «просто создай объектный файл, не компонуй его».

2. На следующем этапе вам нужно получить из тестовой программы *elliptical.c* объектный файл с именем *elliptical.o*:

```
gcc -I./includes -c elliptical.c -o elliptical.o
```

Вам нужно сообщить компилятору, что заголовки находятся в ./includes.

3. Теперь вам необходимо создать архивную библиотеку из файла *hfcal.o* и поместить ее в *./libs*:

Имя библиотеки должно иметь вид lib...a.

```
ar -rcs ./libs/libhfcal.a hfcal.o
```

Архив нужно сохранить в директорию ./libs.

4. И наконец, получите исполняемый файл `elliptical`, используя *elliptical.o* и архив *hfcal*:

Флаг -lhfcal говорит компилятору, что нужно искать libhfcal.a.

```
gcc elliptical.o -L./libs -lhfcal -o elliptical
```

Вы собираете программу с помощью elliptical.o и библиотеки.

-L/libs говорит компилятору, где хранится библиотека.

Собрав программу elliptical, вы можете запустить ее в консоли:

```
File Edit Window Help SilenceInTheLibrary
> ./elliptical
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
>
```

Но все немного сложнее...

Оказывается, есть небольшая проблема. Наши тренажерные залы будут разбросаны *повсюду* — по разным странам, где используются разные языки и системы измерения. Например, в Великобритании компьютеры должны выдавать информацию в **килограммах** и **километрах**:

В тренажерных залах находится множество различного оборудования. Если наши залы расположены в 50 странах и в каждом из них по 20 видов тренажеров, то нужно будет написать ***1000*** разных версий программ. Это очень *много*.

Кроме того, есть и другие проблемы.

- Если инженер заменит датчик на тренажере, придется менять весь код, который с этим датчиком взаимодействует.
- При замене экранов нужно будет менять код, который генерирует вывод.
- И еще много других вариантов.

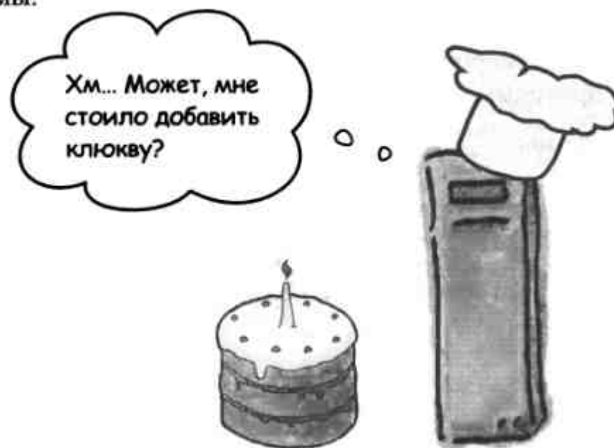
Если подумать, то подобные проблемы могут возникнуть при написании любого программного обеспечения. Разные компьютеры могут взаимодействовать с разными *драйверами устройств*, обращаться к разным *базам данных* или *графическим пользовательским интерфейсам*. Вероятно, вам не удастся собрать версию кода, которая будет работать на любом компьютере. Так что же делать?

Программы состоят из множества частей...

Вы уже знаете, что программы можно собирать с использованием разных фрагментов **объектного кода**. Вы создавали объектные файлы (*.o*) и архивы (*.a*), после чего компоновали их в единый исполняемый файл.

...но вы не можете их изменять после того, как они были скомпонованы

Проблема в том, что при сборке подобной программы все ее составляющие **статические**. После того как вы создадите единый исполняемый файл из нескольких фрагментов объектного кода, у вас больше *не будет возможности* менять составляющие без повторной сборки всей программы.

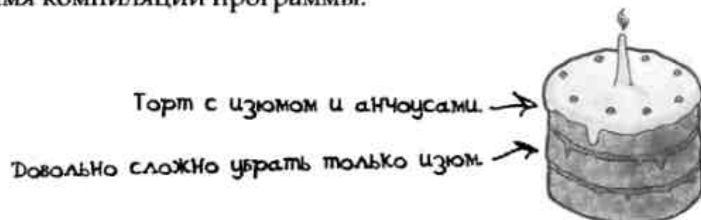


Программа — это просто большой кусок объектного кода. Нельзя разделить ее на отдельные части, которые отвечают за **вывод** и **работу с датчиками**, — теперь это смесь.

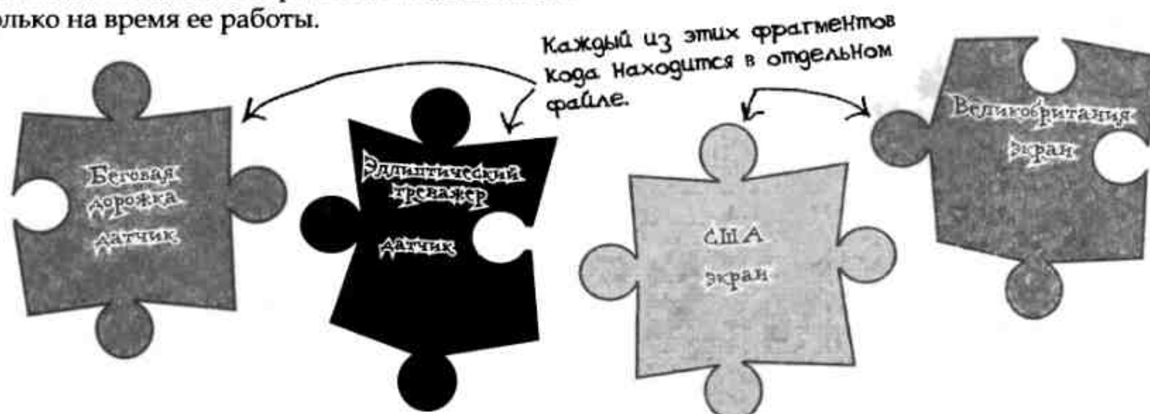
Ах, если бы можно было запускать программы, используя взаимозаменяемые фрагменты объектного кода. Но, наверное, это только мечты...

Динамическая компоновка происходит во время выполнения программы

Вы не можете изменять разные части объектного кода в исполняемом файле, потому что они… ну, потому что они все хранятся в одном файле. Они были **скомпонованы статически** во время компиляции программы.



Но вы могли бы избежать подобных проблем, если бы ваша программа состояла не из одного, а из нескольких отдельных файлов, объединяемых только на время ее работы.



Теперь остается найти способ хранить фрагменты объектного кода в отдельных файлах, ***компонуя их вместе динамически*** на время работы программы.

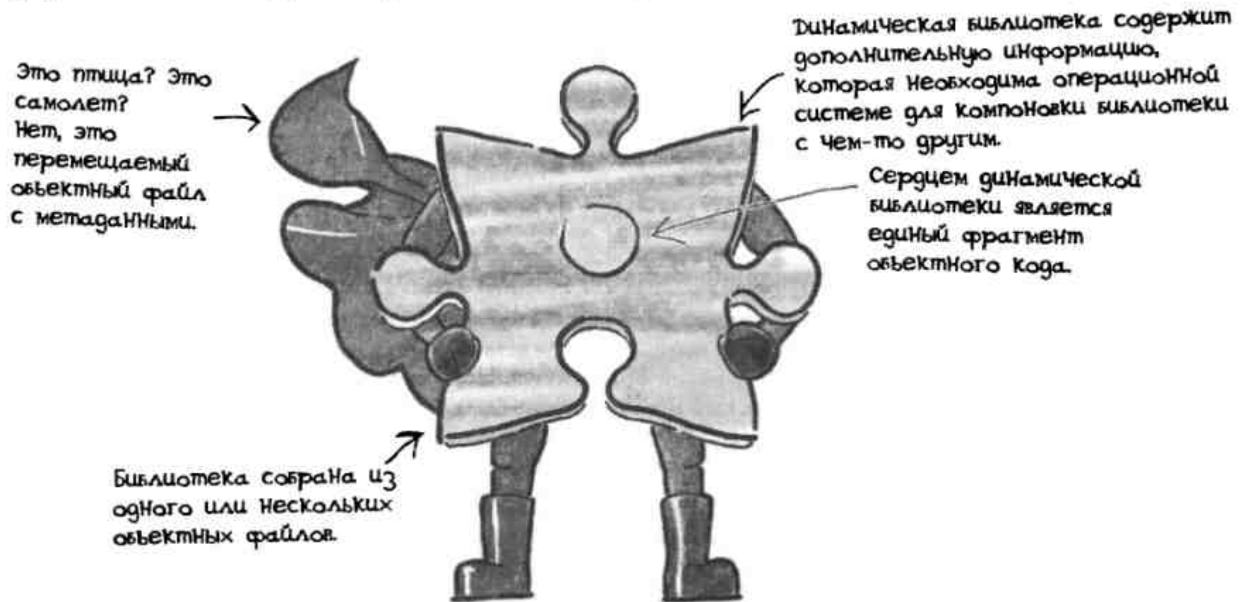
Можно ли скомпоновать архив во время выполнения программы?

Итак, вам нужны отдельные файлы, содержащие разные фрагменты объектного кода. Но ведь они уже есть: это объектные файлы (*.o*) и архивы (*.a*). Означает ли это, что достаточно попросить компьютер не компоновать их, пока вы не запустите программу?

К сожалению, не все так просто. У обычных объектных файлов и архивов недостаточно информации для компоновки во время выполнения. *Динамическим библиотекам* нужны и другие данные, например имена файлов, с которыми они должны компоноваться.

Динамические библиотеки — это объектные файлы на стероидах

Итак, динамические библиотеки *похожи* на те объектные файлы, которые вы создавали в течение некоторого времени, но это не совсем одно и то же. Как и в случае с архивами, динамические библиотеки могут быть собраны из нескольких объектных файлов. Разница заключается в том, что внутри динамической библиотеки эти файлы специальным образом скомпонованы и представляют собой единый кусок объектного кода.



Так как же создаются динамические библиотеки? Давайте разберемся.

Сначала создадим объектный файл

Если вы собираетесь превратить код из файла *hfcal.c* в динамическую библиотеку, первым делом его нужно скомпилировать в объектный файл:

```
gcc -I/includes -fPIC -c hfcal.c -o hfcal.o
```

-c означает «не компоновать код».

Заголовок hfcal.h находится в /includes.

Что означает -fPIC?

Заметили разницу? Как и прежде, вы создаете *hfcal.o*, но теперь добавляется новый ключ **-fPIC**. Это говорит gcc о том, что вы хотите создать **позиционно-независимый код**. Для некоторых процессоров и операционных систем библиотеки должны собираться из позиционно-независимого кода, чтобы при выполнении программы можно было выбрать то место в памяти, куда их нужно загружать.

На практике для *большинства* систем этот параметр указывать не нужно. Попробуйте использовать его на своем компьютере. Он не причинит никого вреда, даже если окажется бесполезным.

Позиционно-независимый код можно перемещать в памяти.

Уголок ботана

Так что же такое **позиционно-независимый код**?

Позиционно-независимым называют код, которому все равно, в какую область памяти его загрузит компьютер. Представьте, что динамическая библиотека ожидает найти некий фрагмент глобальных данных на расстоянии 500 байтов от того места, куда она была загружена. Если операционная система решит загрузить эту библиотеку в какое-нибудь другое место, возникнет проблема. Этого можно избежать, если сказать компилятору, чтобы он создал позиционно-независимый код.

Некоторые операционные системы, такие как Windows, при загрузке динамических библиотек используют технический прием под названием **распределение памяти** — это когда весь код позиционируется эффективно и независимо. Компилируя свой код под Windows, вы, должно быть, заметили, что gcc предупреждает о ненужности ключа -fPIC. Вы можете проигнорировать это предупреждение или убрать ключ. В любом случае с вашим кодом ничего не случится.

На разных платформах динамические библиотеки называются по-разному

Динамические библиотеки доступны на большинстве операционных систем, и принцип их работы в значительной степени одинаков. Однако называются они по-разному. В Windows это **библиотеки динамической компоновки** (dynamic link library) с расширением ***.dll***, в Linux и Unix — **библиотеки общего пользования** (shared object files) с расширением *.so*, в Mac — просто **динамические библиотеки** (dynamic libraries) с расширением ***.dylib***.
Но, несмотря на то что эти файлы имеют разные расширения, вы можете создавать их похожим образом:

```
                           C:\libs\hfcal.dll        <- MinGW на Windows
gcc -shared hfcal.o -o     /libs/libhfcal.dll.a     <- Cygwin на Windows
                           /libs/libhfcal.so        <- Linux или Unix
                           /libs/libhfcal.dylib     <- Mac
```

Параметр `-shared` сообщает `gcc`, что вы хотите конвертировать объектный файл (*.o*) в динамическую библиотеку. При создании динамической библиотеки компилятор помещает ее имя внутрь файла. Таким образом, если вы сгенерируете в Linux библиотеку под названием *libhfcal.so*, в файле *libhfcal.so* будет записано, что библиотека называется *hfcal*. Именно поэтому вы не можете переименовывать файл библиотеки после того, как он был скомпилирован.

Если вам нужно переименовать библиотеку, перекомпилируйте ее с новым именем.

Будьте осторожны!

На некоторых старых версиях Mac параметр -shared недоступен.

Но не переживайте, на таких системах достаточно заменить его на `-dynamiclib` *– и все будет работать точно так же.*

Компиляция программы elliptical

Вы можете использовать динамическую библиотеку так же, как и статическую. Собрать программу `elliptical` можно следующим образом:

```
gcc -I\include -c elliptical.c -o elliptical.o
gcc elliptical.o -L\libs -lhfcal -o elliptical
```

Несмотря на то что вы использовали бы те же команды, будь архив *hfcal* статическим, процесс компиляции будет происходить иначе. Поскольку библиотека динамическая, компилятор не включит ее код в исполняемый файл. Вместо этого он вставит некоторый код-заполнитель, который будет искать библиотеку и выполнять компоновку во время работы приложения.

Теперь давайте посмотрим, работает ли наша программа.

Имена библиотек в MinGW и Cygwin

И MinGW, и Cygwin поддерживают несколько форматов имен для динамических библиотек. Библиотека *hfcal* может иметь любое из следующих названий:

libhfcal.dll.a

libhfcal.dll

hfcal.dll

Тест-драйв

Вы создали динамическую библиотеку в директории */libs* и собрали тестовую программу elliptical. Теперь вам нужно эту программу запустить. Поскольку *hfcal* не находится ни в одной из стандартных директорий, вы должны убедиться, что при запуске программы компьютер сможет найти библиотеку.

В Mac

В Mac вы можете просто запустить программу, так как при компиляции в этой системе полный путь к библиотеке */libs/libhfcal.dylib* записывается внутрь исполняемого файла. Поэтому при запуске программа знает, где именно искать библиотеку.

```
File Edit Window Help I'mAMac
> ./elliptical
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
>
```

← Mac

В Linux

В Linux все происходит немного иначе.

Как и в большинстве версий Unix, в Linux компилятор записывает только имя файла *libhfcal.so*, *не включая* путь к нему. Следовательно, если библиотека хранится за пределами стандартных библиотечных директорий (таких как */usr/lib*), программа никак не сможет ее найти. Чтобы обойти это ограничение, Linux проверяет дополнительные директории, описанные в переменной `LD_LIBRARY_PATH`. Если вы **экспортируете** `LD_LIBRARY_PATH` и убедитесь в том, что ваша библиотечная директория там числится, тогда программа `elliptical` найдет *libhfcal.so*.

Вам нужно убедиться, что переменная экспортирована.

В Linux вам необходимо задать переменную LD_LIBRARY_PATH, чтобы программа смогла найти библиотеку. →

```
File Edit Window Help I'mLinux
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/libs
> ./elliptical
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
>
```

↑ Нет необходимости это делать, если библиотека находится в одной из стандартных директорий, таких как /usr/lib.

↑ Linux

В Windows

Теперь давайте рассмотрим запуск кода, который был скомпилирован с помощью таких версий gcc, как Cygwin и MinGW. Оба эти компилятора создадут библиотеку DLL и исполняемый файл. Как и в Linux, в Windows приложение будет хранить только имя библиотеки *hfcal без* директории, где они находятся.

Но Windows не использует переменную LD_LIBRARY_PATH, чтобы отслеживать местоположение библиотек. Вместо этого программа ищет библиотеку в текущей директории, а если не находит — в директориях, перечисленных в переменной PATH.

Используя Cygwin

Скомпилировав программу с использованием Cygwin, вы можете запустить ее из *интерпретатора bash* следующим образом:

```
File Edit Window Help I'mCygwin
> PATH="$PATH:/libs"
> ./elliptical
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
>
```

Windows с использованием Cygwin

Используя MinGW

Если вы скомпилировали программу с помощью MinGW, вы можете запустить ее из *командной строки*:

```
File Edit Window Help I'mMinGW
C:\code> PATH="%PATH%:C:\libs"
C:\code> ./elliptical
Вес: 115.20 фунта
Расстояние: 11.30 мили
Сожжено калорий: 1028.39 кал
C:\code>
```

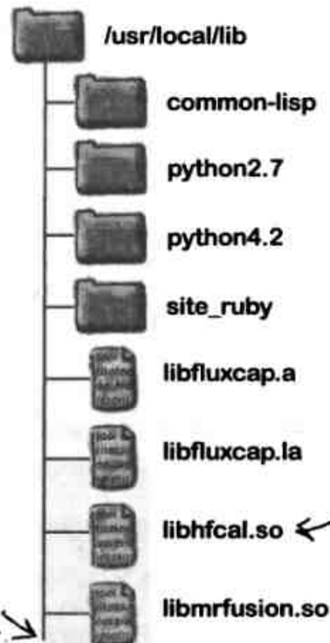
Windows с использованием MinGW

Вам кажется это немного сложным? Так и есть. Поэтому большинство программ, использующих динамические библиотеки, сохраняют их в стандартные директории. Таким образом, в Linux и Mac они обычно находятся в */usr/lib* или */usr/local/lib*. В Windows же разработчики, как правило, хранят свои *.DLL* в одном каталоге с исполняемым файлом.

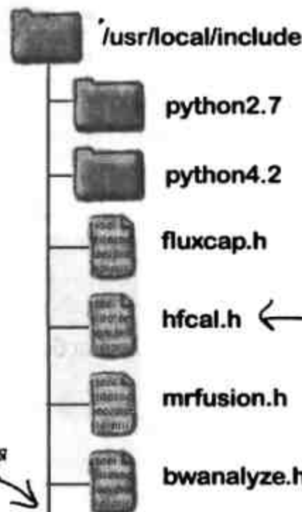
Большое упражнение

Ребята из тренажерного зала Head First собираются поставлять беговые дорожки в Великобританию. Встроенный сервер работает под управлением Linux, и на нем уже установлен код, адаптированный для США.

Технические специалисты установили библиотеку в */usr/local/lib*.



На этом компьютере также присутствует заголовочный файл для библиотеки *hfcal*, который установлен в */usr/local/include*:



Технические специалисты любят устанавливать библиотеки в эти директории, потому что это более-менее стандартный подход. Компьютер полностью настроен для работы в США, но некоторые вещи нужно поменять.

Систему нужно обновить, чтобы ее можно было использовать в тренажерных залах Великобритании. Это значит, что для вывода информации на экран беговой дорожки в коде необходимо поменять мили и фунты на километры и килограммы.

Это код для британского спортзала.

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Вес: %3.2f кг\n", weight);
  printf("Расстояние: %3.2f км\n", distance);
  printf("Сожжено калорий: %4.2f кал\n", coeff * weight * distance);
}
```

Это код для вывода информации в километрах и килограммах.

hfcal_UK.c

Этот файл находится в директории /home/ebrown.

Программное обеспечение, которое уже установлено на компьютере, должно использовать новую версию кода. И поскольку этот код связан с приложениями динамически, вам нужно всего лишь скомпилировать его в директорию */usr/local/lib*.

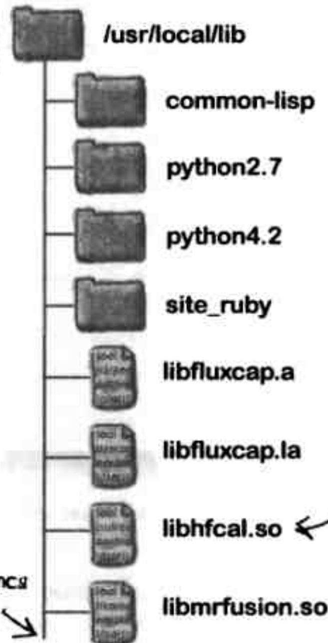
Какую команду вам нужно набрать, чтобы скомпилировать новую версию библиотеки, если предположить, что вы уже находитесь в одном каталоге с *hfcal_UK* и у вас есть права на запись во всех директориях?

Что нужно набрать, чтобы запустить главную программу для беговой дорожки, если она называется */opt/apps/treadmill*?

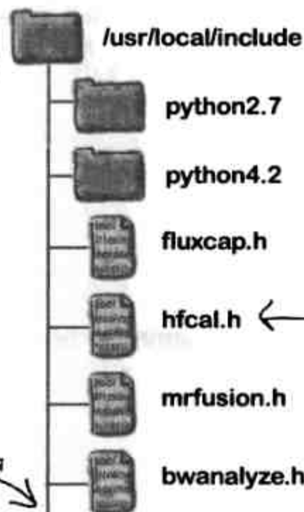
Большое упражнение
Решение

Ребята из тренажерного зала Head First собираются поставлять беговые дорожки в Великобританию. Встроенный сервер работает под управлением Linux, и на нем уже установлен код, адаптированный для США.

Технические специалисты установили библиотеку в */usr/local/lib*.



На этом компьютере также присутствует заголовочный файл для библиотеки *hfcal*, который установлен в */usr/local/include:*



Технические специалисты любят устанавливать библиотеки в эти директории, потому что это более-менее стандартный подход. Компьютер полностью настроен для работы в США, но некоторые вещи нужно поменять.

Систему нужно обновить, чтобы ее можно было использовать в тренажерных залах Великобритании. Это значит, что для вывода информации на экран беговой дорожки в коде необходимо поменять мили и фунты на километры и килограммы.

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Вес: %3.2f кг\n", weight);
  printf("Расстояние: %3.2f км\n", distance);
  printf("Сожжено калорий: %4.2f кал\n", coeff * weight * distance);
}
```

hfcal_UK.c

Программное обеспечение, которое уже установлено на компьютере, должно использовать новую версию кода. И поскольку этот код связан с приложениями динамически, вам нужно всего лишь скомпилировать его в директорию */usr/local/lib*.

Какую команду вам нужно набрать, чтобы скомпилировать новую версию библиотеки, если предположить, что вы уже находитесь в одном каталоге с *hfcal_UK* и у вас есть права на запись во всех директориях?

Нужно скомпилировать исходный код в объектный файл.

```
gcc -c -fPIC hfcal_UK.c -o hfcal.o
gcc -shared hfcal.o -o /usr/local/lib/libhfcal.so
```

Затем нужно преобразовать объектный файл в библиотеку общего пользования.

Вам не нужно указывать параметр -I, потому что заголовочный файл находится в стандартной директории.

Что нужно набрать, чтобы запустить главную программу для беговой дорожки, если она называется */opt/apps/treadmill*?

```
/opt/apps/treadmill
```

Нет необходимости задавать переменную LD_LIBRARY_PATH, потому что библиотека находится в стандартной директории.

Вы заметили, что библиотека и заголовки были установлены в стандартные директории? Это значит, что при компиляции кода вам не нужно использовать ключ -I. Также нет необходимости задавать переменную LD_LIBRARY_PATH при запуске программы.

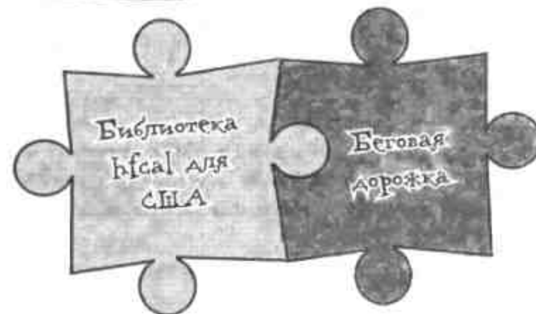
Тест-драйв

Теперь, когда вы обновили библиотеку на беговой дорожке для Великобритании, давайте попробуем в деле **американский** тренажер. Это одна из беговых дорожек для США, в которых используется оригинальная версия библиотеки *libhfcal.so*:

Это американская беговая дорожка.

Приложение `treadmill` запускается при загрузке компьютера. Если поработать на тренажере некоторое время, на экране появится следующее:

Программа `treadmill` на американском тренажере динамически скомпонована с библиотекой *libhfcal.so*, которая была скомпилирована для американской версии программы `hfcal`.



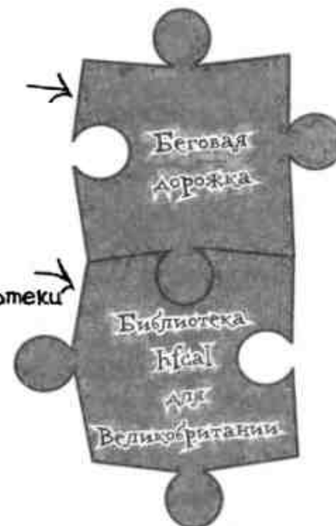
Но что насчет программы treadmill для Великобритании?

На тренажере для Великобритании установлена та же самая программа `treadmill`, но для него библиотека *libhfcal.so* была перекомпилирована с использованием исходного кода из файла *hfcal_UK.c*.

Это беговая дорожка для Великобритании

Это точно такая же программа treadmill

Эта версия скомпонована с британской версией библиотеки hfcal



Если пробежать на этой беговой дорожке ту же дистанцию, на экране появится следующее:

Вес выражается в килограммах.

Дистанция посчитана в километрах.

Калории отображаются в тех же единицах.

Сработало.

Несмотря на то что программа `treadmill` не перекомпилировалась, она смогла **динамически** подхватить код из новой библиотеки.

Динамические библиотеки упрощают замену кода **во время выполнения программы**. Вы можете обновить приложение без перекомпиляции. Если у вас есть несколько программ с общим фрагментом кода, вы можете *обновить их все одновременно*. Теперь, научившись создавать динамические библиотеки, вы стали значительно более сильным разработчиком на языке Си.

Задушевные беседы

В сегодняшнем выпуске: **Два знатных сторонника модульного программного обеспечения обсуждают преимущества и недостатки статической и динамической компоновки.**

Статика: Что ж, мне кажется, мы оба согласны с тем, что создание кода в виде небольших модулей — это хорошая идея.

Динамика: Безусловно.

Статика: Это ведь так разумно, не правда ли?

Динамика: Да.

Статика: Это позволяет поддерживать управление кодом.

Динамика: Да.

Статика: Славные большие программы.

Динамика: Большие?

Статика: Ну да. Славные БОЛЬШИЕ программы с решенными зависимостями.

Динамика: Звучит не очень хорошо.

Статика: О чем ты, старина?

Динамика: Мне кажется, программы должны состоять из множества маленьких файлов, которые компонуются только на время работы программы.

Статика: Ну… <смеется>… это очень… нет, ты серьезно?

Динамика: Абсолютно серьезно.

Статика: Что? Множество отдельных файлов? Соединенных *как попало*?!

Динамика: Я предпочитаю термин *динамически*.

Статика: Но это же… это же… прямой путь к ***хаосу***!

Динамика: Это значит, что позже я могу передумать.

Статика: Ты должен с самого начала сделать все правильно.

Динамика: Но это не всегда возможно. Любая крупная программа должна использовать динамическую компоновку.

Статика:

Любая?

Динамика:

Да, я так считаю.

Статика:

А что насчет ядра Linux, а? Оно достаточно крупное? И, кажется, оно…

Динамика:

…скомпоновано статически. Да, я знаю. Это довод в твою пользу.

Статика:

Может быть, статическая компоновка не такая *естественная* и не дает *столько свободы*, но знаешь что? Статическую программу легко использовать. Это единый исполняемый файл. Хочешь установить — просто скопируй его. Никаких тебе конфликтов с разными версиями библиотек.

Динамика:

Мы просто должны сойтись на том, что у нас разные взгляды.

Статика:

Я не смогу тебя переубедить?

То есть, ты хочешь сказать, что твое мнение скомпоновано статически?

Динамика:

Нет.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Динамические библиотеки компонуются с программами во время их выполнения.
- Динамические библиотеки создаются из одного или нескольких объектных файлов.
- На некоторых компьютерах их нужно компилировать с параметром `-fPIC`.
- Параметр `-fPIC` делает код позиционно-независимым.
- Многие системы позволяют опускать параметр `-fPIC`.
- С указанием параметра `-shared` компилятор создает динамическую библиотеку.
- На разных системах динамические библиотеки называются по-разному.
- Вы упростите себе жизнь, если будете хранить библиотеки в стандартных директориях.
- В противном случае вам может понадобиться задать переменные `PATH` и `LD_LIBRARY_PATH`.

не бывает Глупых Вопросов

В: Почему динамические библиотеки настолько отличаются на разных операционных системах?

О: Операционные системы любят оптимизировать способ загрузки динамических библиотек, поэтому со временем у них выработались разные требования к самим библиотекам.

В: Я попытался переименовать файл, чтобы изменить имя своей библиотеки, но компилятор теперь не может ее найти. Почему?

О: Когда компилятор создает динамическую библиотеку, он записывает ее имя внутрь библиотечного файла. Если вы переименуете файл, его имя перестанет совпадать с именем библиотеки, что приведет к путанице. Если вы хотите изменить название библиотеки, вам придется ее перекомпилировать.

В: Почему Cygwin поддерживает так много разных соглашений об именовании динамических библиотек?

О: Cygwin упрощает процесс компиляции Unix-программ на компьютерах с Windows. Этот компилятор перенял множество соглашений, принятых в Unix, так как он сам формирует Unix-подобную среду. Поэтому он предпочитает давать библиотекам расширение *.a*, даже если это DLL.

В: Те динамические библиотеки, которые производит Cygwin, — это настоящие DLL?

О: Да. Но они зависят от окружения Cygwin. Чтобы заставить их работать с кодом, который не имеет отношения к Cygwin, нужно приложить некоторые усилия.

В: Почему MinGW поддерживает тот же формат динамических библиотек, что и Cygwin?

О: Потому что эти два проекта тесно связаны и у них много общего кода. Существенная разница заключается в том, что программы, скомпилированные MinGW, могут работать на компьютерах, на которых не установлен Cygwin.

В: Почему Linux не сохраняет пути к библиотекам в исполняемых файлах? В таком случае не пришлось бы задавать переменную `LD_LIBRARY_PATH`.

О: Такой выбор был сделан при проектировании. Не сохраняя путь, вы получаете куда больше контроля над версиями библиотек, которые может использовать программа. Это очень полезно при разработке новых библиотек.

В: Почему Cygwin не использует `LD_LIBRARY_PATH` для поиска библиотек?

О: Потому что он должен использовать библиотеки для Windows — DLL. Они загружаются с помощью переменной PATH.

В: Какая компоновка лучше — статическая или динамическая?

О: В зависимости от ситуации. При статической компоновке вы получаете компактный и быстрый исполняемый файл, который легче перемещать с компьютера на компьютер. Динамическая компоновка дает больше возможностей для настройки программы во время ее выполнения.

В: Если динамическую библиотеку используют сразу несколько программ, она загружается более одного раза? Или в памяти она общая для всех?

О: Все зависит от операционной системы: одни загружают отдельные копии для каждого процесса, другие делают копию общей, чтобы сэкономить память.

В: Являются ли динамические библиотеки лучшим способом сконфигурировать приложение?

О: Как правило, проще использовать конфигурационные файлы. Но если вы хотите подключиться к какому-нибудь внешнему устройству, вам понадобятся отдельные динамические библиотеки, выполняющие роль драйверов.

Ваш инструментарий языка Си

Вы изучили главу 8, пополнив свои знания информацией о статических и динамических библиотеках. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

#include <> производит поиск в стандартных директориях, таких как /usr/include.

-l<имя> делает компоновку с файлом, который находится в стандартной директории, например в /usr/lib.

-L<имя> добавляет каталог к списку стандартных директорий.

-I<имя> добавляет каталог к списку стандартных заголовочных директорий.

Команда ar создает библиотечный архив из объектных файлов.

Библиотечные архивы имеют названия вида lib<что-то>.a.

gcc -shared преобразует объектные файлы в динамические библиотеки.

Динамические библиотеки компонуются во время выполнения программы.

Библиотечные архивы компонуются статически.

Динамические библиотеки имеют расширения .so, .dylib, .dll или .dll.a.

В разных операционных системах динамические библиотеки называются по-разному.

Имя: Дата:

Лабораторная работа 2

OpenCV

В этой лабораторной работе содержится описание программы, которую вы должны изучить и создать, применив полученные при чтении нескольких последних глав знания.

Этот проект больше, чем те, с которыми вы сталкивались до этого момента. Поэтому прежде чем начинать, не торопитесь и внимательно все прочтите. Не переживайте, если у вас что-то не получится, — здесь нет никаких новых сведений о языке Си. Если необходимо, вы можете продолжить чтение книги и вернуться к этой лабораторной работе позже.

Выполнение этой задачи целиком и полностью зависит от вас, и кода для ее решения у вас не будет.

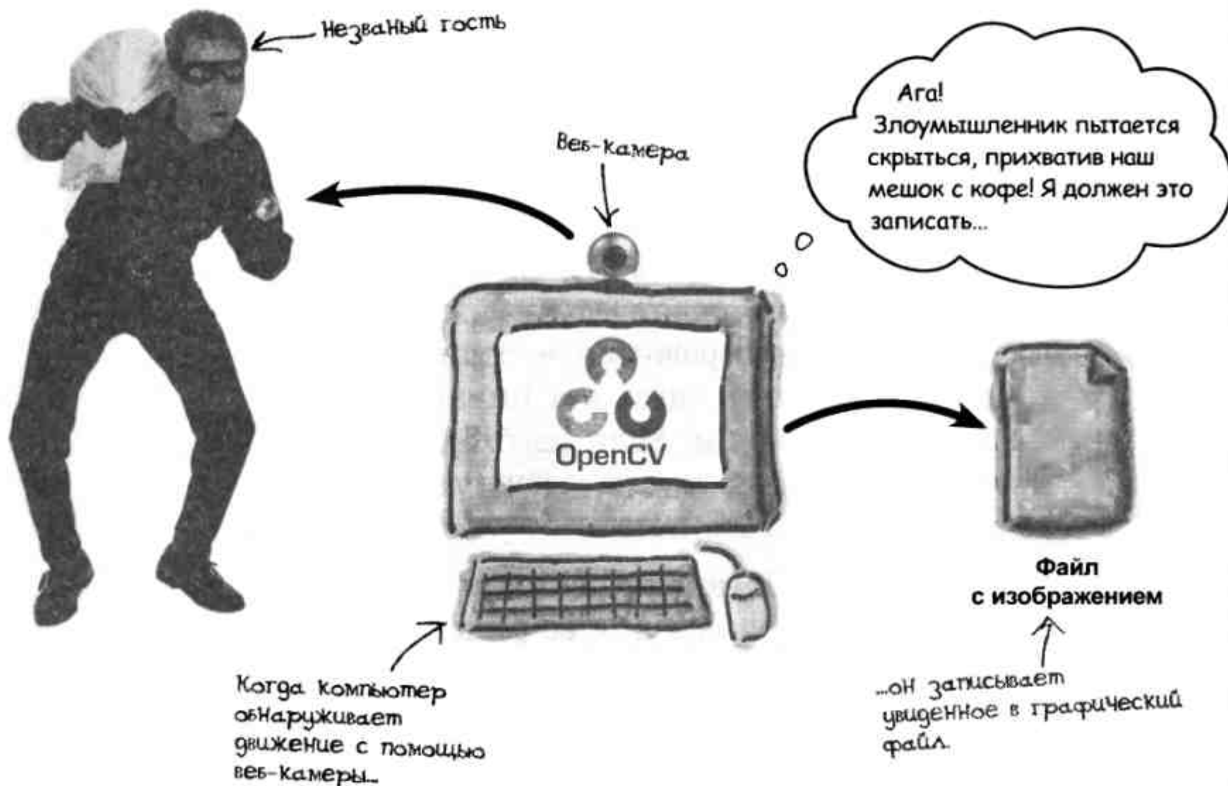
Задание: превратите свой компьютер в прибор для обнаружения незваных гостей

Представьте, что компьютер в ваше отсутствие может присматривать за домом и сообщать о подозрительных личностях, которые бродят вокруг. Что ж, это вполне реально, если воспользоваться обычной веб-камерой и возможностями *OpenCV*!

Вот что вам нужно спроектировать.

Детектор незваных гостей

Ваш компьютер будет постоянно следить за тем, что происходит вокруг. Обнаружив движение, он запишет в файл текущее изображение с веб-камеры. И если вы поместите этот файл на сетевой диск или воспользуетесь сервисом синхронизации вроде Dropbox, то сразу же получите свидетельство о любой попытке проникнуть в ваш дом.



OpenCV

OpenCV — это библиотека того, что видит компьютер, с открытым исходным кодом. Она позволяет принимать ввод с веб-камеры, обрабатывать его, анализировать изображение в реальном времени и принимать решение исходя из того, что увидел ваш компьютер. Более того, вы можете делать все это на языке Си.

OpenCV доступна для Windows, Linux и Mac.

Ее страница в «Википедии» находится по адресу:

http://opencv.willowgarage.com/wiki/FullOpenCVWiki

Установка OpenCV

Вы можете установить OpenCV на Windows, Linux или Mac.
Здесь находится инструкция по установке, в которой содержатся ссылки на последние стабильные версии:

http://opencv.willowgarage.com/wiki/InstallGuide

После того как вы установите OpenCV на ваш компьютер, должен появиться каталог под названием *samples*, на который стоит взглянуть. На странице в «Википедии» вы также увидите ссылки на опубликованные учебники. Чтобы выполнить эту лабораторную работу, вам нужно будет изучить OpenCV.

Если у вас есть желание познакомиться с этой библиотекой более детально, мы можем порекомендовать вам книгу Гари Брандски (Gary Bradski) и Эдриана Кэлера (Adrian Kaehler) *Learning OpenCV* (издательство O'Reilly).

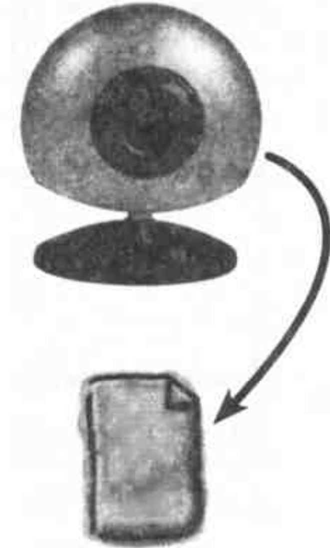
Мы находим книгу
Learning OpenCV
весьма вдохновляющей.

Что должен делать ваш код

Ваш код на Си должен выполнять следующее.

Принять ввод с веб-камеры на вашем компьютере

Вам придется работать с данными, поступающими с веб-камеры вашего компьютера в режиме реального времени. Первым делом эти данные нужно собрать. В OpenCV есть функция, которая поможет вам это сделать. Она называется **`cvCreateCameraCapture(0)`** и возвращает указатель на структуру `CvCapture`. Вы будете использовать этот указатель для связи с веб-камерой и захвата изображений.



Не забывайте делать проверку на ошибки, потому что ваш компьютер может не найти веб-камеру. В этом случае вы получите от `cvCreateCameraCapture(0)` нулевой указатель (`NULL`).

Получить изображение с веб-камеры

Вы можете получить последнее изображение с веб-камеры с помощью функции `cvQueryFrame()`. В качестве параметра она принимает указатель `CvCapture`, возвращая указатель на последнее изображение. Поэтому начало вашего кода должно выглядеть примерно так:

Файл с изображением

```
CvCapture* webcam = cvCreateCameraCapture(0);
if (!webcam)
  /* Exit with an error */
while (1) {
  IplImage* image = cvQueryFrame(webcam);
  if (image) {

  }
}
```

Это означает «не могу найти веб-камеру».

Бесконечный цикл.

Считываем изображение с веб-камеры

Здесь вам нужно обработать полученное изображение.

Если вы решите, что на изображении виден вор, вы можете сохранить картинку в файл следующим образом:

```
cvSaveImage("somefile.jpg", image, 0);
```

Имя файла с изображением.

Изображение, которое вы получили с камеры

Установите параметру значение 0, если не хотите получить изображение в градациях серого цвета.

Обнаружить злоумышленника

Теперь мы подошли к по-настоящему интересной части кода. Как вы будете решать, попал ли злоумышленник в кадр?

Один из способов заключается в отслеживании движения. В OpenCV есть функция для создания **оптического потока Фарнбага**, который в результате сравнения двух изображений делает вывод о том, насколько переместился каждый пиксель.

Здесь вам придется провести самостоятельное исследование. Вероятно, вы захотите использовать функцию `cvCalcOpticalFlowFarneback()`, чтобы сравнить два последовательных изображения с веб-камеры и создать оптический поток. Для этого вам нужно будет написать код, который оценит величину движения между двумя кадрами. Если движение превысит пороговый уровень, вы будете знать, что перед веб-камерой двигается что-то большое.

Пропустить собственные движения

Вам не нужно, чтобы запущенная программа записывала то, как вы сами выходите из дома. Поэтому лучше добавить задержку на то время, пока вы покидаете комнату.

Необязательно: показать текущий вывод с веб-камеры

Во время наших тестов в лаборатории оказалось, что довольно полезно проверять текущие изображения, которые видит программа. Для этого мы открывали окно и отображали в нем текущий вывод с веб-камеры.

Создать окно в OpenCV довольно легко:

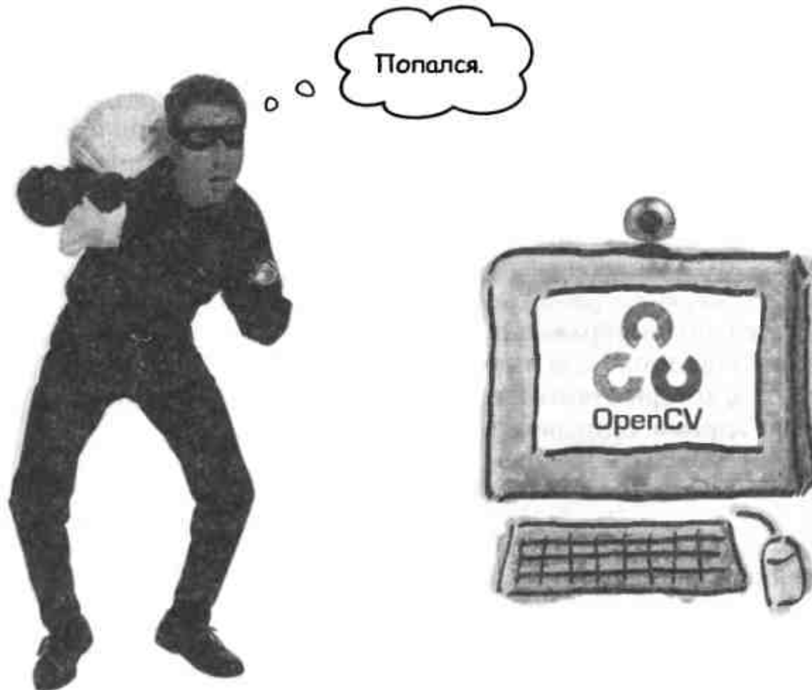
```
cvNamedWindow("Вор", 1);
```

Чтобы отобразить в окне текущее изображение, используйте следующую функцию:

```
cvShowImage("Вор", image);
```

Конечный продукт

Вы сможете считать свой проект завершенным, когда ваш компьютер будет способен автоматически фотографировать людей, которые пытаются незаметно к нему подкрасться.



Почему мы на этом остановились? Мы уверены, что у вас полно захватывающих идей насчет практического применения OpenCV. Наша лаборатория с нетерпением ждет ваших рассказов об использовании этой библиотеки.

Пришло время стать гуру языка Си...

Заключительная часть этой книги посвящена ***темам повышенной сложности.***

Вам предстоит иметь дело с некоторыми из самых продвинутых функций языка Си, поэтому вы должны убедиться, что все эти возможности доступны на вашем компьютере. Если вы используете Linux или Mac — все в порядке. Но в случае с Windows вам придется установить Cygwin.

Как только будете готовы, переворачивайте страницу и проходите сквозь врата...

9 Процессы и системные вызовы

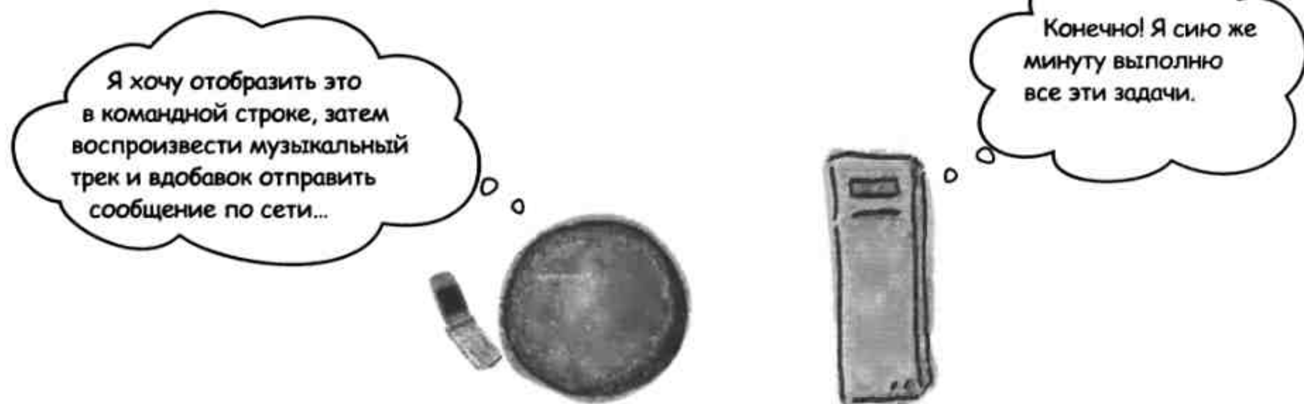
Разрушая границы

Пришло время мыслить нестандартно.

Вы уже знаете, что с помощью командной строки можно собирать сложные приложения из небольших утилит. Но что если вы хотите *вызывать другие программы* из собственного кода? В этой главе вы научитесь использовать **системные сервисы** для создания ***процессов*** и управления ими. Благодаря этому вы получите доступ к *электронной* почте, *браузеру* и *любому установленному внешнему приложению*. К концу этой главы вы сможете выйти ***за рамки языка Си***.

Системные вызовы — это ваша прямая связь с операционной системой

Программы на языке Си практически во всем полагаются на операционную систему. Они выполняют **системные вызовы**, если хотят взаимодействовать с аппаратной частью компьютера. Системные вызовы — это всего лишь функции, обитающие *внутри* **ядра** операционной системы. От них зависит большая часть кода из стандартной библиотеки Си. Каждый раз, когда вы вызываете `printf()`, чтобы вывести что-нибудь в командной строке, где-то глубоко внутри операционной системы выполняются системные вызовы для отображения текста на экране.



Давайте рассмотрим пример системного вызова с характерным названием — **system()**.

`system()` принимает один строковый параметр и выполняет его так, будто он был набран в командной строке:

```
system("dir D:");   ← Это выведет содержимое диска D:.

system("gedit");   ← Это запустит редактор в Linux.

system("say 'Конец строки'");   ← Это прочитает текст вслух в Mac
```

Функция `system()` — это простой способ запуска других программ из своего кода. Если вы создаете прототип на скорую руку, то, скорее всего, предпочтете вызвать внешнее приложение, чем писать большое количество кода на Си.

Магнитики с кодом

Эта программа записывает текст с датой и временем в конец журнального файла. Идеальным решением было бы написать ее целиком на Си, но, чтобы быстро справиться с управлением файлами, программист использовал вызов функции `system()`.

Попробуйте дописать код, который создает строку с системной командой для вывода текстового комментария с датой и временем.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char* now()
{
  time_t t;
  time (&t);
  return asctime(localtime (&t));
}

/* Главная управляющая программа.
   Записывает отчеты о работе охранного патруля. */
int main()
{
  char comment[80];
  char cmd[120];

  ..........(.......... ,.......... ,..........);

  ..........(.......... ,

         .......... ,

         .......... ,..........);
  system(cmd);
  return 0;
}
```

Эта строка возвращает строку, содержащую текущие дату и время. (← `char* now()`)

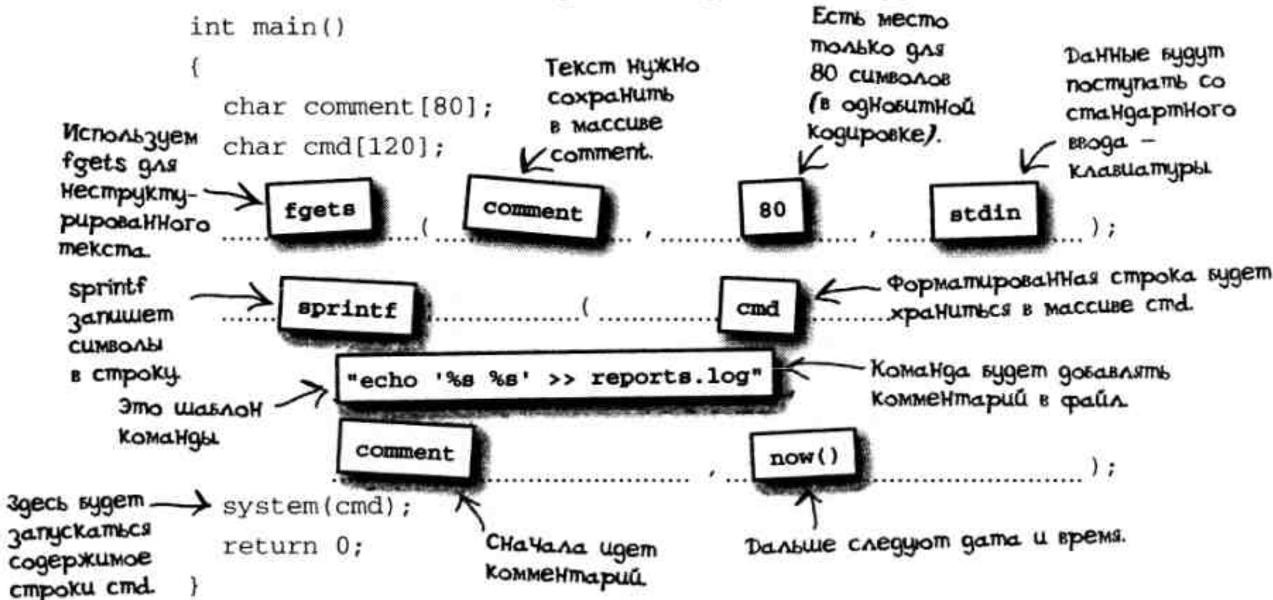
Магнитики: `sprintf`, `"echo '%s %s' >> reports.log"`, `80`, `stdin`, `cmd`, `printf`, `120`, `comment`, `fgets`, `comment`, `now()`, `scanf`, `stdout`

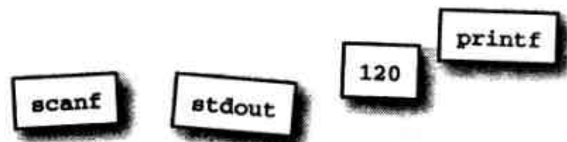
Магнитики с кодом. Решение

Эта программа записывает текст с датой и временем в конец журнального файла. Идеальным решением было бы написать ее целиком на Си, но, чтобы быстро справиться с управлением файлами, программист использовал вызов функции `system()`.

От вас требовалось дописать код, который создает строку с системной командой для вывода текстового комментария с датой и временем.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char* now()
{
  time_t t;
  time (&t);
  return asctime(localtime (&t));
}

/* Главная управляющая программа.
   Записывает отчеты о работе охранного патруля. */
```





Тест-драйв

Давайте скомпилируем программу и посмотрим на нее в действии.

```
File Edit Window Help Who'sYourUser
> gcc guard_log.c -o guard_log
> ./guard_log
Отметился в Crom — сложной программе для расчета
процентов.
> ./guard_log
Лидер синих рапортует о дыре в барьере.
>
```

Здесь программа компилируется.

Здесь программа запускается.

Запускается во второй раз.

Это комментарий

Еще один комментарий

Теперь загляните в директорию, где находится программа, — там появился новый файл под названием *reports.log:*

```
Отметился в Crom — сложной программе для расчета
процентов.
Чт Окт 29 11:25:53 2015


Лидер синих рапортует о дыре в барьере.
Чт Окт 29 11:26:06 2015
```

Это временные отметки.

Это файл reports.log, который создала наша программа.

reports.log

Программа работает. Она считывает комментарии из командной строки и записывает их в конец файла с помощью вызова команды echo.

Все это вы могли полностью написать на Си, однако использование функции system() упростило вашу программу и для ее создания потребовалось минимум усилий.

не бывает Глупых Вопросов

В: Встраивается ли функция system() внутрь моей программы?

О: Нет. Функция system(), как и все системные вызовы, находится за пределами вашей программы в главной операционной системе.

В: Выходит, если я выполняю системный вызов, то я обращаюсь к внешнему фрагменту кода, такому как библиотека?

О: Что-то вроде. Но в зависимости от операционной системы могут быть нюансы. В одних операционных системах код для системных вызовов может находиться в самом ядре, в других — храниться в виде каких-нибудь динамических библиотек.

Кто-то проник в систему

У функции `system()` есть и свои недостатки. Несмотря на высокую скорость и простоту в использовании, она немного ненадежная. Но прежде чем подробно рассматривать проблемы, связанные с функцией `system()`, давайте подумаем, насколько сложно сломать нашу программу.

ТРЕВОГА! ТРЕВОГА! Главная система безопасности была взломана!

Код собирает воедино строку, которая содержит команду:

```
echo ' | <комментарий> | | <временная отметка> | ' >> reports.log
```

Но что если кто-то введет комментарий вроде этого?

```
echo ' | ' && ls / && echo ' | | <timestamp> | ' >> reports.log
```

Вставляя в текст код командной строки, вы можете заставить программу выполнить **все что угодно**:

С помощью этой программы пользователь может запускать на компьютере любую команду, какую он только захочет

```
File Edit Window Help Yikes
> ./guard_log
' && ls / && echo '

Applications            System   dev           private
Developer               Users    etc           sbin
Library                 Volumes  home          tmp
Network                 bin      mach_kernel   usr
Space Paranoids Source  cores    net           var
>
```

Это листинг корневой директории

Насколько все серьезно? Имея доступ к `guard_log`, пользователь может легко запустить какую-либо программу. Представьте, что код вызывается из *веб-сервера*. А что если он принимает данные из *файла*?

Безопасность не единственная проблема

В этом примере вставляется код для вывода содержимого корневой директории. но таким же образом можно было бы *удалить файлы* или *запустить вирус*. Однако безопасность — не единственная причина для беспокойства.

- **Что если комментарий содержит апострофы?**
  Это может разрушить кавычки внутри команды.

- **Что если из-за переменной PATH функция system() вызовет не ту программу?**

- **Что если вызываемая программа нуждается в предварительной настройке специфических переменных окружения?**

Функцию `system()` легко использовать, но в большинстве случаев вам понадобится что-то более структурированное. Нужен способ для вызова *конкретных* программ с указанием аргументов командной строки и, возможно, даже *переменных окружения*.

Уголок ботана

Что такое ядро?

На большинстве компьютеров системные вызовы представляют собой функции, находящиеся в **ядре** операционной системы. Но что такое ядро? Вы не можете увидеть его на экране, но оно всегда рядом — управляет вашим компьютером. Ядро — самая важная часть вашей системы, которая отвечает за **три вещи**:

Процессы

Ни одна программа в системе не сможет работать, если ядро не загрузит ее в память. Ядро создает процессы и следит за тем, чтобы те получали необходимые им ресурсы. Оно также выискивает программы, которые становятся слишком требовательными к ресурсам или аварийно завершают свою работу.

Память

Ваш компьютер обладает ограниченным объемом памяти, поэтому ядро должно тщательно относиться к тому, сколько памяти может занять тот или иной процесс. Ядро может увеличить **размер виртуальной памяти**, незаметно загружая на диск участки памяти (и выгружая их оттуда при необходимости).

Аппаратное обеспечение

Чтобы взаимодействовать с оборудованием, подключенным к вашему компьютеру, ядро использует **драйверы устройств**. Ваша программа может работать с клавиатурой, экраном и графическим процессором, почти ничего о них не зная, потому что ядро обращается к ним от вашего имени.

Системные вызовы — это функции, которые ваша программа использует для общения с ядром.

Функция exec() дает вам больший контроль

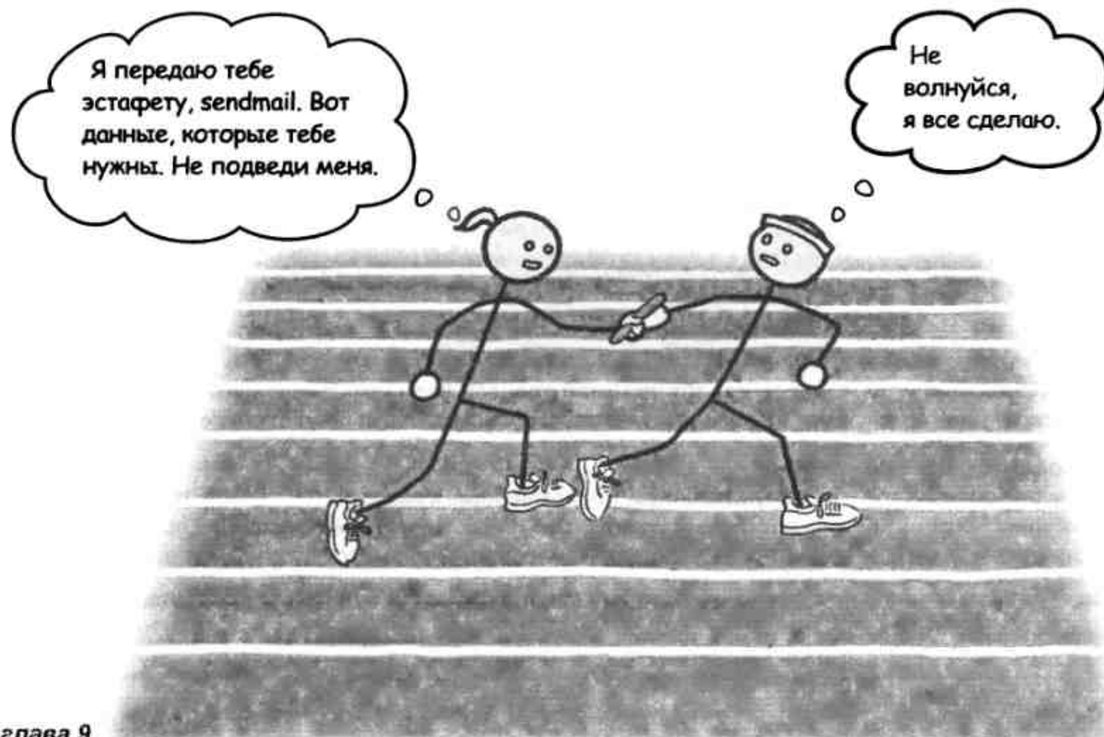
Когда вы вызываете функцию `system()`, операционная система должна интерпретировать строку с командой и решить, какие программы нужно запускать и как именно это делать. В этом и заключается проблема: строка должна быть *интерпретирована*, а вы уже видели, как легко при этом допустить ошибки. Поэтому нужно избавиться от **неопределенности** и сообщить операционной системе, с какой именно программой мы хотим работать. Для этого есть функция **`exec()`**.

Функции exec() подменяют текущий процесс

Процесс — это всего лишь программа, работающая в памяти. Набрав **`taskmgr`** (в Windows) или **`ps -ef`** (на большинстве других компьютеров), вы увидите процессы, запущенные в вашей системе. Операционная система присваивает каждому процессу число — **PID** (**process identifier** — «идентификатор процесса»).

Процесс — это программа, работающая в памяти.

Функция `exec()` **подменяет текущий процесс**, запуская какую-то другую программу. Вы можете указать *аргументы командной строки* или *переменные окружения*, которые нужно использовать, и, когда новое приложение запустится, оно будет иметь точно такой же PID, как и старое. Это похоже на эстафету: ваша программа передает свой процесс другой программе.



Существует много функций exec()

Со временем программисты создали несколько вариантов функции `exec()`. Каждая версия отличается именем и набором параметров. Однако, несмотря на множество этих версий, вам по-настоящему понадобятся только две из них — с поддержкой **списков** и **массивов**.

Функции exec() находятся в unistd.h.

Функции с поддержкой списков — execl(), execlp(), execle()

Эти функции принимают аргументы командной строки в виде списка параметров, например:

- **Программа.**
  Функциям `execl()` и `execle()` нужен полный путь к программе, а `execlp()` — просто имя, которое следует найти. В любом случае первый параметр должен сообщить функции `exec()`, какую программу он будет запускать.

- **Аргументы командной строки.**
  Вы должны последовательно перечислить аргументы командной строки, которые хотите использовать. Запомните: ***первым*** аргументом командной строки всегда является имя программы. Это значит, что для функций `exec()` два первых параметра, поддерживающих списки, всегда должны быть ***одинаковыми***.

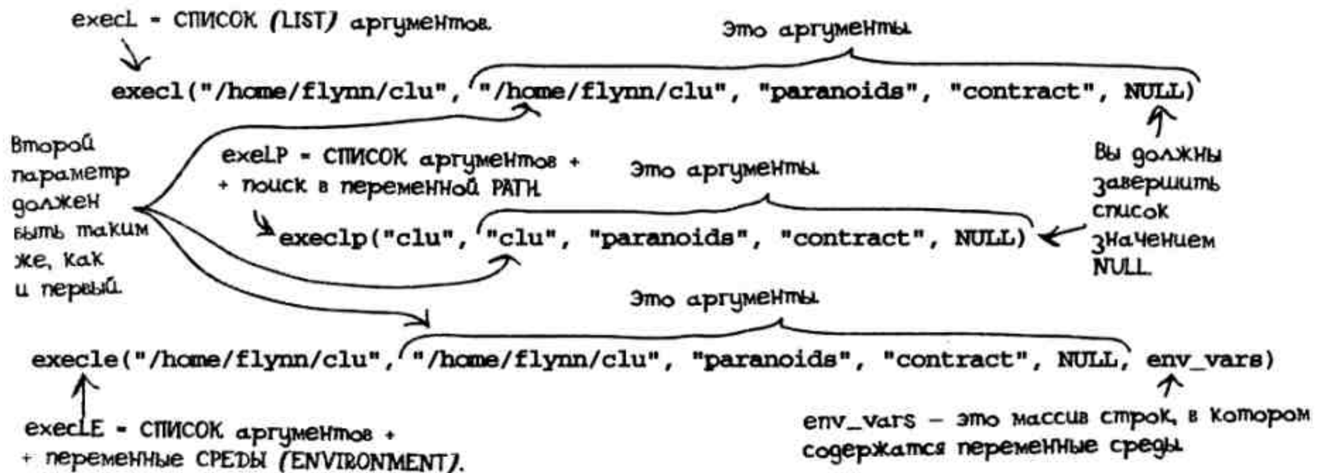
- **NULL.**
  Именно так. Вслед за последним аргументом командной строки вам нужно указать **NULL**. Благодаря этому функция поймет, что больше аргументов нет.

- **Переменные среды (необязательно).**
  Если вы вызовете функцию `exec()`, чье имя заканчивается на `...e()`, то сможете также передать массив переменных среды. Это всего лишь набор строк, например `"POWER=4", "SPEED=17", "PORT=OPEN", ...` .

Будьте осторожны!

Пробелы в аргументах командной строки могут запутать MinGW.

*Если вы передадите два аргумента: "Я люблю" и "черепах", то программы, скомпилированные MinGW, могут послать **три** значения: "Я", "люблю" и "черепах".*



Функции с поддержкой массивов — execv(), execvp(), execve()

Если ваши аргументы командной строки уже хранятся в массиве, то эти две функции могут показаться вам более легкими в использовании:

execV - массив или ВЕКТОР (VECTOR) с аргументами →

```
execv("/home/flynn/clu", my_args);
```

execVP - массив/ВЕКТОР с аргументами + поиск в переменной PATH →

```
execvp("clu", my_args);
```

← Аргументы должны храниться в строковом массиве my_args.

Единственная разница между этими двумя функциями заключается в том, что **execvp** будет искать программу с помощью переменной PATH.

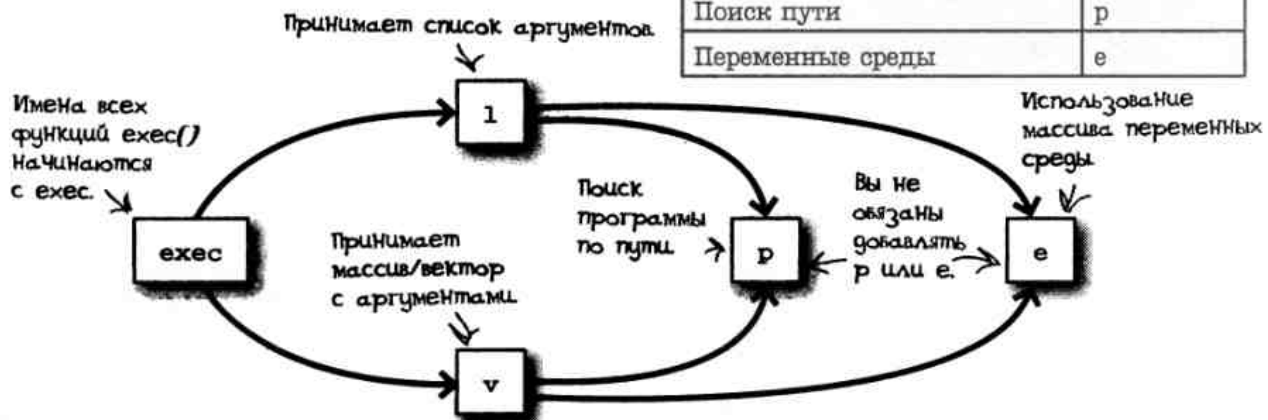
Как запомнить функции exec()

Вы можете догадаться, какая из функций exec() вам нужна, составляя ее имя. После exec() могут следовать один или два символа, ими могут быть **l**, **v**, **p** или **e**. По ним вы можете судить о возможностях функции. Например, для функции execle():

execle = exec + l + e = СПИСОК (LIST) аргументов + + переменные СРЕДЫ (ENVIRONMENT)

Символы l и v всегда идут перед p и e, причем два последних являются необязательными.

Применение	Символ
Список аргументов	l
Массив/вектор с аргументами	v
Поиск пути	p
Переменные среды	e



Передача переменных среды

У каждого процесса есть набор *переменных среды*. Это значения, которые выводятся, если ввести в командной строке `set` или `env`. Обычно они предоставляют процессам полезную информацию, например местоположение домашней директории или место поиска других программ. Приложения на языке Си могут считывать переменные среды с помощью системного вызова **getenv()**. Справа вы можете видеть, как `getenv()` используется в программе `diner_info`.

Если вы хотите запустить программу с указанием аргументов командной строки *и* переменных среды, вы можете сделать это следующим образом:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Обедов: %s\n", argv[1]);
  printf("Сок: %s\n", getenv("JUICE"));
  return 0;
}
```

Функция getenv() из stdlib.h позволяет считывать переменные среды

diner_info.c

Вы можете создать набор переменных среды в виде массива строковых указателей

Любая переменная среды имеет вид имя = значение.

Последний элемент в массиве должен равняться NULL

```
char *my_env[] = {"JUICE=персик и яблоко", NULL};
execle("diner_info", "diner_info", "4", NULL, my_env);
```

execle принимает список аргументов и переменных среды

my_env содержит переменные среды

Функция `execle()` сначала задаст аргументы командной строки и установит переменные среды, а затем заменит текущий процесс на `diner_info`.

```
File Edit Window Help MoreOJ
> ./my_exec_program
Обедов: 4
Сок: персик и яблоко
>
```

Будьте осторожны!

Если вы передаете переменные среды в Cygwin, не забудьте включить в их список переменную PATH.

В Cygwin переменная `PATH` *используется при загрузке программ. Поэтому при передаче переменных среды не забудьте добавить* `PATH=/usr/bin`.

Но что если возникнет проблема?

Если при вызове программы возникнут проблемы, текущий процесс продолжит свою работу. Это удобно, поскольку в случае неудачного запуска второго процесса вы сможете продолжить выполнение программы с места, где возникла ошибка, предоставив пользователю более подробную информацию о случившемся. И, к счастью, в стандартной библиотеке Си есть код, который может вам в этом помочь.

Большинству системных вызовов присущи одни и те же проблемы

Поскольку системные вызовы зависят от *внешних* факторов, они могут повести себя неконтролируемым образом. Чтобы этого избежать, для большинства из них были предусмотрены одинаковые виды ошибок.

Возьмем, к примеру, вызов `execle()`. Если он сработал неправильно, это очень легко заметить. При успешном выполнении `exec()` текущий процесс останавливает свою работу. Если после вызова `exec()` работа программы *возобновилась*, то, вероятно, возникла проблема:

```
execle("diner_info", "diner_info", "4", NULL, my_env);
puts("Старик, код из diner_info должно быть сломался");
```

Если функция execle() сработала, эта строчка кода никогда не выполнится.

Но недостаточно просто понять, *сработал ли* системный вызов. Как правило, хочется знать, *почему* он завершился неудачей. В связи с этим большинство системных вызовов следуют **золотым правилам неудачного выполнения**.

Золотые правила неудачного выполнения

* Привести все в порядок, насколько это возможно.
* Присвоить переменной errno код ошибки.
* Вернуть -1.

В файле *errno.h* объявлена глобальная переменная **errno**, равно как и множество других значений для описания ошибок, например:

EPERM=1	Операция недопустима
ENOENT=2	Нет такого файла или каталога
ESRCH=3	Нет такого процесса
EMULLET=81	Плохая стрижка

Это значение доступно не во всех системах.

Теперь вы *можете* проверить `errno` на соответствие каждому из этих значений или же получить стандартный текст ошибки, используя функцию **`strerror()`**, которая находится в *string.h*:

```
puts(strerror(errno));
```

strerror() преобразует код ошибки в сообщение.

Таким образом, если система не найдет программу, которую вы хотите запустить, и присвоит переменной `errno` значение ENOENT, приведенный ниже код выведет такое сообщение:

```
No such file or directory
```

Упражнение

Чтобы получить данные о конфигурации сети, на разных компьютерах нужно вводить разные команды. В Linux и Mac это программа `/sbin/ifconfig`, в Windows — утилита `ipconfig`, которая находится в **директории для системных команд**.

Данная программа пытается запустить команду `/sbin/ifconfig`, а в случае неудачи — команду `ipconfig`. Ни для одной из этих утилит не нужно передавать никаких аргументов. Однако хорошо подумайте, какой именно вид команды `exec()` вам нужен.

```
#include <stdio.h>
.............................................
```

Какой заголовок вам понадобится?

Здесь нужно запустить /sbin/ifconfig. Что мы должны проверить?

Здесь мы должны запустить команду ipconfig и узнать, сработала ли она.

```
int main()
{
  if (.........................................................)
    if (execlp(...............................................) {
      fprintf(stderr, "Не удалось запустить ipconfig: %s", ..........);
      return 1;
    }
  return 0;
}
```

Как вы думаете, что сюда нужно подставить?

Упражнение Решение

Чтобы получить данные о конфигурации сети, на разных компьютерах нужно вводить разные команды. В Linux и Mac это программа `/sbin/ifconfig`, в Windows — утилита `ipconfig`, которая находится в **директории для системных команд**.

Данная программа пытается запустить команду `/sbin/ifconfig`, а в случае неудачи — команду `ipconfig`. Ни для одной из этих утилит не нужно передавать никаких аргументов. Однако хорошо подумайте, какой именно вид команды `exec()` вам нужен.

```
#include <stdio.h>
#include <unistd.h>     ← Этот заголовок нужен для функций exec().
#include <errno.h>      ← Этот нужен для переменной errno.
#include <string.h>     ← Этот позволяет выводить ошибки с помощью strerror().

int main()
{
  if ( execl("/sbin/ifconfig", "/sbin/ifconfig", NULL) == -1 )
    if (execlp( "ipconfig", "ipconfig", NULL) == -1 ) {
      fprintf(stderr, "Не удалось запустить ipconfig: %s", strerror(errno) );
      return 1;
    }
  return 0;
}
```

Используйте execl(), поскольку у вас есть путь к файлу программы.

Если execl() возвращает -1, значит, она завершилась неудачно. Поэтому нам, наверное, стоит поискать ipconfig.

execlp() позволяет найти ipconfig в одном из системных каталогов.

Проверка на значение -1 на случай, если команда завершилась неудачей.

strerror() отобразит любые возникшие проблемы.

не бывает Глупых Вопросов

В: Разве не проще использовать функцию system(), нежели exec()?

О: Да, проще. Но с system() могут возникать разные ошибки, поскольку операционной системе нужно интерпретировать строку, которую вы ей передаете. Особенно если строка создается динамически.

В: Почему у exec() так много производных функций?

О: Постепенно возникла необходимость создавать процессы разными способами. Разные версии exec() были придуманы для большей гибкости.

В: Мне нужно каждый раз проверять значение, которое возвращает системный вызов? Не слишком ли это удлиняет программу?

О: Если вы не станете проверять свои системные вызовы на ошибки, код будет короче. Но при этом он, скорее всего, будет больше предрасположен к сбоям. Об ошибках лучше думать во время написания кода — позже их будет легче отловить.

В: Могу ли я сделать что-то после вызова функции exec()?

О: Нет. Если функция exec() выполнится успешно, она сделает так, что в текущем процессе вместо вашей программы будет работать другая. Это значит, что приложение, которое содержит вызов exec(), остановится сразу, как только этот вызов произойдет.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Системные вызовы — это функции, которые обитают внутри операционной системы.
- Системный вызов — это вызов кода, находящегося за пределами вашей программы.
- system() — это системный вызов для запуска строк с командами.
- Функцию system() легко использовать, но она может привести к ошибкам.
- Системные вызовы вида exec() позволяют получить больше контроля над запускаемыми программами.
- Существует несколько версий системного вызова exec().
- При возникновении проблемы системные вызовы, как правило (но не всегда), возвращают значение –1.
- Они также присваивают переменной errno код ошибки.

Перемешанные сообщения

Ребята из кофейни Starbuzz придумали новую программу `coffee` для генерирования заказов:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *w = getenv("EXTRA");
  if (!w)
    w = getenv("FOOD");
  if (!w)
    w = argv[argc - 1];
  char *c = getenv("EXTRA");
  if (!c)
    c = argv[argc - 1];
  printf("%s c %s\n", c, w);
  return 0;
}
```

Чтобы опробовать ее на деле, они создали тестовую программу. Можете ли вы сопоставить следующие фрагменты кода с текстом, который выводится на экран?

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[]){
```

Сюда подставляются варианты кода.

```
  fprintf(stderr,"Не могу создать заказ: %s\n", strerror(errno));
    return 1;
  }
  return 0;
}
```

Варианты Кода:

Соедините каждый вариант кода с одним из возможных программных выводов.

Возможный вывод:

```
char *my_env[] = {"FOOD=кофе", NULL};
if(execle("./coffee", "./coffee", "пончики", NULL, my_env) == -1){
  fprintf(stderr,"Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
кофе с пончиками
```

```
char *my_env[] = {"FOOD=пончиками", NULL};
if(execle("./coffee", "./coffee", "сливки", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
сливки с пончиками
```

```
if(execle("./coffee", "кофе", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
пончики с кофе
```

```
char *my_env[] = {"FOOD=пончиками", NULL};
if(execle("./coffee", "кофе", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
кофе с кофе
```

Перемешанные сообщения Решение

Ребята из кофейни Starbuzz придумали новую программу `coffee` для генерирования заказов:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *w = getenv("EXTRA");
  if (!w)
    w = getenv("FOOD");
  if (!w)
    w = argv[argc - 1];
  char *c = getenv("EXTRA");
  if (!c)
    c = argv[argc - 1];
  printf("%s c %s\n", c, w);
  return 0;
}
```

Чтобы опробовать ее на деле, они создали тестовую программу. Можете ли вы сопоставить следующие фрагменты кода с текстом, который выводится на экран?

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[]){

  fprintf(stderr,"Не могу создать заказ: %s\n", strerror(errno));
    return 1;
  }
  return 0;
}
```

Сюда подставляются варианты кода.

Варианты Кода:

```
char *my_env[] = {"FOOD=кофе", NULL};
if(execle("./coffee", "./coffee", "пончики", NULL, my_env) == -1){
  fprintf(stderr,"Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=пончиками", NULL};
if(execle("./coffee", "./coffee", "сливки", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
if(execle("./coffee", "кофе", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=пончиками", NULL};
if(execle("./coffee", "кофе", NULL, my_env) == -1){
  fprintf(stderr," Не могу запустить процесс 0: %s\n", strerror(errno));
  return 1;
}
```

Возможный вывод:

```
кофе с пончиками
```

```
сливки с пончиками
```

```
пончики с кофе
```

```
кофе с кофе
```

Читаем новости с помощью RSS

RSS-потоки на сайтах — распространенный способ публикации последних новостей. Любой RSS-поток представляет собой обычный файл в формате XML с краткими описаниями статей и ссылками. Конечно, на Си можно написать программу, которая будет читать RSS-файлы прямо из Интернета, однако для этого нужно затронуть концепции, которые мы еще не рассматривали. Но это не проблема, если вы можете найти другую программу, которая будет выполнять обработку RSS-потоков за вас.

Загрузите программу RSS Gossip по ссылке *https://github.com/dogriffiths/rssgossip/zipball/master.* Если у вас не установлен язык программирования Python, вы можете взять его здесь: *http://www.python.org/.*

Я хочу получать последние новости о Pajama Death.

Редактор

RSS Gosip — это небольшой **скрипт на языке Python** для поиска в RSS-потоке новостей, содержащих определенный фрагмент текста. Чтобы этот скрипт работал, у вас должен быть установлен Python. Используя этот язык программирования и *rssgossip.py*, вы можете искать новости следующим образом:

Запуск происходит в Unix-подобной среде.

Вам нужно создать переменную среды, которая содержит адрес RSS-потока.

Здесь запускается скрипт rssgossip, которому передается поисковый запрос.

Это ненастоящий поток. Вы должны заменить его на тот, который найдете в Интернете.

```
File Edit Window Help ReadAllAboutIt
> export RSS_FEED=http://www.cnn.com/rss/celebs.xml
> python rssgossip.py 'pajama death'
Музыкальная группа Pajama Death выпустила собственную
линейку кухонной посуды.
У солиста Pajama Death появилась новая пассия.
"Я никогда не ел летучих мышей", - заявил Хенкок из Pajama Death.
```

О, у меня только что родилась замечательная идея! Почему бы не написать программу, которая могла бы вести поиск одновременно **во множестве** RSS-потоков? Это можно сделать?

Упражнение

Редактор хочет установить себе на компьютер программу, которая сможет производить одновременный поиск по многим RSS-потокам, поочередно запуская *rssgossip.py* для нескольких адресов. К счастью, **безработные актеры** уже начали писать эту программу за вас. Однако у них возникли проблемы с запуском *rssgossip.py* с помощью функции `exec()`. Хорошо подумайте, что требуется для запуска скрипта, а затем допишите код приложения **`newshound`**.

Чтобы сэкономить место, мы не включили в этот листинг строчки с директивами #include.

```
int main(int argc, char *argv[])
{
  char *feeds[] = {"http://www.cnn.com/rss/celebs.xml",
                   "http://www.rollingstone.com/rock.xml",
                   "http://eonline.com/gossip.xml"};
  int times = 3;
  char *phrase = argv[1];
  int i;
  for (i = 0; i < times; i++) {
    char var[255];
    sprintf(var, "RSS_FEED=%s", feeds[i]);
    char *vars[] = {var, NULL};
    if (..................("/usr/bin/python", "/usr/bin/python",
                 ..........................................) == -1) {
      fprintf(stderr, "Не могу запустить скрипт: %s\n",
strerror(errno));
      return 1;
    }
  }
  return 0;
}
```

Это RSS-потоки, которые нужны редактору (вы можете выбрать свои собственные).

Мы будем передавать поисковый запрос в виде аргумента.

Перебираем каждый поток.

Это массив с переменными окружения.

Python на компьютере редактора с операционной системой Mac установлен в этом каталоге.

Вам нужно подставить сюда имя функции.

Вы должны вставить сюда остальные параметры функции.

newshound.c

И для дополнительных баллов...

Что выведет программа во время своей работы?

Упражнение решение

Редактор хочет установить себе на компьютер программу, которая сможет производить одновременный поиск по многим RSS-потокам, поочередно запуская *rssgossip.py* для нескольких адресов. К счастью, **безработные актеры** уже начали писать эту программу за вас. Однако у них возникли проблемы с запуском *rssgossip.py* с помощью функции `exec()`. Вам нужно было подумать, что требуется для запуска скрипта, а затем дописать код приложения **newshound**.

```
int main(int argc, char *argv[])
{
  char *feeds[] = {"http://www.cnn.com/rss/celebs.xml",
                   "http://www.rollingstone.com/rock.xml",
                   "http://eonline.com/gossip.xml"};
  int times = 3;
  char *phrase = argv[1];
  int i;
  for (i = 0; i < times; i++) {
    char var[255];
    sprintf(var, "RSS_FEED=%s", feeds[i]);
    char *vars[] = {var, NULL};
    if ( execle ("/usr/bin/python", "/usr/bin/python",
         "./rssgossip.py", phrase, NULL, vars ) == -1) {
      fprintf(stderr, "Не могу запустить скрипт: %s\n",
strerror(errno));
      return 1;
    }
  }
  return 0;
}
```

Вы используете список аргументов и переменные среды, поэтому сюда вставляется execlE.

Это название скрипта на Python.

Это поисковый запрос, который передается в виде аргумента командной строки.

В качестве дополнительного параметра идет массив с переменными среды.

newshound.c

Но все же, что выведет программа во время своей работы?

Тест-драйв

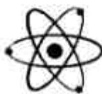
После того как вы скомпилируете и запустите программу, убедитесь, что она работает:

```
File Edit Window Help ReadAllAboutIt
> ./newshound 'pajama death'
Бывший барабанщик Pajama Death делится откровениями.
Новый альбом Pajama Death должен выйти в следующем месяце.
```

Программа `newshound` взаимодействует со скриптом *rssgossip.py*, который использует данные из массива RSS-потоков.

На самом деле есть одна проблема.

Несмотря на то что программа `newshound` смогла запустить скрипт *rssgossip.py*, похоже, она сделала это не для *всех потоков*. Фактически новости выводятся только для **первого потока в списке**. Следовательно, остальные новости, соответствующие поисковому запросу, были утеряны.

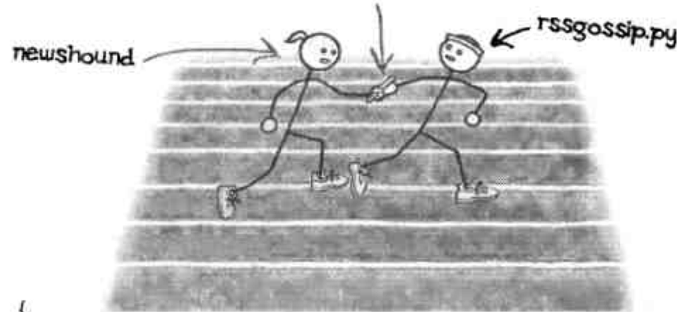


Сила мозга

Взгляните еще раз на код программы `newshound` и подумайте о том, как она работает. Как вы думаете, почему она не смогла запустить *rssgossip.py* для остальных новостных потоков?

exec() — конечная остановка для вашей программы

Функция exec() *замещает* текущий процесс новой программой. Но что происходит с программой, которая была запущена изначально? Она завершается, причем **немедленно**. Поэтому скрипт *rssgossip.py* был запущен только для первого новостного потока. После первого вызова execle() программа newshound завершила свою работу.



Цикл сработает всего один раз.

```
for (i = 0; i < times; i++) {
  ...
  if (execle("/usr/bin/python", "/usr/bin/python",
             "./rssgossip.py", phrase, NULL, vars) == -1) {
  }
}
```

Вызвав execle(), программа завершается.

Будьте осторожны!

В отличие от Linux и Mac, Windows не имеет «родной» поддержки fork().

Чтобы использовать fork() в Windows, нужно сначала установить Cygwin.

Как добиться того, чтобы после запуска *другой* программы ваш изначальный процесс продолжал работать?

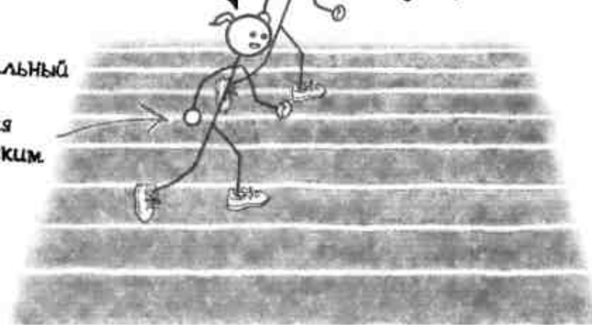
fork() клонирует ваш процесс

Мы решим эту проблему, используя системный вызов под названием **fork()**.

fork() создает точную **копию** текущего процесса. Вновь созданная копия продолжит выполнять ту же программу с той же строчки кода. У нее будут точно такие же переменные с точно такими же значениями. Отличаться будут только идентификаторы процессов.

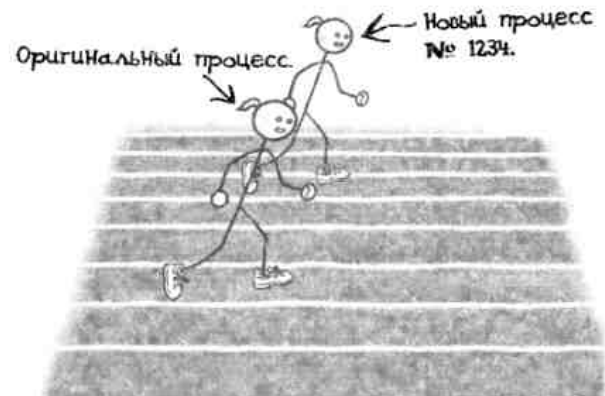
Оригинальный процесс называется **родительским**, а вновь созданный — **дочерним**.

Но как клонирование текущего процесса поможет нам решить проблему с exec()? Давайте разберемся.



Запуск дочернего процесса с помощью fork() + exec()

Весь фокус в том, чтобы вызвать функцию `exec()` только в *дочернем процессе*. Таким образом, ваш первоначальный процесс сможет продолжать свою работу. Давайте посмотрим, как это делается шаг за шагом.



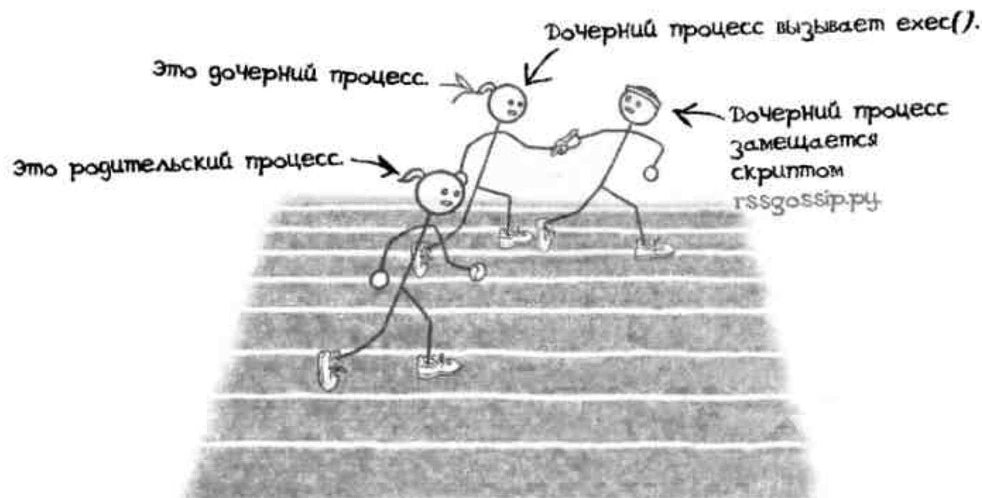
1. Создаем копию

Вызвав функцию `fork()`, скопируем наш текущий процесс.

Чтобы различать типы процессов, функция `fork()` возвращает 0 для дочернего процесса и ненулевое значение для родительского.

2. Если это дочерний процесс, вызываем exec()

На этом этапе у нас есть два идентичных процесса с одинаковым кодом. Но теперь дочерний процесс (который получил 0 от `fork()`) должен заместить себя вызовом `exec()`:



Теперь у вас есть два отдельных процесса: дочерний, выполняющий *rssgossip.py*, и первоначальный родительский, который продолжит делать что-то другое.

Магнитики с кодом

Пришло время обновить программу newshound. Она должна запускать отдельный процесс со скриптом *rssgossip.py* для каждого RSS-потока. Объем кода сократился, поэтому вы должны сосредоточиться только на главном цикле. Будьте осторожны при проверке ошибок и не смешивайте родительские и дочерние процессы!

Разместите здесь свои магнитики.

```
for (i = 0; i < times; i++) {
  char var[255];
  sprintf(var, "RSS_FEED=%s", feeds[i]);
  char *vars[] = {var, NULL};

}
```

Что за fork()?

Функция fork() вызывается следующим образом:

```
pid_t pid = fork();
```

Для каждого из процессов fork() возвращает целочисленное значение: 0 для дочернего и положительное число для родительского. Таким образом, родительский процесс получит идентификатор дочернего.

Но что такое **pid_t**? Для хранения идентификатора процессов в разных операционных системах используются разные типы чисел: иногда это short, иногда int. Поэтому pid_t всегда получает тот тип, который используется в операционной системе.

```
fprintf(stderr, "Не могу клонировать процесс: %s\n", strerror(errno));
```

```
fprintf(stderr, "Не могу запустить скрипт: %s\n", strerror(errno));
```

```
if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
           phrase, NULL, vars) == -1) {
```

```
return 1;
```

```
pid_t pid = fork();
```

```
}
```

```
if (pid == -1) {
```

```
if (!pid) {
```

```
}
```

```
}
```

```
return 1;
```

Магнитики с кодом Решение

Пришло время обновить программу newshound. Она должна запускать отдельный процесс со скриптом *rssgossip.py* для каждого RSS-потока. Объем кода сократился, поэтому вам нужно было сосредоточиться только на главном цикле, проявить осторожность при проверке ошибок и не смешивать родительские и дочерние процессы.

```
for (i = 0; i < times; i++) {
  char var[255];
  sprintf(var, "RSS_FEED=%s", feeds[i]);
  char *vars[] = {var, NULL};
  pid_t pid = fork();
  if (pid == -1) {
    fprintf(stderr, "Не могу клонировать процесс: %s\n", strerror(errno));
    return 1;
  }
  if (!pid) {
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                         phrase, NULL, vars) == -1) {
      fprintf(stderr, "Не могу запустить скрипт: %s\n", strerror(errno));
      return 1;
    }
  }
}
```

Сначала вызываем функцию fork(), чтобы клонировать процесс.

Если функция fork() вернула -1, значит при клонировании возникла проблема.

Если функция fork() вернула 0, значит код выполняется в дочернем процессе.

Это то же самое, что if (pid == 0).

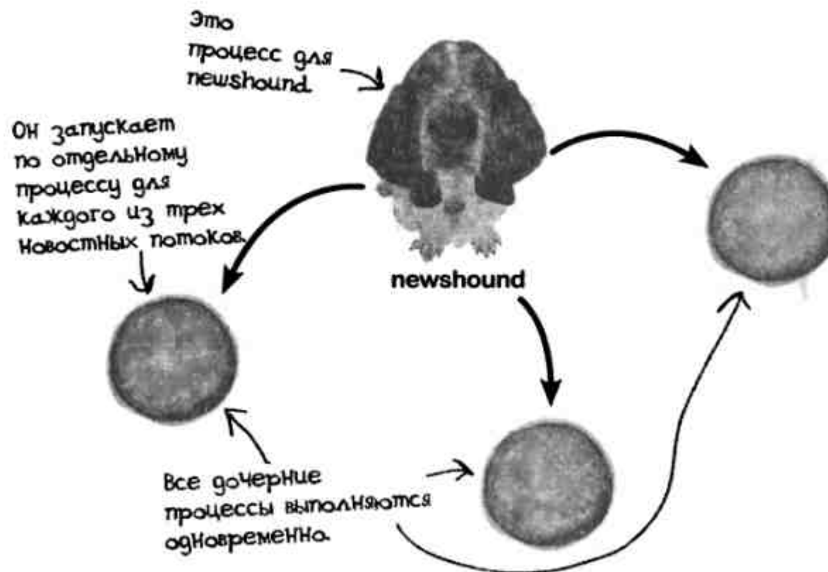
Если мы добрались сюда, значит это дочерний процесс. Поэтому нам нужно запустить скрипт с помощью exec().

Тест-драйв

Теперь, когда вы скомпилируете и запустите этот код, произойдет следующее:

```
File Edit Window Help ReadAllAboutIt
> ./newshound 'pajama death'
Бывший барабанщик Pajama Death делится откровениями.
Новый альбом Pajama Death должен выйти в следующем месяце.
Фотографии с концерта-сюрприза группы Pajama Death.
Официальные пижамы от Pajama Death уже в продаже.
«Pajama Death на пике популярности» от Генри В.
Последние новости: Pajama Death посетили премьеру.
```

Копируя себя и запуская скрипт на Python в новом процессе, программа newshound может выделять отдельный процесс для каждого RSS-потока. Но самое замечательное, что все эти процессы будут работать **одновременно**.



Это намного быстрее, чем чтение новостных потоков по одному за раз. Научившись запускать отдельные процессы с помощью `fork()` и `exec()`, вы можете не только выжать максимум из уже готовых приложений, но и улучшить производительность собственного кода.

не бывает Глупых Вопросов

В: Функция `system()` выполняет программы в отдельном процессе?

О: Да. Но `system()` предоставляет вам меньше возможностей контролировать работу этих программ.

В: Эффективно ли клонирование? Ведь копируется весь процесс целиком, а потом мы замещаем копию с помощью `exec()`.

О: Операционные системы используют множество уловок, чтобы ускорить процесс клонирования. Так, например, они не копируют информацию из родительского процесса. Вместо этого родительский и дочерний процессы делят общие данные.

В: Что произойдет, если один из процессов изменит какие-нибудь данные в памяти? Разве это не испортит все?

О: Могло бы испортить. Но операционная система перехватит фрагмент памяти, который должен измениться, и сделает отдельную его копию для дочернего процесса.

В: Довольно впечатляющий подход. У него есть какое-то название?

О: Да, он называется COW (от английского copy-on-write — «копирование при записи»).

В: `pid_t` — это просто `int`?

О: Все зависит от платформы. Единственное, что можно сказать наверняка, — это будет один из целочисленных типов.

В: Я сохранил результат вызова `fork()` в `int`, и это сработало.

О: Для хранения идентификатора процесса всегда лучше использовать `pid_t`. Иначе у вас могут возникнуть проблемы при работе с другими системными вызовами или при компиляции вашего кода на другом компьютере.

В: Почему Windows не поддерживает системный вызов `fork()`?

О: Диспетчер процессов в Windows существенно отличается от своих аналогов в других системах. В этой операционной системе очень сложно воспроизвести ухищрения, необходимые для эффективной работы `fork()`. Вероятно, по этой причине в Windows не предусмотрено никакой версии данной функции.

В: Но ведь Cygwin позволяет работать с `fork()` в Windows, верно?

О: Да. Разработчики Cygwin приложили много усилий, чтобы процессы в Windows выглядели так же, как в Unix, Linux и Mac. Но они все равно должны полагаться на Windows при создании базовых процессов, поэтому функция `fork()` в Cygwin может работать немного медленней, чем на других платформах.

В: Значит, если я заинтересован в написании кода, который будет работать в Windows, мне стоит использовать что-то другое?

О: Да, есть функция под названием `CreateProcess()` — что-то вроде улучшенной версии `system()`. Чтобы узнать о ней больше, пройдите по ссылке *http://msdn.microsoft.com* и поищите `CreateProcess`.

В: Не получится ли так, что вывод разных потоков смешается?

О: Операционная система позаботится о том, чтобы каждая строка была выведена полностью.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Системные вызовы — это функции, которые находятся в ядре.
- Функции `exec()` предоставляют вам больше контроля, чем `system()`.
- Функции `exec()` замещают текущий процесс.
- Функция `fork()` дублирует текущий процесс.
- В случае неудачи системные вызовы, как правило, возвращают –1.
- Неудачные системные вызовы присваивают переменной `errno` код ошибки.

Ваш инструментарий языка Си

Вы изучили главу 8, пополнив свои знания информацией о процессах и системных вызовах. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

system(), запускает строку как консольную команду.

execl() = список аргументов.
execle() = список аргументов + переменные среды.
execlp() = список аргументов + поиск по пути.
execv() = массив аргументов.
execve() = массив аргументов + переменные среды.
execvp() = массив аргументов + поиск по пути.

fork() дублирует текущий процесс.

Комбинация fork() и exec() создает дочерний процесс.

10 Межпроцессное взаимодействие

Общение — это хорошо

Создание процессов — это только первый шаг.

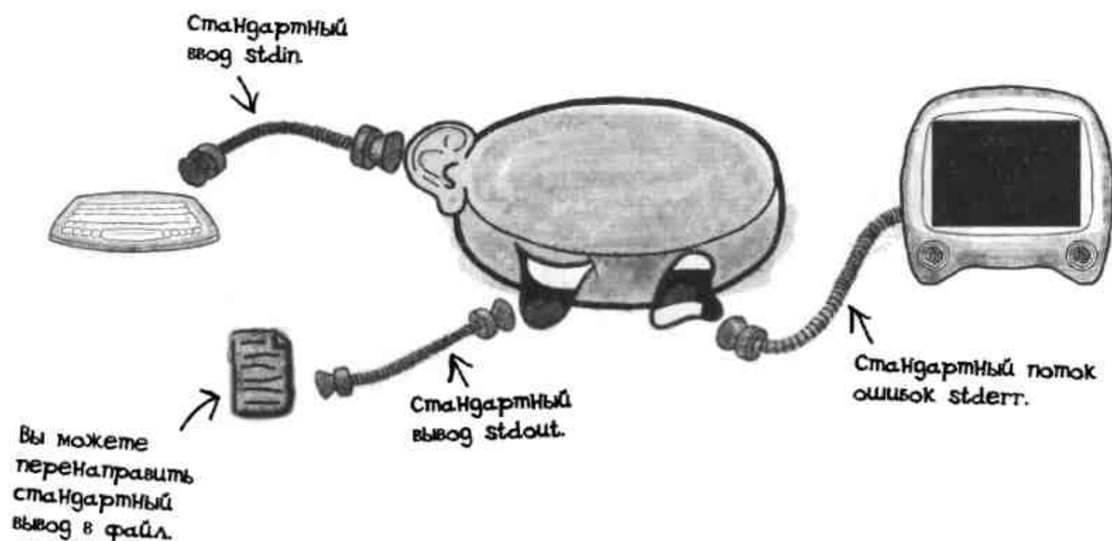
Что если вам захочется *управлять* запущенным процессом? Что если вам захочется *отправить ему данные*? Или *прочитать его вывод*? Благодаря **межпроцессному взаимодействию** процессы могут *выполнять работу* сообща. Вы увидите, что **мощь** вашего кода может возрасти многократно, если позволить ему ***общаться*** с другими программами системы.

Перенаправление ввода и вывода

Запуская программы из командной строки, вы можете перенаправлять стандартный вывод в файл, используя оператор >:

```
python ./rssgossip.py Snooki > stories.txt
```

Вы можете перенаправить вывод с помощью оператора >.



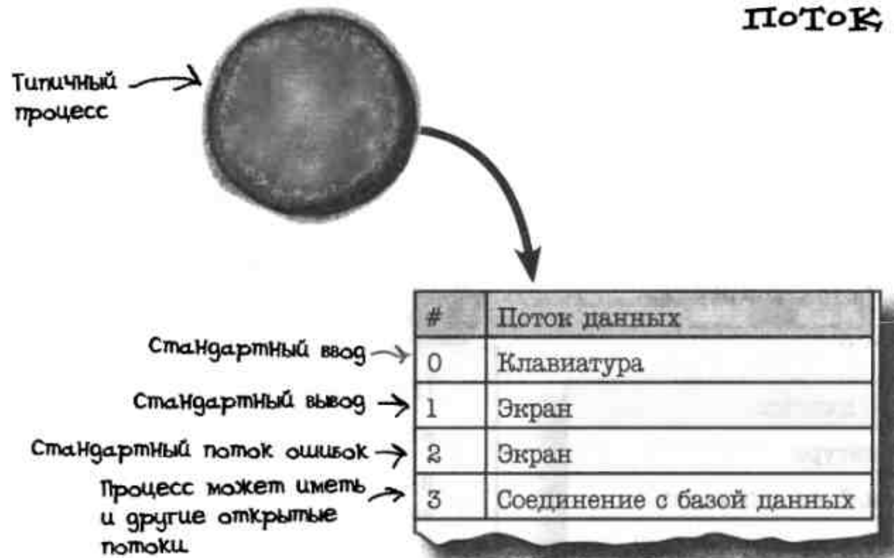
Стандартный вывод — это один из трех стандартных **потоков данных**. Исходя из своего названия, *поток данных* представляет собой поток, по которому данные попадают в процесс (или выходят из него). Помимо потоков для вывода, ввода и ошибок есть и другие, например для сетевых соединений. Перенаправляя поток процесса, вы меняете пункт назначения, в который отправляются данные. Таким образом, вы можете направить данные из стандартного вывода не на экран, а в файл.

Перенаправление действительно полезно в рамках командной строки, но можно ли сделать так, чтобы процесс ***перенаправлял себя сам***?

Типичный процесс изнутри

Любой процесс содержит внутри себя программу, которую он выполняет, а также место в стеке и куче, отведенное для данных. Ему также необходимо записывать потоки данных, к которым он подключен (например, стандартный ввод). Каждый такой поток представлен **файловым дескриптором**, который по существу является обычным числом. Процесс поддерживает порядок, храня файловые дескрипторы и их потоки данных в соответствующей **таблице**.

Файловый дескриптор — это число, которым представлен поток данных.



#	Поток данных
0	Клавиатура
1	Экран
2	Экран
3	Соединение с базой данных

В данной таблице есть колонка, содержащая числа для каждого файлового дескриптора. И хотя дескрипторы называются **файловыми,** они не обязательно должны быть подключены к реальному файлу на диске. Напротив каждого числа в таблице записывается соответствующий поток данных — подключение к клавиатуре или экрану, указатель на файл или сетевое соединение.

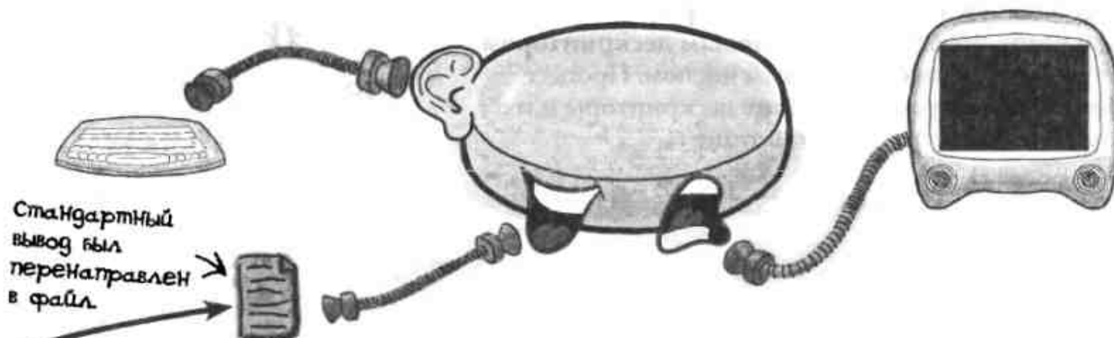
Файловый дескриптор не обязательно должен указывать на файл.

Первые три строки в таблице всегда одинаковы: строка № 0 для стандартного ввода, строка № 1 для стандартного вывода и строка № 2 для стандартного потока ошибок. Остальные строки либо пусты, либо связаны с потоками данных, которые инициировал сам процесс. Например, каждый раз, когда ваш код открывает файл для чтения или записи, в таблицу добавляется еще одна строка.

При создании процесса его стандартный ввод подключается к клавиатуре, а стандартный вывод, равно как и поток ошибок, — к экрану. Такое подключение сохранится до тех пор, пока потоки не будут перенаправлены куда-нибудь еще.

При перенаправлении всего лишь меняются потоки данных

Стандартные ввод, вывод и поток ошибок всегда остаются на своих местах в таблице дескрипторов. Но потоки данных, на которые они указывают, могут меняться.



Следовательно, если вы хотите перенаправить стандартный вывод, нужно всего лишь поменять в таблице поток данных для дескриптора № 1.

#	Поток данных
0	Клавиатура
1	~~Экран~~ Файл stories.txt
2	Экран
3	Соединение с базой данных

Все функции наподобие `printf()`, которые отправляют данные в стандартный вывод, первым делом сверяются с этой таблицей, чтобы узнать, куда указывает дескриптор № 1. Затем они записывают данные в соответствующий поток.

Процессы могут перенаправлять сами себя

До сих пор вы применяли перенаправление только в командной строке, используя операторы > и <. Но процессы могут *перенаправлять себя сами*, **перезаписывая таблицу дескрипторов**.

Уголок ботана

Так вот почему тот оператор выглядел как 2>...

Вы можете перенаправить стандартный вывод и стандартный поток ошибок, используя операторы > и 2> в командной строке:

```
./myprog > output.txt 2> errors.log
```

Теперь понятно, почему стандартный поток ошибок перенаправляется с помощью **2>**. Цифра **2** указывает на номер стандартного потока ошибок в таблице дескрипторов. На большинстве платформ вы можете перенаправлять стандартный вывод, используя оператор **1>**, а в Unix-подобных операционных системах даже перенаправить стандартный поток ошибок туда, куда направлен стандартный вывод:

```
./myprog 2>&1
```

2> означает «перенаправить стандартный поток ошибок».

&1 означает «в стандартный вывод».

С помощью fileno() можно получить дескриптор

Каждый раз, когда вы открываете файл, операционная система регистрирует новый элемент в таблице дескрипторов. Допустим, мы открыли файл следующим образом:

```
FILE *my_file = fopen("guitar.mp3", "r");
```

Помимо открытия файла *guitar.mp3* и возвращения указателя на него, операционная система также бегло просмотрит таблицу дескрипторов в поисках свободного места и зарегистрирует там новый файл.

#	Поток данных
0	Клавиатура
1	Экран
2	Экран
3	Соединение с базой данных
4	Файл guitar.mp3

Но как найти файл в таблице дескрипторов, если у вас есть только указатель? Ответ прост — нужно вызвать функцию **fileno()**.

```
int descriptor = fileno(my_file);
```

Функция вернет значение 4.

fileno() — одна из тех системных функций, которая не возвращает -1 при неудачном завершении. Если вы передаете ей указатель на открытый файл, она должна обязательно вернуть номер дескриптора.

dup2() дублирует потоки данных

Открывая файл, мы заполняем строку в таблице дескрипторов. Но что если нужно *изменить* поток данных, для которого дескриптор уже зарегистрирован? Как сделать, чтобы дескриптор № 3 указывал на другой поток данных? Для этого есть функция **dup2()**. Она копирует потоки данных из одной строки в другую. Таким образом, если указатель на файл *guitar.mp3* связан с дескриптором № 4, вы можете подключить его и к дескриптору № 3, используя следующий код:

```
dup2(4, 3);
```

#	Поток данных
0	Клавиатура
1	Экран
2	Экран
3	~~Соединение с базой данных~~ Файл guitar.mp3
4	Файл guitar.mp3

У вас по-прежнему останется один файл *guitar.mp3*, и к нему все так же будет подключен один поток данных. Но теперь этот поток будет ассоциироваться с файловыми дескрипторами № 3 и 4.

Теперь, когда вы знаете, как находить и менять элементы в таблице дескрипторов, вы можете перенаправлять стандартный вывод процесса в файл.

Вас беспокоит код для обработки ошибок?

Замечали ли вы, что при любом системном вызове код для обработки ошибок постоянно дублируется? Не стоит переживать! Мы покажем вам, как с помощью нашего запатентованного метода можно делать проверку ошибок, не повторяясь.

Взгляните на эти два проблемных участка кода:

```
pid_t pid = fork();
if (pid == -1) {
  fprintf(stderr, "Не могу клонировать процесс: %s\n", strerror(errno));
  return 1;
}

if (execle(...) == -1) {
  fprintf(stderr, "Не могу запустить скрипт: %s\n", strerror(errno));
  return 1;
}
```

Дублирование кода может стать для программиста причиной неоправданного стресса.

Есть ли способ избавиться от повторяющихся блоков кода? **Ну конечно же, есть!** Создавая функции, которые работают по принципу «написал и забыл», вы можете больше не беспокоиться о дублировании кода.

Вы спросите, о чем это мы? Как мы справимся с проблемным оператором return? Мы ведь не можем поместить **это** в функцию?

А нам и не нужно! Системный вызов exit() — это самый быстрый способ остановить работу программы. Больше нет нужды переживать о возвращении к функции main() — просто вызовите exit() и забудьте о вашей программе!

Вот как это работает. Сначала вынесите весь свой код для проверки ошибок в отдельную функцию под названием error(), после чего замените сложный оператор return системным вызовом exit().

Чтобы получить доступ к системному вызову exit, нужно подключить stdlib.h

```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

exit(1) НЕМЕДЛЕННО завершит вашу программу со статусом 1

Теперь вы можете заменить тот запутанный код для проверки ошибок чем-то более простым:

```
pid_t pid = fork();
if (pid == -1) {
  error("Не могу клонировать процесс");
}

if (execle(...) == -1) {
  error("Не могу запустить скрипт");
}
```

Предупреждение: во время выполнения программы вызывать exit() можно один раз. Не применяйте эту функцию, если вас пугает внезапное завершение программы.

Наточите свой карандаш

Эта программа сохраняет вывод скрипта *rssgossip.py* в файл с названием *stories.txt*. Она похожа на программу newshound, за исключением того, что производит поиск только по одному RSS-потоку. Используя свои знания о таблицах дескрипторов, попробуйте подобрать пропущенные фрагменты кода, в которых **стандартный вывод** дочернего процесса перенаправляется в файл *stories.txt*.

Директивы #include и функция error() были убраны для экономии места.

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  FILE *f = fopen("stories.txt", "w");
  if (!f) {        <— Если запись в stories.txt запрещена, f будет равняться нулю.
    error("Не могу открыть stories.txt");   <— Мы будем сообщать об ошибках
  }                                             с помощью функции error(),
                                                которую написали ранее.
  pid_t pid = fork();
  if (pid == -1) {
    error("Не могу клонировать процесс");
  }
  if (!pid) {                    <— Как вы думаете, что нужно сюда вписать?
    if (.........................................) {
      error("Не могу перенаправить стандартный вывод");
    }
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                     phrase, NULL, vars) == -1) {
      error("Не могу запустить скрипт");
    }
  }
  return 0;
}
```

newshound2.c

Наточите свой карандаш Решение

Эта программа сохраняет вывод скрипта *rssgossip.py* в файл с названием *stories.txt*. Она похожа на программу newshound, за исключением того, что производит поиск только по одному RSS-потоку. Используя свои знания о таблицах дескрипторов, вам нужно было подобрать пропущенные фрагменты кода, в которых **стандартный вывод** дочернего процесса перенаправляется в файл *stories.txt*.

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  FILE *f = fopen("stories.txt", "w");   <- Здесь stories.txt открывается для записи.
  if (!f) {   <- Если f равно нулю, значит мы не смогли открыть файл.
    error("Не могу открыть stories.txt");
  }
  pid_t pid = fork();
  if (pid == -1) {
    error("Не могу клонировать процесс");
  }
  if (!pid) {   <- Этот код изменяет дочерний процесс, потому что pid равен нулю.
    if (dup2(fileno(f), 1) == -1) {   <- Здесь дескриптор № 1 перенаправляется в файл stories.txt.
      error("Не могу перенаправить стандартный вывод");
    }
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                    phrase, NULL, vars) == -1) {
      error("Не могу запустить скрипт");
    }
  }
  return 0;
}
```

newshound2.c

Вы нашли правильный ответ? Программа изменит таблицу дескрипторов дочернего скрипта следующим образом:

#	Поток данных
0	Клавиатура
1	Файл stories.txt
2	Экран
3	Файл stories.txt

Это значит, что данные, которые скрипт *rssgossip.py* отправляет в стандартный вывод, должны появляться в файле *stories.txt*.

Тест-драйв

Вот что произойдет, когда вы скомпилируете и запустите программу:

Здесь программа запускается.

Здесь выводится содержимое файла stories.txt.

Если у вас компьютер с операционной системой Windows, вы должны использовать Cygwin.

```
> ./newshound2 'pajama death'
> cat stories.txt
Бывший барабанщик Pajama Death делится откровениями.
Новый альбом Pajama Death должен выйти в следующем месяце.
```

Новости сохранены в файле stories.txt.

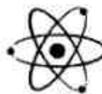
Что произошло?

Когда программа открыла файл *stories.txt* с помощью `fopen()`, операционная система зарегистрировала указатель `f` в таблице дескрипторов. При этом выражение `fileno(f)` использовалось в качестве номера дескриптора. Функция `dup2()` перенаправила стандартный вывод дескриптора № 1 в тот же файл.

Мне кажется, с этой программой могут возникнуть проблемы. Смотрите, я только что попробовал сделать то же самое на своем компьютере, и файл оказался пустым. Что произошло?

В файле нет данных? Почему?!?

Где же факты?



```
> ./newshound2 'pajama death'
> cat stories.txt
>
```

Сила мозга

Если предположить, что в потоке были новости, которые мы искали, почему после завершения работы программы файл *stories.txt* оказался пустым?

Иногда нужно подождать

Программа newshound2 порождает отдельный процесс для запуска скрипта *rssgossip.py*. Но после того как дочерний процесс был создан, он становится **независимым** от своего родителя. Вы могли бы запустить программу newshound2 еще раз и все равно получить пустой файл *stories.txt*, просто потому что *rssgossip.py* не успел завершить свою работу. Следовательно, операционная система должна дать вам возможность **подождать** завершения дочернего процесса.

Функция waitpid()

Функция `waitpid()` не вернет значение, пока не завершится дочерний процесс. Следовательно, вы можете добавить в свою программу немного кода, чтобы она не закрывалась, пока скрипт *rssgossip.py* не выполнит свою работу:

Вам нужно подключить заголовок sys/wait.h

```
#include <sys/wait.h>
```

Этот новый код нужно вставить в конец программы newshound2.

```
  int pid_status;
  if (waitpid(pid, &pid_status, 0) == -1) {
    error("Ошибка во время ожидания дочернего
процесса");
  }
  return 0;
}
```

Эта переменная используется для хранения информации о процессе.

Это указатель на int.

Вы можете добавлять сюда параметры

Идентификатор процесса.

newshound2.c

Подробнее о функции waitpid()

waitpid() принимает три параметра:

```
waitpid( pid, pid_status, options )
```

- **pid**
  Это идентификатор, который получает родительский процесс при клонировании дочернего.

- **pid_status**
  Здесь хранится *информация о завершении* процесса. Поскольку waitpid() будет ее обновлять, это должен быть указатель.

- **options**
  Есть несколько параметров, которые вы можете передать в waitpid(). Больше информации можно получить, набрав man waitpid. Если установить options равным **0**, то функция будет ждать, пока процесс завершится.

Что такое pid_status?

Когда функция waitpid() завершается, она сохраняет внутри pid_status данные о том, как отработал процесс. Чтобы получить *код завершения* дочернего процесса, вам нужно пропустить значение pid_status через макрос **WEXITSTATUS()**:

```
if (WEXITSTATUS(pid_status))
  puts("Код ошибки не равен нулю");
```

Если код завершения не равен нулю

Зачем нужен макрос? pid_status содержит несколько фрагментов информации, и только 8 битов описывают код завершения. Макрос возвращает вам эти самые 8 битов.

Тест-драйв

Теперь, когда вы запустите программу `newshound2`, она сначала проверит, не закончил ли работать скрипт *rssgossip.py*, а только потом завершится сама:

Теперь файл stories.txt будет содержать новости еще во время работы newshound2.

```
File Edit Window Help ReadAllAboutIt
> ./newshound2 'pajama death'
> cat stories.txt
Бывший барабанщик Pajama Death делится откровениями.
Новый альбом Pajama Death должен выйти в следующем месяце.
```

Отличино! Отныне я никогда не пропущу очередные новости.

Добавить `waitpid()` в программу было довольно легко, и это сделало ее более надежной. Прежде вы не могли быть уверены, что подпроцесс завершил запись, поэтому программу `newshound2` нельзя было рассматривать в качестве инструмента. Вы не могли использовать ее в скриптах, а также создать вокруг нее надстройку с графическим пользовательским интерфейсом.

Перенаправление ввода и вывода, ожидание процессами друг друга — это все простые виды **межпроцессного взаимодействия**. Когда процессы могут работать вместе, обмениваясь данными или ожидая друг друга, они становятся значительно мощнее.

КЛЮЧЕВЫЕ МОМЕНТЫ

- `exit()` — быстрый способ завершить программу.
- Все открытые файлы записываются в таблицу дескрипторов.
- Вы можете перенаправлять ввод и вывод, внося изменения в таблицу дескрипторов.
- `fileno()` находит дескриптор в таблице.
- `dup2()` можно использовать для изменения таблицы дескрипторов.
- `waitpid()` будет ждать, пока процесс завершится.

не бывает Глупых Вопросов

В: Завершить программу с помощью `exit()` быстрее, чем просто выйти из функции `main()`?

О: Нет. Но при вызове `exit()` вам не нужно структурировать свой код для возвращения в `main()`. Программа завершится, как только функция `exit()` будет вызвана.

В: Должен ли я проверять, не вернула ли функция `exit()` -1 на случай, если она не сработала?

О: Нет. Функция `exit()` не возвращает значение, потому что она никогда не завершается неудачей. Это единственная функция, которая абсолютно точно никогда ничего не возвращает и всегда отрабатывает успешно.

В: Число, которое я передаю в `exit()`, — это код завершения?

О: Да.

В: Всегда ли стандартные ввод, вывод и поток ошибок хранятся в таблице дескрипторов в строках 0, 1 и 2?

О: Да, всегда.

В: Значит, если я открою новый файл, он автоматически добавится в таблицу дескрипторов?

О: Да.

В: По какому принципу он будет помещен в таблицу?

О: Новый файл всегда добавляется в свободную строку с наименьшим номером. Таким образом, если первая доступная строка имеет номер 4, то именно она будет использована для нового файла.

В: Насколько велика таблица дескрипторов?

О: Она содержит от 0 до 255 строк.

В: Таблица дескрипторов выглядит довольно запутанно. Зачем она нужна?

О: Она позволяет влиять на работу программ. Без нее нельзя было бы делать перенаправление.

В: Есть ли способ отправить данные на экран, минуя стандартный вывод?

О: Да, в некоторых системах. Например, на компьютерах с Unix-подобной операционной системой данные на терминал можно вывести, открыв */dev/tty*.

В: Я могу использовать `waitpid()` для ожидания любых процессов или только тех, что запущены мною?

О: Вы можете использовать `waitpid()` для ожидания любого процесса.

В: Почему `pid_status` в `waitpid(..., &pid_status, ...)` не является обычным кодом завершения?

О: Потому что `pid_status` содержит и другую информацию.

В: Например?

О: Например, `WIFSIGNALED(pid_status)` вернет false, если процесс завершился естественным образом, и true, если его что-то прервало.

В: Почему целочисленная переменная, такая как `pid_status`, содержит несколько фрагментов информации?

О: Она хранит разную информацию в разных битах. Первые 8 битов описывают код завершения, далее идут другие сведения.

В: Значит, если я сам могу извлечь из `pid_status` первые 8 битов, мне не нужно использовать `WEXITSTATUS()`?

О: В любом случае лучше использовать макрос `WEXITSTATUS()`. Его легче прочитать, и он работает с типом `int` стандартного для платформы размера.

В: Почему название `WEXITSTATUS()` состоит из заглавных букв?

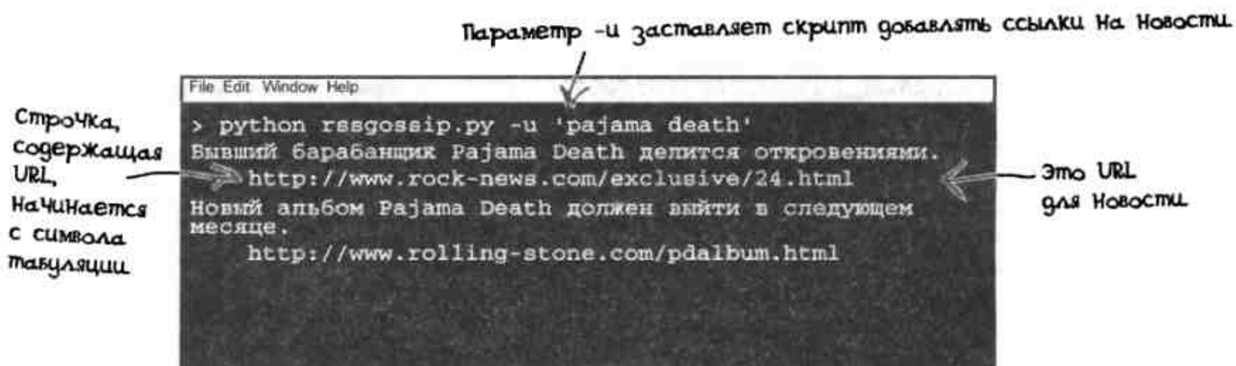
О: Потому что это макрос, а не функция. При выполнении программы компилятор заменяет макросы небольшими фрагментами кода.

Поддерживайте связь со своим детищем

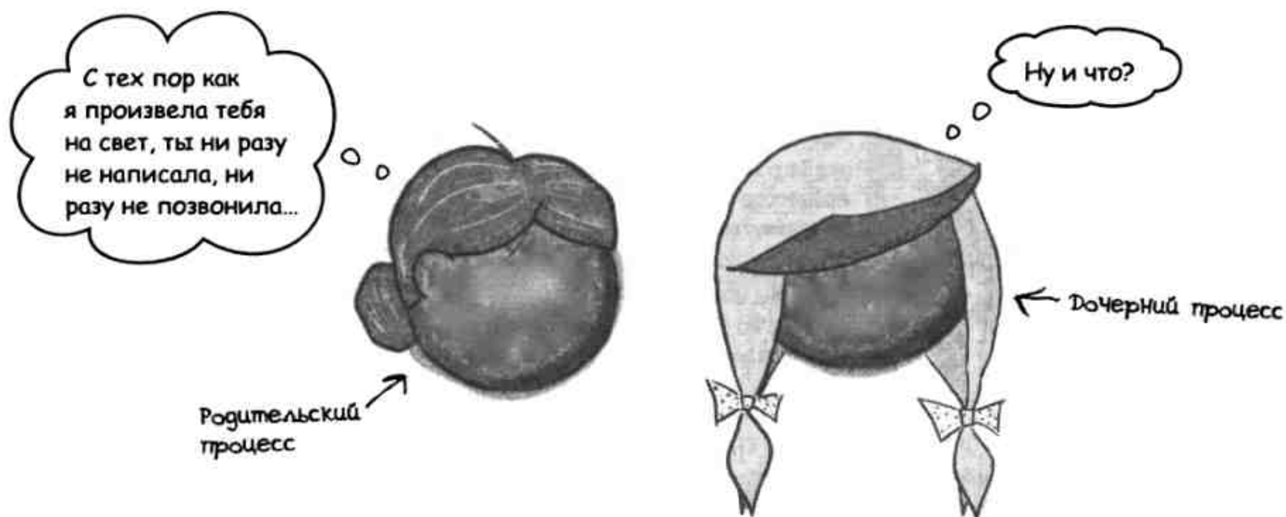
Вы уже видели, как запускать отдельные процессы с помощью `exec()` и `fork()`, и знаете, как перенаправлять вывод дочернего процесса в файл. Но что если вы захотите общаться с дочерним процессом напрямую? Возможно ли это? Раньше вы ждали, пока он запишет все данные в файл, чтобы потом его прочитать. Можно ли запустить процесс и считывать данные, которые он генерирует, ***в режиме реального времени***?

Считывание ссылок на новости из программы rssgossip

В качестве примера используем параметр скрипта *rssgossip.py*, который позволяет выводить URL для любой найденной новости:



Теперь вы *могли бы* запустить скрипт и сохранить его вывод в файл, но это займет много времени. Было бы намного лучше, если бы родительский и дочерний процессы могли общаться друг с другом, пока дочерний все еще работает.



Соединяйте свои процессы с помощью каналов

Вы уже использовали кое-что для установления связей между процессами. Это были каналы (pipes).

Два процесса соединены с помощью канала.

grep фильтрует вывод скрипта.

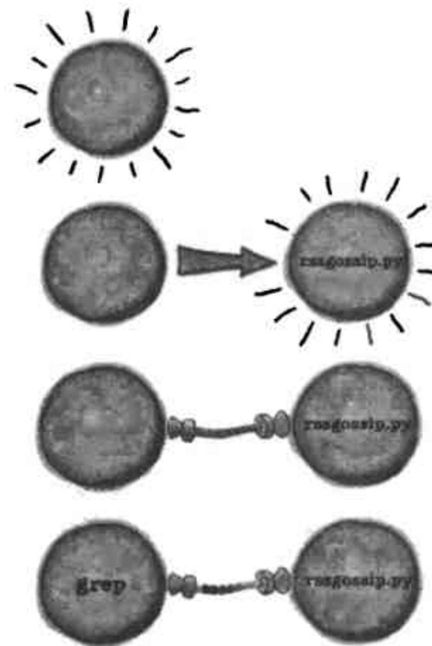
rssgossip.py отправляет свой вывод в канал.

```
File Edit Window Help ReadAllAboutIt
python rssgossip.py -u 'pajama death' | grep 'http'
    http://www.rock-news.com/exclusive/24.html
    http://www.rolling-stone.com/pdalbum.html
```

В командной строке каналы используются для связывания **вывода** одного процесса с **вводом** другого. В приведенном примере вы вручную запускаете *rssgossip.py*, после чего передаете его вывод команде **grep**. И уже эта команда ищет все строки, содержащие **http**.

У команд, связанных каналом, возникает иерархия

Когда два приложения связываются в командной строке с помощью канала, один из них выступает родительским процессом, а другой — дочерним. Следовательно, в вышеприведенном примере команда grep является **родителем** по отношению к скрипту *rssgossip.py*.



1. В командной строке создается родительский процесс.
2. Родительский процесс клонирует скрипт rssgossip.py в дочерний процесс.
3. Родительский процесс подключает вывод дочернего процесса к своему вводу, используя канал.
4. Родительский процесс запускает команду grep.

Каналы часто используются в командной строке: с их помощью пользователи могут связывать процессы вместе. Но что если вы не хотите выходить за рамки кода на языке Си? Как подключить канал к дочернему процессу, чтобы считывать его вывод по мере формирования?

Практический пример: открытие новостей в браузере

Допустим, вы хотите, чтобы скрипт *rssgossip.py* открывал найденные новости в браузере. Ваша программа будет работать в родительском процессе, а *rssgossip.py* — в дочернем. Вам нужно создать канал, связывающий вывод *rssgossip.py* с вводом вашей программы.

Но как создаются каналы?

Функция pipe() открывает два потока данных

Поскольку дочерний процесс станет отправлять данные в родительский, необходим канал, который будет связывать стандартный вывод первого процесса со стандартным вводом второго. Вы будете создавать каналы с помощью функции **`pipe()`**. Помните, мы говорили, что любой поток, например к файлу, добавляется в таблицу дескрипторов при открытии? Именно этим и занимается функция `pipe()`: она создает два связанных потока и добавляет их в таблицу. То, что записывается в один поток, можно прочитать из другого.

#	Поток данных
0	Стандартный ввод
1	Стандартный вывод
2	Стандартный поток ошибок
3	Конец канала для чтения
4	Конец канала для записи

Это fd[0]. →

Это fd[1]. →

Эти дескрипторы создаются при вызове pipe().

Все, что записывается сюда...

...может быть считано отсюда.

Создавая две новые строки в таблице, функция `pipe()` сохраняет соответствующие файловые дескрипторы в массиве из двух элементов:

Дескрипторы будут сохранены в этом массиве.

Имя массива передается в функцию pipe().

```
int fd[2];
if (pipe(fd) == -1) {
  error("Не могу создать канал");
}
```

Команда `pipe()` создает два канала и передает вам два дескриптора: `fd[1]`, который производит **запись** в канал, и `fd[0]`, который из него **считывает**. Вы должны использовать полученные дескрипторы в родительском и дочернем процессах.

fd[1] записывает в канал, fd[0] считывает из него.

Дочерний процесс

В дочернем процессе вы должны **закрыть** конец канала, связанный с `fd[0]`. Затем нужно перенаправить стандартный вывод дочернего процесса в тот же поток, в который направлен дескриптор `fd[1]`.

Дочерний процесс не будет считывать из канала.

Этим мы закрываем конец канала, предназначенный для чтения.

```
close(fd[0]);
dup2(fd[1], 1);
```

Затем дочерний процесс подключает свой стандартный вывод к концу, предназначенному для записи.

#	Поток данных
0	Стандартный ввод
1	~~Стандартный вывод~~ Конец канала для записи
2	Стандартный поток ошибок
3	~~Конец канала для чтения~~
4	Конец канала для записи

Это fd[0] – конец канала, предназначенный для чтения.

Дочерний процесс не будет считывать из канала...

...Но будет в него записывать.

Это fd[1] – конец канала, предназначенный для записи.

Таким образом, все, что дочерний процесс отправляет в стандартный вывод, будет записано в канал.

Родительский процесс

В родительском процессе вам нужно закрыть конец канала, связанный с `fd[1]` (поскольку вы не будете в него записывать). Затем необходимо перенаправить стандартный ввод этого процесса для считывания данных из того места, на которое указывает дескриптор `fd[0]`:

fd[0] – конец канала, предназначенный для чтения.

Родительский процесс подключает свой стандартный ввод к концу, предназначенному для чтения.

```
dup2(fd[0], 0);
close(fd[1]);
```

Этим мы закрываем конец канала, предназначенный для записи.

#	Поток данных
0	~~Стандартный ввод~~ Конец канала для чтения
1	Стандартный вывод
2	Стандартный поток ошибок
3	Конец канала для чтения
4	~~Конец канала для записи~~

Родительский процесс будет считывать из канала...

...Но не будет в него записывать.

Все, что дочерний процесс запишет в канал, будет считываться через стандартный ввод родительского процесса.

Открытие веб-страницы в браузере

Вашей программе нужно будет открывать веб-страницы с помощью системного браузера. На первый взгляд, это довольно сложная задача, ведь разные операционные системы по-разному общаются с программами наподобие браузера.

К счастью, безработные актеры скооперировались и выдали некий кусок кода, который способен открывать веб-страницы на большинстве систем. Похоже, они куда-то спешили, поэтому результат получился довольно простой; не обошлось и без использования `system()`:

Код, готовый к выпечке

```
void open_url(char *url)
{
  char launch[255];
  sprintf(launch, "cmd /c start %s", url);
  system(launch);
  sprintf(launch, "x-www-browser '%s' &", url);
  system(launch);
  sprintf(launch, "open '%s'", url);
  system(launch);
}
```

Так открывается веб-страница в Windows.

Так открывается веб-страница в Linux.

Так открывается веб-страница в Mac.

В коде запускаются **три отдельные команды** для открытия URL: по одной для Mac, Windows и Linux. Две из них всегда будут завершаться неудачей, но достаточно срабатывания и одной команды.

Сходим с проложенной лыжни

Считаете, что вы программируете лучше, чем те безработные актеры? Тогда почему бы вам не переделать код для вашей любимой операционной системы, используя функции `fork()` и `exec()`?

Упражнение

Похоже, большая часть программы уже написана. Все, что нужно, — дописать код для подключения *родительских* и *дочерних* процессов к каналу. Чтобы сэкономить место, мы убрали директивы `#include`, а также функции `error()` и `open_url()`. Не забывайте, что в этой программе *дочерний* процесс будет «разговаривать» с *родительским*, поэтому убедитесь, что каналы подключены правильно.

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  int fd[2];
  ..........................................................
  ..........................................................
  ..........................................................
  pid_t pid = fork();
  if (pid == -1) {
    error("Не могу клонировать процесс");
  }
  if (!pid) {

    ..........................................................
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                  "-u", phrase, NULL, vars) == -1) {
      error("Не могу запустить скрипт");
    }
  }
  ..........................................................
  ..........................................................
  char line[255];
  while (fgets(line, 255, ..............................)) {
    if (line[0] == '\t')
      open_url(line + 1);
  }
  return 0;
}
```

Возможно, вы захотите заменить этот RSS-поток на какой-то другой (→ строка `char *vars[]`).

Этот массив будет хранить дескрипторы для вашего канала (→ `int fd[2];`).

Здесь создается ваш канал (→ пустые строки после `int fd[2];`).

Это родительский или дочерний процесс? Какой код нужно вписать в эти строки? (→ `if (!pid) {`)

-u заставляет скрипт выводить URL для новостей.

Вы находитесь в родительском или дочернем процессе? Что нужно сделать с каналом?

Что нужно сюда вписать? Откуда вы будете считывать данные? (→ аргумент `fgets`)

Если строчка начинается табуляцией...

...значит это URL

line + 1 — это строка, следующая за символом табуляции

news_opener.c

Упражнение Решение

Похоже, большая часть программы уже написана. Вам нужно было дописать код для подключения *родительских* и *дочерних* процессов к каналу. Чтобы сэкономить место, мы убрали директивы `#include`, а также функции `error()` и `open_url()`.

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  int fd[2];
    if (pipe(fd) == -1) {
      error("Не могу создать канал");
    }
  pid_t pid = fork();
  if (pid == -1) {
    error("Не могу клонировать процесс");
  }
  if (!pid) {
    dup2(fd[1], 1);
    close(fd[0]);
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                  "-u", phrase, NULL, vars) == -1) {
      error("Не могу запустить скрипт");
    }
  }
  dup2(fd[0], 0);
  close(fd[1]);
  char line[255];
  while (fgets(line, 255, stdin)) {
    if (line[0] == '\t')
      open_url(line + 1);
  }
  return 0;
}
```

Комментарии к коду:

- `int fd[2];` — Здесь создается канал, чьи дескрипторы сохраняются в fd[0] и fd[1].
- `if (pipe(fd) == -1) { error(...); }` — Нужно проверить этот код возврата на случай, если мы не смогли создать канал.
- `if (!pid) {` — Это ветка для дочернего процесса.
- `dup2(fd[1], 1);` — Здесь стандартный вывод направляется в конец канала, выделенного для записи.
- `close(fd[0]);` — Дочерний процесс не будет считывать из канала, поэтому мы закрываем конец, предназначенный для чтения.
- После `}` — Это ветка для родительского процесса.
- `dup2(fd[0], 0);` — Вы перенаправите стандартный ввод в конец канала, выделенного для чтения.
- `close(fd[1]);` — Здесь конец канала, предназначенного для чтения, будет закрыт, потому что родительский процесс не будет в него записывать.
- `stdin` — Считываем стандартный ввод, так как он подключен к каналу. ← Вы также могли бы подставить сюда fd[0].

news_opener.c

Тест-драйв

Когда вы скомпилируете и запустите свой код, произойдет следующее:

Отлично, код работает.

Программа `news_opener` запустила в отдельном процессе скрипт *rssgossip.py* и заставила его вывести URL для каждой найденной новости. Весь экранный вывод был перенаправлен в **канал**, подключенный к родительскому процессу `news_opener`. Это значит, что `news_opener` мог произвести поиск любого URL и затем открыть его в браузере.

Программа открывает в браузере все новости, которые может найти

Каналы отлично подходят для связывания процессов. Теперь у вас есть возможность не только **запускать** программы и **управлять** их окружением, но и **считывать их вывод**. Это открывает перед вами огромные возможности. Теперь вы способны контролировать ***любую программу***, запущенную в командной строке, с помощью кода на Си.

Сходим с проложенной лыжни

Теперь, когда вы знаете, как управлять скриптом *rssgossip.py*, почему бы вам не проделать то же самое и с другими программами? Следующие утилиты есть на любой Unix-подобной системе и на любом компьютере с Windows и Cygwin.

curl/wget
Эти программы позволяют общаться с веб-серверами. Используя их в своем коде, вы можете писать программы, взаимодействующие со Всемирной паутиной.

mail/mutt
Эти программы позволяют отправлять электронную почту из командной строки. Если они есть на вашем компьютере, вы можете отправлять письма из созданных вами программ на языке Си.

convert
Эта команда может преобразовать один графический формат в другой. Почему бы вам не создать программу на Си, которая будет выводить графики в текстовом формате SVG, а затем конвертировать их в PNG-изображения?

не бывает Глупых Вопросов

В: Канал — это файл?

О: Это зависит от операционной системы, но каналы, созданные с помощью функции `pipe()`, обычно файлами не являются.

В: Значит, канал *может* быть файлом?

О: Существуют каналы, основанные на файлах. Как правило, их называют *именованными*, или *FIFO-файлами* (от английского first-in/first-out — «первым пришел/первым ушел»).

В: Зачем могут понадобиться каналы на основе файлов?

О: У таких каналов есть имена. Они могут быть полезны в ситуации, когда два общающихся между собой процесса не являются родительским и дочерним. Пока им известно имя канала, они могут взаимодействовать.

В: Отлично! А как использовать эти именованные каналы?

О: С помощью системного вызова `mkfifo()`. Чтобы узнать о нем больше, перейдите по адресу *http://tinyurl.com/cdf6ve5*.

В: Если большинство каналов не являются файлами, что же они из себя представляют?

О: Как правило, это просто фрагменты памяти. Данные записываются в одной точке, а считываются в другой.

В: Что произойдет, если я попытаюсь считать данные из канала, в котором ничего нет?

О: Ваша программа будет ждать, пока в нем что-то появится.

В: Как родительский процесс узнает, когда завершается дочерний?

О: При завершении дочернего процесса закрывается канал. Команда `fgets()` получает символ конца файла (EOF) и возвращает 0, цикл заканчивается.

В: Могут ли родительские процессы обращаться к дочерним?

О: Безусловно. Вам ничто не мешает подключить каналы наоборот, чтобы данные поступали к дочернему процессу.

В: Можно ли получить канал, который работает в обоих направлениях? Так мои родительские и дочерние процессы могли бы вести диалог.

О: Нет, это невозможно. Каналы всегда работают только в одном направлении. Но вы можете создать два канала: один от родительского процесса к дочернему, а второй — от дочернего к родительскому.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Родительские и дочерние процессы могут общаться с помощью каналов.
- Функция `pipe()` создает канал и два дескриптора.
- Дескрипторы нужны для чтения и записи на обоих концах канала.
- Вы можете перенаправить стандартные ввод и вывод в канал.
- Родительские и дочерние процессы используют противоположные концы канала.

Смерть процесса

Вы уже видели, как процессы создаются, как общаются между собой и как сконфигурированы их окружения. Но что насчет того, как они умирают? К примеру, если ваша программа считывает данные с клавиатуры и пользователь нажмет Ctrl+C, то программа прекратит свою работу.

Как это происходит? По тексту на экране видно, что программа никогда не доберется до второй функции `printf()`, значит, Ctrl+C останавливает не только команду `fgets()` — перестает выполняться вся программа целиком. Что же случилось? Операционная система выгрузила приложение или, может, функция `fgets()` сделала вызов `exit()`?

```
#include <stdio.h>

int main()
{
  char name[30];
  printf("Введите свое имя: ");
  fgets(name, 30, stdin);
  printf("Привет, %s\n", name);
  return 0;
}
```

```
> ./greetings
Введите свое имя: ^C
>
```

Если вы нажмете Ctrl+C, программа перестанет работать. Но почему?

Операционные системы контролируют программы с помощью сигналов

Вся магия происходит на уровне операционной системы. Когда вы вызываете функцию `fgets()`, она считывает данные с клавиатуры. Видя, что пользователь нажал Ctrl+C, операционная система шлет программе сигнал о прерывании процесса.

Эй! Он нажал Ctrl+C. Запускай свой обработчик прерывания.

Сигнал — это просто короткое сообщение, одиночное целочисленное значение. Процесс должен обработать поступивший сигнал, бросив все, чем он до этого занимался. Процесс сверяется с таблицей, в которой каждому *сигналу соответствует* определенная функция — **обработчик сигнала**. Стандартным обработчиком для сигнала прерывания является функция `exit()`.

Так почему же операционная система просто не завершила процесс? Дело в том, что при получении процессом сигнала вы можете запустить *свой собственный код* с помощью этой таблицы.

Таблица соответствий для сигналов.

Сигнал	Обработчик
SIGURG	Ничего не делать
SIGINT	Вызвать exit()

Это сигнал прерывания.

SIGINT имеет значение 2.

Стандартный обработчик вызывает exit().

Перехват сигналов для запуска собственного кода

Иногда при получении сигнала прерывания необходимо запустить собственный код. Например, прежде чем завершить работу, процессу может понадобиться закрыть файлы и сетевые соединения, почистить кэш. Но как сообщить компьютеру, что в ответ на его сигнал нужно запускать ваш код? Вы можете сделать это с помощью структуры **sigaction**.

sigaction — «обертка» для функции

sigaction — это *структура*, которая содержит указатель на функцию. Ее используют, чтобы сообщить операционной системе, какой код она должна вызвать в ответ на отправленный сигнал. Следовательно, если вы хотите, чтобы при получении вашим процессом сигнала о *прерывании* операционная система вызвала функцию diediedie(), вы должны обернуть эту функцию в структуру sigaction.

Вот как создается sigaction:

```
struct sigaction action;
action.sa_handler = diediedie;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
```

Создаем новое действие.

Это имя функции, которую должен вызвать компьютер.

Функция, для которой sigaction служит оберткой, называется обработчиком.

Это дополнительные параметры. Вы можете просто присвоить им ноль.

С помощью шаблона sa_mask отфильтровываются сигналы, которые будут обрабатываться структурой sigaction.

Чаще всего вы будете использовать пустой шаблон, как в данном случае.

Функция внутри sigaction называется **обработчиком**, потому что с ее помощью будет *обрабатываться* отосланный сигнал. Если вы хотите создать обработчик, вам нужно написать его определенным образом.

Все обработчики принимают сигнал в качестве аргумента

Сигналы — это просто целочисленные значения. Функция, созданная вами для обработки сигнала, должна принимать аргумент типа int:

```
void diediedie(int sig)
{
  puts ("Прощай, жестокий мир....\n");
  exit(1);
}
```

Это код сигнала, перехватываемого обработчиком.

Можно использовать один и тот же обработчик для нескольких сигналов или несколько отдельных обработчиков для одного сигнала — выбор за вами.

Предполагается, что обработчики должны быть лаконичными и быстрыми. Их задача заключается *ровно в том*, чтобы отреагировать на полученный сигнал.

Будьте осторожны при использовании в обработчиках стандартного вывода и стандартного потока ошибок.

Несмотря на то что в коде, приведенном для примера, мы направляем текст в стандартный вывод, вам следует быть осторожным, проделывая то же в более сложных программах. Сигналы могут возникать вследствие того, что с программой случилось нечто плохое, а это может означать, что стандартный вывод недоступен. Будьте внимательны.

sigaction регистрируется с помощью функции sigaction()

После того как вы создали структуру `sigaction`, вам нужно сообщить о ней операционной системе. Это делается с помощью функции **`sigaction()`**:

```
sigaction(signal_no, &new_action, &old_action);
```

`sigaction()` принимает три параметра:

- **Код сигнала.**
  Это целочисленное значение сигнала, который вы хотите обработать. Как правило, вы будете получать одно из стандартных обозначений — `SIGINT` или `SIGQUIT`. ← Со временем вы узнаете больше о стандартных сигналах.

- **Новое действие.**
  Это **адрес** новой структуры `sigaction`, которую вы хотите зарегистрировать.

- **Старое действие.**
  Если вы передадите указатель на другую структуру `sigaction`, она будет заполнена информацией о *текущем* обработчике, которого вы собираетесь заменить. Если вас не интересует существующий обработчик сигнала, можете установить этот параметр равным `NULL`.

В случае неудачи функция `sigaction()` вернет -1 и задаст значение для функции `errno`. Чтобы сохранить краткость, в некоторых случаях мы будем пропускать в коде проверку на ошибки, однако вы ***никогда*** не должны этого делать.

Код, готовый к выпечке

Благодаря этому коду вам будет легче регистрировать функции в качестве обработчиков сигналов:

```
int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

Это код сигнала. (→ sig)

Указатель на функцию-обработчик. (→ handler)

Создаем действие. (→ struct sigaction action;)

Присваиваем обработчику действия функцию, которую мы передали. (→ action.sa_handler = handler;)

Используем пустой шаблон. (→ sigemptyset)

Возвращаем значение sigaction(), чтобы сделать проверку на ошибки.

Этот код позволит вам регистрировать обработчик, указывая код сигнала и имя функции.

```
catch_signal(SIGINT, diedieie)
```

Перепишем код с использованием обработчика сигнала

Сейчас у вас есть весь необходимый код, чтобы отреагировать на нажатие клавиш Ctrl+C:

```
#include <stdio.h>
#include <signal.h>   <- Вам нужно подключить заголовок signal.h
#include <stdlib.h>
                      Это наш новый обработчик сигнала.
void diediedie(int sig)   <- Операционная система передает сигнал в обработчик.
{
  puts ("Прощай, жестокий мир...\n");
  exit(1);
}
                      Это функция для регистрации обработчика.
int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
                      SIGINT означает, что мы получили сигнал о прерывании
int main()            Здесь сигнал о прерывании передается обработчику handle_interrupt().
{
  if (catch_signal(SIGINT, handle_interrupt) == -1) {
    fprintf(stderr, "Не могу подобрать обработчик");
    exit(2);
  }
  char name[30];
  printf("Введите свое имя: ");
  fgets(name, 30, stdin);
  printf("Привет, %s\n", name);
  return 0;
}
```

Обработчики имеют возвращаемый тип void.

Программа запросит имя пользователя и подождет, пока тот будет вводить текст. Но если вместо этого пользователь нажмет Ctrl+C, операционная система автоматически пошлет процессу *сигнал о прерывании* (`SIGINT`). Этот сигнал будет перехвачен структурой `sigaction`, зарегистрированной с помощью функции `catch_signal()`. `sigaction` содержит указатель на функцию `diediedie()`, которая в итоге и будет вызвана. Таким образом, программа выведет сообщение и вызовет `exit()`.

Тест-драйв

Когда вы запустите новую версию программы и нажмете Ctrl+C, произойдет следующее:

```
File Edit Window Help
> ./greetings
Введите свое имя: ^CПрощай, жестокий мир…
>
```

Операционная система получит комбинацию Ctrl+C и пошлет процессу сигнал `SIGINT`. Процесс запустит ***вашу*** функцию `handle_interrupt()`.

Попробуйте догадаться

Операционная система может посылать вашему процессу множество разных сигналов. Соедините каждый сигнал с причиной его возникновения.

`SIGINT`	Процесс был прерван.
`SIGQUIT`	Окно терминала изменило размер.
`SIGFPE`	Процесс попытался обратиться к недоступному участку памяти.
`SIGTRAP`	В ядро поступил запрос о завершении процесса.
`SIGSEGV`	Процесс сделал запись в канал, из которого никто не читает.
`SIGWINCH`	Ошибка при операции с плавающей точкой.
`SIGTERM`	Процесс попросили остановиться и записать дамп памяти в файл.
`SIGPIPE`	Отладчик запрашивает местоположение процесса.

Попробуйте догадаться. Решение

Операционная система может посылать вашему процессу множество разных сигналов. Вам нужно было соединить каждый сигнал с причиной его возникновения.

не бывает Глупых Вопросов

В: Если обработчик сигнала о прерывании не вызовет `exit()`, программа все равно завершится?

О: Нет.

В: Значит, я могу написать программу, которая полностью игнорирует сигналы о прерывании?

О: Да, вы можете это сделать, но это не очень хорошая идея. В целом, если ваша программа получает сигнал об ошибке, лучше ее завершить с ошибкой, даже если сначала нужно выполнить свой код.

Чтобы посылать сигналы, используйте команду kill

Допустим, вы написали код для обработки сигналов, как его протестировать? К счастью, в Unix-подобных системах существует команда **kill**. Она имеет такое название, потому что обычно ее используют, чтобы «убивать» процессы (в переводе с английского kill — «убивать»).В действительности команда `kill` просто шлет процессу сигнал — по умолчанию это `SIGTERM`. Однако вы можете посылать с ее помощью любой сигнал на свое усмотрение.

Включая Cygwin в Windows.

Чтобы посмотреть, как это работает, откройте *два терминала*. В одном вы можете запустить свою программу, а во втором будете посылать ей сигналы с помощью команды `kill`:

ps выводит ваши текущие процессы.

```
File Edit Window Help
> ps
77868 ttys003    0:00.02 bash
78222 ttys003    0:00.01 ./testprog
> kill 78222
> kill -INT 78222
> kill -SEGV 78222
> kill -KILL 78222
```

Здесь программе посылается сигнал SIGTERM.

Здесь программе посылается сигнал SIGINT.

Здесь программе посылается сигнал SIGSEGV.

Программа, которой мы хотим посылать сигналы 78222 — это ID процесса.

Здесь посылается сигнал SIGKILL, который не может быть проигнорирован.

Каждая из этих команд пошлет сигнал процессу и запустит зарегистрированный для него обработчик. Исключением служит сигнал **SIGKILL** — он не может быть перехвачен в коде программы, и его нельзя проигнорировать. Поэтому, если в вашей программе есть ошибка, из-за которой она не отвечает ни на один из сигналов, вы **всегда** можете остановить ее с помощью `kill -KILL`.

SIGSTOP тоже нельзя проигнорировать. Он используется для приостановки процессов.

kill -KILL <pid> всегда закрывает программу.

Посылаем сигналы с помощью raise()

Иногда может понадобиться, чтобы процесс послал себе сигнал сам. Это можно сделать с помощью команды `raise()`.

```
raise(SIGTERM);
```

Как правило, команда `raise()` используется внутри переопределенных обработчиков сигналов. Таким образом, получив сообщение о чем-то незначительном, вы можете сгенерировать более серьезный сигнал.

Это называется **эскалацией сигнала**.

Посылаем коду сигнал будильника

Операционная система шлет процессу сигналы о событиях, которые ему нужно знать. Возможно, пользователь попытался прервать процесс, или кто-то попробовал этот процесс «убить», или сам процесс сделал что-то такое, чего ему делать не следовало, например попытался обратиться к недоступному участку памяти.

Однако сигналы используются не только в случаях, когда что-то пошло не так. Иногда у процесса может возникнуть необходимость сгенерировать собственный сигнал, например **сигнал тревоги SIGALRM**. Как правило, он создается **таймером** процесса. Таймер похож на будильник: вы выставляете его на какое-то время, а ваша программа пока может заняться чем-нибудь другим.

Тик-так, тик-так, всего пара минут...

Здесь мы делаем так, чтобы таймер сработал через 120 секунд. →

```
alarm(120);
do_important_busy_work();
do_more_busy_work();
```

Тем временем ваш код делает что-то другое. →

Вызов alarm(120) устанавливает таймер на 120 секунд.

Даже несмотря на то что ваша **программа занята другими** делами, таймер продолжает **работать в фоновом режиме.** И когда наступит 120-я секунда…

...таймер пошлет сигнал SIGALRM

Когда процесс получает сигнал, он **останавливает остальную работу** и приступает к его обработке. Что же происходит с процессом в случае поступления сигнала тревоги? Он ***останавливает процесс***. Маловероятно, что вам понадобится «убить» свою программу по таймеру, поэтому чаще всего вы будете регистрировать обработчик для выполнения других задач.

Этот код перехватит сигнал с помощью функции, которую Вы создали ранее. →

```
catch_signal(SIGALRM, pour_coffee);
alarm(120);
```

Будьте осторожны!

Не используйте одновременно функции alarm() и sleep().

Функция `sleep()` *«усыпляет» программу на несколько секунд. Однако в своей работе она применяет тот же таймер, что и* `alarm()`*, поэтому при одновременном использовании эти функции будут мешать друг другу.*

Сигнал тревоги позволяет использовать **многозадачность**. `SIGALRM` отлично подходит для случаев, когда программе нужно ***прервать саму себя***, например необходимо каждые несколько секунд выполнять определенную задачу или ограничить время, выделяемое на выполнение какой-то работы.

Подробнее о сбросе и игнорировании сигналов

Вы уже видели, как задавать нестандартные обработчики для сигналов. Но что если вам понадобится восстановить изначальный обработчик? К счастью, в заголовке *signal.h* есть специальное обозначение **SIG_DFL** — ***«обработать стандартным образом» (handle it the default way).***

```
catch_signal(SIGTERM, SIG_DFL);
```

Хорошо, если я получу сигнал TERM, мне нужно будет, как и прежде, выполнить exit()...

Есть и другое обозначение — **SIG_IGN**. Оно говорит о том, что процесс должен полностью **игнорировать** сигнал.

```
catch_signal(SIGINT, SIG_IGN);
```

Ctrl+C? Читай по губам: я ничего не буду делать.

Но, решив проигнорировать сигнал, вы должны быть *очень осторожны*. Сигналы играют важную роль в управлении процессами и их остановке. Если вы будете их игнорировать, вашу программу станет сложнее остановить.

не бывает Глупых Вопросов

В: Можно ли установить сигнал тревоги на время меньше одной секунды?

О: Да, но это немного сложнее. Вам нужно будет использовать функцию `setitimer()`. Она позволяет задавать значения для таймера в секундах или долях секунд.

В: Как это сделать?

О: Перейдите по ссылке *http://tinyurl.com/3o7hzbm*.

В: Почему для процесса используется всего один таймер?

О: Таймером управляет ядро операционной системы. Если у каждого процесса будет множество таймеров, ядро станет работать все медленней и медленней. Чтобы этого избежать, операционная система ограничивается одним таймером для каждого процесса.

В: Таймеры позволяют использовать многозадачность?! Отлично, значит, с их помощью я могу выполнять много всего одновременно?

О: Нет. Не забывайте, что на время обработки сигнала ваш процесс всегда останавливает текущую работу. Следовательно, в любой момент времени он выполняет только одну задачу. Позже вы увидите, как действительно можно заставить свой код выполнять несколько задач одновременно.

В: Что случится, если я установлю таймер, который до этого уже был установлен?

О: Вызывая функцию `alarm()`, вы каждый раз сбрасываете таймер. Следовательно, если вы установите таймер сначала на 10 секунд, а затем на 10 минут, то таймер не сработает, пока не пройдет 10 минут. Изначальная установка на 10 секунд будет утеряна.

Большое упражнение

Это исходный код программы для проверки математических навыков пользователя. Программа задает вопросы о простейшем умножении чисел и следит за количеством правильных ответов. Она завершится в одном из двух случаев:

1) пользователь нажмет Ctrl+C;

2) пользователю понадобится **более пяти секунд**, чтобы ответить на вопрос.

Окончив работу, программа выводит на экран итоговый счет и устанавливает код завершения, равный 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int score = 0;

void end_game(int sig)
{
  printf("\nИтоговой счет: %i\n", score);
  ..........................................................
}

int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

Что должно произойти после того, как счет будет выведен на экран?

```
void times_up(int sig)
{
  puts("\nВРЕМЯ ВЫШЛО!");
  raise(.................................................);
}
```

Вызываем что?

```
void error(char *msg)
{
  printf("\nСколько будет %i умножить на %i? ", a, b);
  exit(1);
}

int main()
{
  catch_signal(SIGALRM,......................................);
  catch_signal(SIGINT, ......................................);
  srandom (time (0));
  while(1) {
    int a = random() % 11;
    int b = random() % 11;
    char txt[4];
    ..............................................................................
    printf("\nWhat is %i times %i? ", a, b);
    fgets(txt, 4, stdin);
    int answer = atoi(txt);
    if (answer == a * b)
      score++;
    else
      printf("\nНеправильно! Счет: %i\n", score);
  }
  return 0;
}
```

Что будут делать функции signal()?

Это гарантирует, что каждый раз вы будете получать разные случайные числа.

a и b будут случайными числами от 0 до 10.

Хм... Какую строчку мы пропустили? Нужно свериться с заданием...

Большое упражнение

Решение

Это исходный код программы для проверки математических навыков пользователя. Программа задает вопросы о простейшем умножении чисел и следит за количеством правильных ответов. Она завершится в одном из двух случаев:

1) пользователь нажмет Ctrl+C;

2) пользователю понадобится **более пяти секунд**, чтобы ответить на вопрос.

Окончив работу, программа выводит на экран итоговый счет и устанавливает код завершения, равный 0.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int score = 0;

void end_game(int sig)
{
  printf("\nИтоговой счет: %i\n", score);
  exit(0);
}

int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

Вам нужно установить код завершения, равный 0, и закрыть программу.

```
void times_up(int sig)
{
  puts("\nВРЕМЯ ВЫШЛО!");
  raise(SIGINT);
}

void error(char *msg)
{
  printf("\nСколько будет %i умножить на %i? ", a, b);
  exit(1);
}

int main()
{
  catch_signal(SIGALRM, times_up);
  catch_signal(SIGINT, end_game);
  srandom (time (0));
  while(1) {
    int a = random() % 11;
    int b = random() % 11;
    char txt[4];
    alarm(5);
    printf("\nWhat is %i times %i? ", a, b);
    fgets(txt, 4, stdin);
    int answer = atoi(txt);
    if (answer == a * b)
      score++;
    else
      printf("\nНеправильно! Счет: %i\n", score);
  }
  return 0;
}
```

Отправка SIGINT заставит программу вывести итоговый счет и запустить end_game().

Функции signal() будут регистрировать обработчики.

Это гарантирует, что каждый раз вы будете получать разные случайные числа.

Делаем так, чтобы сигнал тревоги отправился через 5 секунд.

Если цикл будет пройден менее чем за 5 секунд, таймер сбросится и ничего не сработает.

Тест-драйв

Чтобы увидеть, как работает программа, вам нужно скомпилировать и запустить ее несколько раз.

Тест 1: нажимаем Ctrl+C

При первом запуске ответьте на несколько вопросов, а затем нажмите Ctrl+C.

Ctrl+C пошлет процессу сигнал прерывания (`SIGINT`), чтобы программа вывела итоговый счет и завершилась с помощью `exit()`.

```
> ./math_master

Сколько будет 0 умножить на 1? 0

Сколько будет 6 умножить на 1? 6

Сколько будет 4 умножить на 10? 40

Сколько будет 2 умножить на 3? 6

Сколько будет 7 умножить на 4? 28

Сколько будет 4 умножить на 10? ^C
Итоговый счет: 5
>
```

Здесь пользователь нажал Ctrl+C.

Перед завершением программа отобразила итоговый счет.

Тест 2: ждем 5 секунд

В этот раз, отвечая на один из вопросов, вместо нажатия Ctrl+C мы подождем как минимум 5 секунд и посмотрим, что произойдет.

Сработает сигнал тревоги (`SIGALRM`). Программа ожидала ответ пользователя, но это заняло слишком много времени, поэтому таймер послал сигнал и процесс немедленно переключился на функцию-обработчик `times_up()`. Обработчик вывел сообщение **«ВРЕМЯ ВЫШЛО!»** и повысил сигнал до `SIGINT`. Это привело к тому, что программа отобразила итоговый счет.

```
> ./math master

Сколько будет 5 умножить на 9? 45

Сколько будет 2 умножить на 8? 16

Сколько будет 9 умножить на 1? 9

Сколько будет 9 умножить на 3?
ВРЕМЯ ВЫШЛО!
Итоговый счет: 3
>
```

Ой! Похоже, кто-то опоздал.

Относительная сложность сигналов компенсируется их чрезвычайной полезностью. С их помощью вы можете изящно завершать свои программы, а также справляться с задачами, которые выполняются слишком долго.

не бывает Глупых Вопросов

В: Всегда ли сигналы принимаются в порядке их отправления?

О: Нет, если они отправлены практически одновременно. Операционная система может поменять сигналы местами, если посчитает, что один из них важнее остальных.

В: Так происходит всегда?

О: Все зависит от платформы. Например, в большинстве версий Cygwin сигналы всегда будут отправляться и приниматься в одном и том же порядке. Но в целом вы не должны на это рассчитывать.

В: Если я отправлю один и тот же сигнал два раза подряд, получит ли его дважды сам процесс?

О: Опять же все зависит от платформы. Если в Linux и Mac один сигнал повторится слишком быстро, ядро может послать его процессу только один раз. В Cygwin всегда дойдут оба сигнала. Но, как и в предыдущем случае, вы не должны рассчитывать на то, что отправленный дважды сигнал будет оба раза принят процессом.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Операционная система общается с процессами посредством сигналов.
- Как правило, программы останавливаются с помощью сигналов.
- Приняв сигнал, процесс запустит обработчик.
- В ответ на большинство сигналов об ошибках обработчик останавливает программу.
- Обработчик можно заменить функцией `signal()`.
- Вы можете посылать сигналы сами себе, используя `raise()`.
- Таймер посылает сигналы `SIGALRM`.
- Функция `alarm()` задает интервал таймера.
- У каждого процесса есть только один таймер.
- Не используйте одновременно функции `sleep()` и `alarm()`.
- Команда `kill` посылает сигналы процессу.
- `kill -KILL` всегда завершает процесс.

Ваш инструментарий языка Си

Вы изучили главу 10, пополнив свои знания информацией о межпроцессном взаимодействии. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

exit() немедленно останавливает программу.

fileno() находит дескриптор.

dup2() дублирует поток данных.

waitpid() ждет, пока завершится процесс.

pipe() создает канал связи.

Сигналы — это сообщения операционной системы.

sigaction() позволяет обрабатывать сигналы.

Процессы могут взаимодействовать с помощью каналов.

С помощью raise() программа может посылать сигналы сама себе.

Команда kill посылает сигнал.

alarm() посылает сигнал SIGALARM с задержкой в несколько секунд.

11 Сокеты и работа в Сети

Нет места лучше, чем 127.0.0.1

Программам на разных компьютерах тоже нужно общаться.

Вы уже научились использовать ввод/вывод для организации взаимодействия с файлами и узнали, каким образом могут общаться два процесса на одном и том же компьютере. Теперь вы сможете *разговаривать со всем остальным миром*, создавая код на языке Си, способный взаимодействовать с другими программами по **Сети в любой точке *планеты***. К концу этой главы вы научитесь создавать как **серверные**, так и **клиентские приложения**.

Интернет-сервер «Тук-тук»

Язык Си использовался для написания большей части низкоуровневого сетевого кода в Интернете. Как правило, сетевые приложения состоят из двух отдельных программ — **сервера** и **клиента**.

Создадим сервер на Си, который будет рассказывать шутки по Интернету. Вы сможете запустить его на отдельном компьютере следующим образом:

```
File Edit Window Help KnockKnock
> ./ikkp_server
Ожидание подключения
```

Будьте осторожны!

Для тестирования своего кода в этой главе вы будете довольно часто использовать утилиту telnet.

При использовании утилиты telnet, встроенной в Windows, у вас могут возникнуть проблемы, поскольку она по-особому взаимодействует с сетью. Если вы установите версию telnet для Cygwin, то все должно быть в порядке.

Сервер не выведет на экран ничего, кроме сообщения о том, что он работает. Однако, открыв вторую консоль, вы сможете подключиться к нему с помощью программы **telnet**. Эта программа принимает два параметра: *адрес* сервера и *порт*, на котором он работает. Если вы запускаете telnet на том же компьютере, где находится сервер, то в качестве адреса можно использовать **127.0.0.1**.

30000 — это номер сетевого порта.

Если вы запускаете сервер на том же компьютере, используйте 127.0.0.1

Сервер ответил

Эти ответы пишете вы

```
File Edit Window Help Who'sThere?
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
> Кто там?
Смерть
> Ну и что?
Ну и все
Connection closed by foreign host.
>
```

Чтобы связываться с сервером, вам понадобится программа **telnet**. На большинстве систем она установлена изначально. Чтобы проверить ее наличие, наберите

```
telnet
```

в командной строке.

Если у вас *нет* этой утилиты, вы можете установить ее одним из следующих способов:

Cygwin
Запустите файл `setup.exe` для Cygwin и поищите надпись *telnet*.

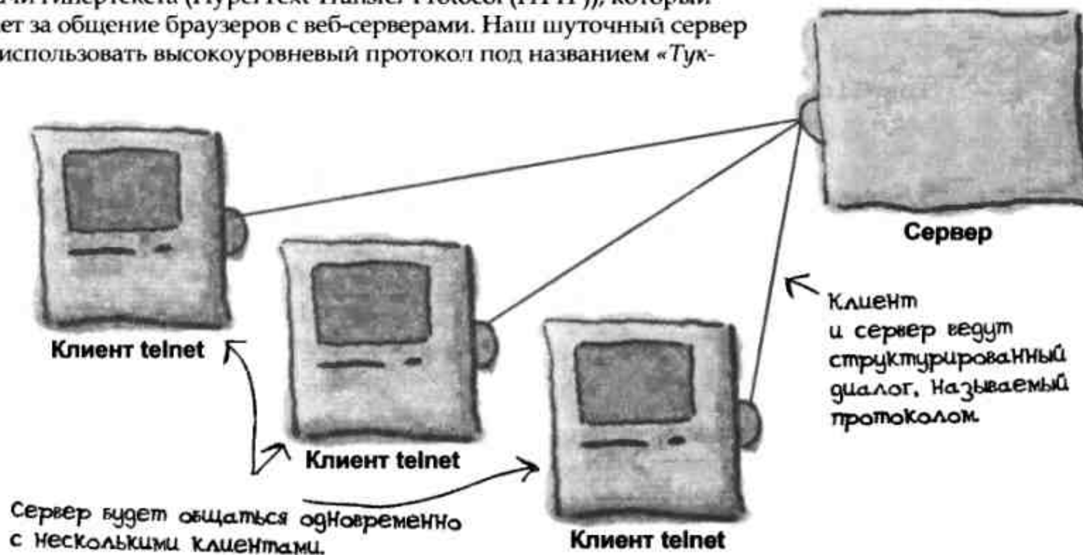
Linux
Поищите *telnet* в своем пакетном менеджере. В большинстве систем пакетный менеджер называется **Synaptic**.

Mac
Если у вас нет программы telnet, вы можете загрузить ее, перейдя по ссылке *www.macports.org* или *www.finkproject.org*.

Обзор сервера «Тук-тук»

Наш сервер сможет общаться одновременно с несколькими клиентами. Общение клиента и сервера является *структурированным* и называется **протоколом**. В Интернете используются разные протоколы. Одни из них *низкоуровневые*, например межсетевой протокол (*Internet Protocol* (IP)), который управляет единичками и нолями, пересылаемыми по Глобальной сети. Другие — *высокоуровневые*, например протокол передачи гипертекста (HyperText Transfer Protocol (HTTP)), который отвечает за общение браузеров с веб-серверами. Наш шуточный сервер будет использовать высокоуровневый протокол под названием *«Тук-тук»*.

Протокол — это структурированный диалог.



Клиент и сервер будут обмениваться сообщениями следующим образом:

Протокол требует, чтобы вы ответили на вопрос «Кто там?» Поэтому я должен немедленно прекратить этот разговор.

У протокола всегда есть строго определенный набор правил. Пока клиент и сервер их соблюдают, все идет хорошо. Но если правила будут нарушены одной из сторон, то диалог, скорее всего, довольно внезапно прервется.

Как сервер разговаривает с Интернетом

Когда программа, написанная на Си, хочет связаться с внешним миром, для чтения и записи байтов она использует **потоки данных**. Вы уже применяли их для соединения с файлами и стандартным вводом/выводом. Но чтобы общаться с Сетью, вашей программе необходим особый вид потока данных — ***сокет***.

```
#include <sys/socket.h>
...
int listener_d = socket(PF_INET, SOCK_STREAM, 0);
if (listener_d == -1)
  error("Не могу открыть сокет");
```

listener_d — это дескриптор для сокета.

Это Интернет-сокет.

Вам понадобится этот заголовок.

Это Номер протокола. Вы можете оставить здесь 0.

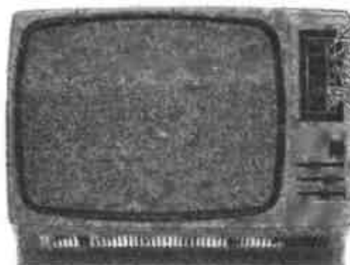
Это функция error(), которую вы создали в предыдущей главе.

Прежде чем сервер сможет общаться с клиентской программой с помощью сокета, он должен пройти несколько стадий: **Привязаться, Прослушать, Принять, Начать.**

Привязаться к порту.
Прослушать порт.
Принять подключение.
Начать разговор.

1. Привязываемся к порту

Компьютер может выполнять несколько серверных приложений одновременно: отправлять веб-страницы, пересылать электронные письма, обслуживать чат и т. д. Чтобы не смешивать сетевые диалоги, все серверы используют разные **порты**. Порт — это как канал на телевидении. Разные порты применяются для разных сетевых сервисов точно так же, как разные каналы — для трансляции разных телепередач.



При запуске сервер должен сообщить операционной системе, какой порт он будет использовать. Это называется ***привязкой к порту***. Сервер «Тук-тук» собирается закрепить за собой порт 30000. Для этого ему потребуются **дескриптор сокета** и его **имя**. Имя — это просто структура, обозначающая «интернет-порт 30000».

```
#include <arpa/inet.h>
...
struct sockaddr_in name;
name.sin_family = PF_INET;
name.sin_port = (in_port_t)htons(30000);
name.sin_addr.s_addr = htonl(INADDR_ANY);
int c = bind (listener_d, (struct sockaddr *) &name, sizeof(name));
if (c == -1)
  error("Не могу связаться с сокетом");
```

Этот заголовок понадобится вам для создания интернет-адресов.

В этих строках создается имя порта, означающее «интернет-порт 30000».

2. Прослушиваем

Если ваш сервер станет популярным, наверняка, к нему будет подключаться множество клиентов одновременно. Если вы хотите, чтобы клиенты ожидали подключения в очереди, вам пригодится системный вызов `listen()`. Он сообщает операционной системе, какой длины должна быть эта очередь, по вашему мнению:

Вы будете использовать очередь длиной в 10 клиентов.

```
if (listen(listener_d, 10) == -1)
  error("Не могу прослушать порт");
```

Вызов `listen()` с параметром 10 означает, что к серверу может подключиться до десяти клиентов одновременно. Не все они мгновенно получат ответ, но смогут подождать его. 11-й клиент будет уведомлен о том, что сервер слишком занят.

У первых 10 клиентов будет возможность подождать.

11-й и 12-й клиенты получат сообщение о том, что сервер слишком занят.

3. Принимаем подключение

Соединившись с портом и настроив очередь для прослушивания, вам нужно просто... подождать. Большую часть времени серверы проводят в ожидании клиентов. Системный вызов `accept()` ждет, пока клиент свяжется с сервером. Как только это случится, он возвратит **дескриптор для второго сокета,** с помощью которого вы сможете вести дальнейший разговор.

client_addr будет хранить данные о клиенте, который только что подключился.

```
struct sockaddr_storage client_addr;
unsigned int address_size = sizeof(client_addr);
int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
if (connect_d == -1)
  error("Не могу открыть второй сокет");
```

Это тот самый **дескриптор соединения** (`connect_d`), с помощью которого сервер...

начнет общение.

Тренажер для ума

Как вы думаете, зачем системный вызов `accept()` создает дескриптор для нового сокета? Почему бы серверу не воспользоваться тем же сокетом, с помощью которого он прослушивает порт?

Сокеты — это не совсем обычные потоки данных

До этого момента все потоки данных ничем не отличались друг от друга. Будь то файл или стандартные ввод/вывод, для общения с ними вы всегда могли использовать функции `fprintf()` и `fscanf()`. Но с сокетами все немного иначе — они могут использоваться в *двух направлениях*: для ввода *и* для вывода. Следовательно, для взаимодействия с сокетами нужны другие функции.

Вы не можете выводить данные в сокет с помощью `fprintf()`. Вместо этого вам нужно использовать функцию **send()**:

Это сообщение, которое вы собираетесь отправить по сети.

```
char *msg = "Сервер с протоколом Тук-тук\r\nВерсия 1.0\r\nТук-тук!\r\n> ";
if (send(connect_d, msg, strlen(msg), 0) == -1)
  error("отправка");
```

Это дескриптор сокета.

Это сообщение и его длина.

Последний параметр используется для тонкой настройки. Можете оставить 0.

Запомните: важно всегда проверять значение, которое возвращают такие системные вызовы, как `send()`. Сетевые ошибки возникают довольно часто, и ваши серверы должны с ними справляться.

Уголок ботана

Какой порт нужно использовать?

Вы должны тщательно подойти к выбору номера порта для серверного приложения. Существует множество разных серверов, и вам необходимо убедиться, что вы не занимаете порт, отведенный для других программ. В Cygwin и на большинстве Unix-подобных компьютеров существует файл ***/etc/services***, в котором перечислены порты большинства распространенных серверов. При выборе порта удостоверьтесь, что его не использует какое-нибудь другое приложение.

Номера портов могут находиться в диапазоне от 0 до 65535, и вам нужно решить, каким будет это число — маленьким (< 1024) или большим. Номера меньше 1024 в большинстве систем доступны только для суперпользователей или администраторов, поскольку они зарезервированы для общеизвестных приложений, таких как веб- и почтовые серверы. Операционные системы предоставляют доступ к этим портам только администраторам, чтобы предостеречь обычных пользователей от запуска нежелательных сервисов.

В большинстве случаев вы, скорее всего, будете выбирать номера портов свыше 1024.

Наточите свой карандаш

Этот сервер пока еще не дописан. Он генерирует случайные советы для любого клиента, который к нему подключается. Вам нужно подставить пропущенные системные вызовы. Кроме того, текущая версия программы завершает соединение сразу после отправки совета. Чтобы этого не происходило, необходимо поместить часть кода в цикл. Но какую часть?

В целях экономии места директивы #include пропущены

```
int main(int argc, char *argv[])
{
  char *advice[] = {
    "Кушайте меньшими порциями \r\n",
     "Прикупите облегающие джинсы. Нет, они Вас НЕ полнят.\r\n",
     "Два слова: не годится\r\n",
     "Будьте честными хотя бы сегодня. Скажите своему начальнику, что Вы *на самом деле*
о нем думаете\r\n",
     "Возможно, Вам стоит подобрать другую прическу\r\n"
  };
  int listener_d = ....................(PF_INET, SOCK_STREAM, 0);

  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  ....................(listener_d, (struct sockaddr *) &name, sizeof(name));

  ....................(listener_d, 10);
  puts("Ожидание подключения");

  struct sockaddr_storage client_addr;
  unsigned int address_size = sizeof(client_addr);
  int connect_d = ................(listener_d, (struct sockaddr *)&client_addr, &address_size);
  char *msg = advice[rand() % 5];

  ....................(connect_d, msg, strlen(msg), 0);
  close(connect_d);

  return 0;
}
```

Вопрос на засыпку. Если вы добавите пропущенные директивы `#include`, программа будет работать, но программист упустил кое-что еще. Что именно? **Подсказка: взгляните на системные вызовы.**

Программист забыл ....

Наточите свой карандаш
Решение

Этот сервер пока еще не дописан. Он генерирует случайные советы для любого клиента, который к нему подключается. Вам нужно было подставить пропущенные системные вызовы. Кроме того, текущая версия программы завершает соединение сразу после отправки совета. Чтобы этого не происходило, необходимо было поместить часть кода в цикл. Но какую часть?

```
int main(int argc, char *argv[])
{
  char *advice[] = {
    "Кушайте меньшими порциями \r\n",
     "Прикупите облегающие джинсы. Нет, они Вас НЕ полнят.\r\n",
     "Два слова: не годится\r\n",
     "Будьте честными хотя бы сегодня. Скажите своему начальнику, что Вы *на самом деле*
о нем думаете\r\n",
     "Возможно, Вам стоит подобрать другую прическу\r\n"
  };
  int listener_d = socket(PF_INET, SOCK_STREAM, 0);   <- Создаем сокет

  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  bind(listener_d, (struct sockaddr *) &name, sizeof(name));   <- Привязываем сокет к порту 30000.

  listen(listener_d, 10);   <- Задаем длину очереди, равную 10.
  puts("Ожидание подключения");
  while (1) {   <- Вам нужно поместить блок с приемом/началом в цикл.
    struct sockaddr_storage client_addr;
    unsigned int address_size = sizeof(client_addr);
    int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
    char *msg = advice[rand() % 5];   <- Принимаем соединение от клиента.

    send(connect_d, msg, strlen(msg), 0);
    close(connect_d);   <- Начинаем диалог с клиентом.
  }
  return 0;
}
```

Вопрос на засыпку. Если вы добавите пропущенные директивы `#include`, программа будет работать, но программист упустил кое-что еще. Что именно? **Подсказка: взгляните на системные вызовы.**

Программист забыл проверить на ошибки. <- Вы всегда должны проверять, не возвращают ли функции socket, bind, listen, accept или send значение -1

Тест-драйв

Давайте скомпилируем наш сервер-советчик и посмотрим, что получилось.

```
File Edit Window Help I'mTheServer
> gcc advice_server.c -o advice_server
> ./advice_server
Ожидание подключения
```

Пока сервер работает, подключитесь к нему несколько раз с помощью telnet, открыв вторую консоль.

```
File Edit Window Help I'mTelnet
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Два слова: не годится
Connection closed by foreign host.
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Возможно, Вам стоит подобрать другую прическу
Connection closed by foreign host.
>
```

Отлично, сервер работает. Здесь в качестве IP-адреса вы использовали 127.0.0.1, поскольку клиент запущен на том же компьютере. Однако вы могли бы подключиться к серверу из любой точки Сети и получить тот же ответ.

Иногда сервер стартует не так, как положено

Если я сначала запущу сервер, а потом клиент — все будет работать...

Серверная консоль

```
File Edit Window Help I'mTheServer
> ./advice_server
Ожидание подключения
```

Сервер запустился.

Клиентская консоль

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Два слова: не годится
Connection closed by foreign host.
>
```

Сервер посылает ответ.

...но если потом я остановлю сервер и практически сразу запущу его вновь, клиент больше не сможет получать ответ!

Серверная консоль

```
File Edit Window Help I'mTheServer
> ./advice_server
Ожидание подключения
^C
> ./advice_server
Ожидание подключения
```

Нажатие Ctrl+C останавливает сервер.

Сервер запущен заново.

Клиентская консоль

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
telnet: Unable to connect to remote host
>
```

Что за...?!?!

Где ответ???

Похоже, что сервер запустился без проблем и во второй раз, но клиент не может получить от него никакого ответа. Почему?

Вспомните, код был написан **без единой проверки на ошибки**. Давайте исправим это упущение и попробуем разобраться, что произошло.

Говорила же мама всегда делать проверку на ошибки

Если в строку, где сокет привязывается к порту, добавить проверку на ошибки:

Вместо этого... ...вот это.

```
bind (listener_d, (struct sockaddr *) &name, sizeof (name));
if (bind (listener_d, (struct sockaddr *) &name, sizeof(name)) == -1)
  error("Не могу привязаться к порту");
```

Здесь вызывается функция error(), которую вы когда-то написали. Она выведет причину ошибки и завершит программу.

то при быстром перезапуске сервера можно будет получить чуть больше информации:

```
> ./advice_server
Ожидание подключения
^C
> ./advice_server
Не могу привязаться к порту
>
```

Привязка заканчивается неудачей!

Если сервер ответит клиенту, а потом перезапустится, то системный вызов функции `bind()` закончится неудачей. Однако остальная часть кода продолжит работу, несмотря на то что использование серверного порта невозможно, поскольку оригинальная версия программы не содержала проверки на ошибки.

ВСЕГДА проверяйте системные вызовы на ошибки.

Привязанные порты отвязываются не сразу

Привязав сокет к определенному порту, операционная система будет предотвращать любые попытки повторной привязки к нему на протяжении следующих 30 секунд (или около того). Это касается и той программы, которая выполняла ее изначально. Чтобы избежать этой проблемы, перед привязкой сокета нужно установить определенный параметр:

```
int reuse = 1;
if (setsockopt(listener_d, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse, sizeof(int)) == -1)
  error("Не могу установить для сокета параметр повторного
  использования");
```

Для хранения параметра вам понадобится переменная типа int. Присвоив ей 1, вы как бы говорите: «Да, порт можно использовать повторно».

Благодаря этому сокет может использовать порт повторно.

Данный код позволяет сокету **повторно использовать** привязанный порт. Таким образом, если вы перезапустите сервер и привяжете порт снова, никаких ошибок не произойдет.

Прием данных от клиента

Вы научились отсылать данные клиенту, но как *прочитать* то, что клиент сам вам отправляет? Аналогично функции `send()` для записи у сокета есть специальная функция **`recv()`** для чтения данных.

```
<прочитанные байты> = recv(<дескриптор>, <буфер>, <количество байт для чтения>, 0);
```

Если набрать в клиентской программе строку текста и нажать клавишу Enter, то функция `recv()` сохранит введенный текст в виде массива символов:

К	т	о		т	а	м	?	\r	\n

При использовании однобитной кодировки, recv() вернет 10, поскольку клиент отправил 10 символов.

Нужно запомнить несколько моментов:

- **Строка не заканчивается символом \0.**
- **При наборе текста в telnet строка всегда заканчивается символами \r\n.**
- **recv() вернет либо количество символов, либо -1 (если произошла ошибка), либо 0 (если клиент закрыл соединение).**
- **Нет никакой гарантии, что вы получите все символы за один вызов recv().**

Последний пункт особенно важен. Из него следует, что иногда вам придется вызвать `recv()` несколько раз:

К	т		о		т	а		м	?	\r	\n

Возможно, вам понадобиться вызвать recv() не один раз, чтобы получить все символы.

Это значит, что вызов `recv()` может оказаться довольно непростым делом. Лучше всего «завернуть» `recv()` в функцию, которая будет записывать в переданный ей массив обычную строку с символом `\0` в конце.

```
int read_in(int socket, char *buf, int len)
{
  char *s = buf;
  int slen = len;
  int c = recv(socket, s, slen, 0);
  while ((c > 0) && (s[c-1] != '\n')) {
    s += c; slen -= c;
    c = recv(socket, s, slen, 0);
  }
  if (c < 0)
    return c;
  else if (c == 0)
    buf[0] = '\0';
  else
    s[c-1]='\0';
  return len - slen;
}
```

Здесь считываются все символы вплоть до '\n'.

Продолжаем считывание до тех пор, пока не закончатся все символы или не будет достигнут символ '\n'.

На случай ошибки.

Нечего считывать. Отправляем обратно пустую строку.

Меняем '\r' на '\0'.

Сходим с проложенной лыжни

Это всего лишь один из способов упростить `recv()`. Может, у *вас* получится лучше? Почему бы вам не написать собственную версию функции `read_in()` и не отправить ее нам на *headfirstlabs.com*?

Код, готовый к выпечке

Вот еще несколько функций, которые могут пригодиться при написании сервера. Понятен ли вам принцип работы каждой из них?

В этой книге вы ЧАСТО использовали функцию error().

Не вызывайте эту функцию, если хотите, чтобы программа продолжала работать.

```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

Выводим ошибку... ...после чего завершаем программу.

Создаем потоковый интернет-сокет.

```
int open_listener_socket()
{
  int s = socket(PF_INET, SOCK_STREAM, 0);
  if (s == -1)
    error("Can't open socket");

  return s;
}
```

Да, используем сокет повторно (чтобы можно было перезапускать сервер без проблем).

```
void bind_to_port(int socket, int port)
{
  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  int reuse = 1;
  if (setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse, sizeof(int)) == -1)
    error("Не могу установить для сокета параметр повторного использования");
  int c = bind (socket, (struct sockaddr *) &name, sizeof(name));
  if (c == -1)
    error("Не могу привязаться к сокету");
}
```

Интернет-порт 30000.

Захватываем порт 30000.

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "Ошибка при общении с клиентом", strerror(errno));
  return result;
}
```

Отправляем строку клиенту.

Если возникнет ошибка, не вызывайте функцию error(). Не нужно останавливать сервер из-за проблем с одним клиентом.

Теперь, когда у вас есть набор серверных функций, давайте опробуем их на деле...

Большое упражнение

Пришло время написать код для **интернет-сервера «Тук-тук»**. На этот раз его будет больше, чем обычно, но вы сможете использовать функции, которые мы подготовили для вас на предыдущей странице. Вот начало программы.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
```

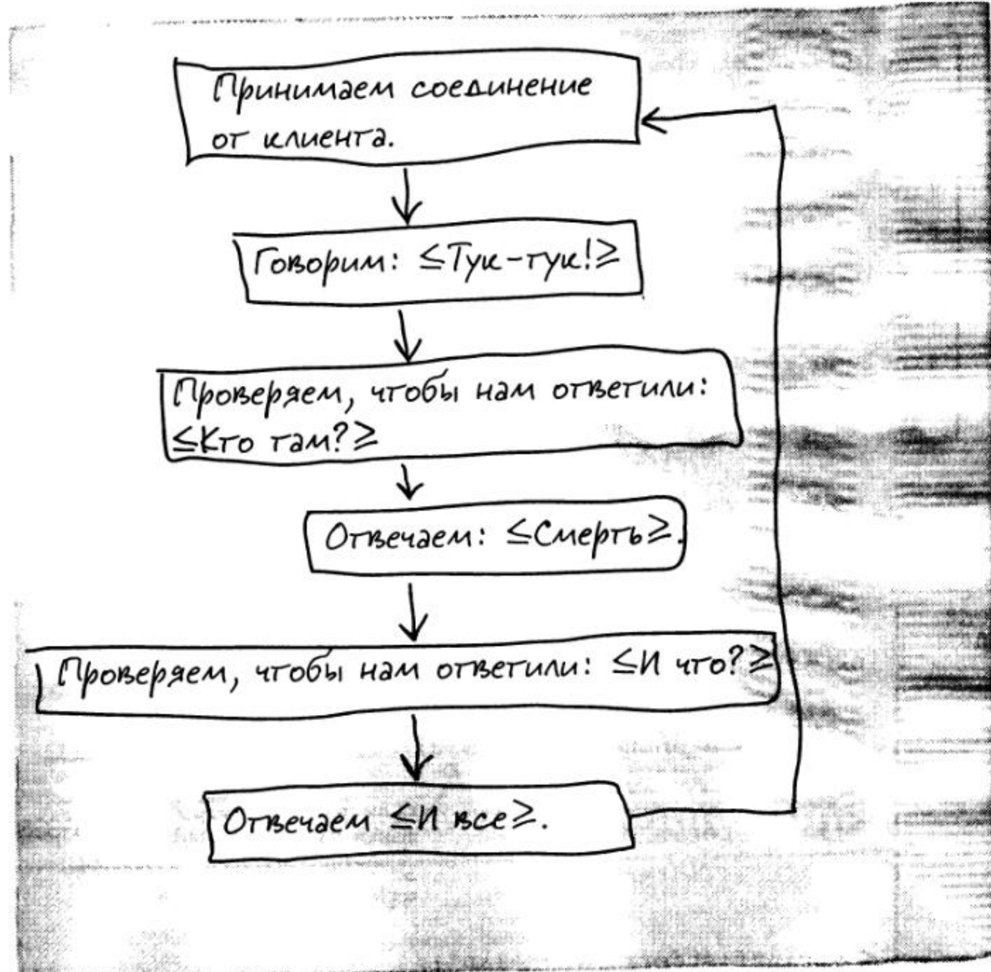
Здесь размещаются функции, заготовленные на предыдущей странице.

Здесь будет храниться главный слушающий сокет сервера.

```
int listener_d;

void handle_shutdown(int sig)
{
  if (listener_d)
    close(listener_d);

  fprintf(stderr, "Пока!\n");
  exit(0);
}
```

Если кто-то нажмет Ctrl+C во время работы сервера, то перед завершением программы эта функция закроет сокет.

Теперь ваша очередь написать главную функцию. Вам необходимо создать новый серверный сокет и сохранить его в переменной `listener_d`. Сокет привязан к порту 30000, длина очереди равна 10. В результате ваш код должен работать следующим образом:



Старайтесь проверять коды ошибок. Если пользователь ответит неправильно, отправьте ему соответствующее сообщение, закройте соединение и ожидайте следующего клиента.

Желаем удачи!

Большое упражнение

Решение

Пришло время написать код для **интернет-сервера «Тук-тук»**. На этот раз его было больше, чем обычно, но вы могли использовать функции, которые мы подготовили для вас на предыдущей странице. Вот начало программы.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
```

Здесь размещаются функции, заготовленные на предыдущей странице.

Здесь будет храниться главный слушающий сокет сервера.

```
int listener_d;

void handle_shutdown(int sig)
{
  if (listener_d)
    close(listener_d);

  fprintf(stderr, "Пока!\n");
  exit(0);
}
```

Если кто-то нажмет Ctrl+C во время работы сервера, то перед завершением программы эта функция закроет сокет.

Вы должны были написать код наподобие этого. Неважно, если он выглядит *не совсем* похожим. Главное, чтобы ваш код мог рассказать шутку и справиться с ошибками

```
int main(int argc, char *argv[])
{
 if (catch_signal(SIGINT, handle_shutdown) == -1)
  error("Can't set the interrupt handler");     <- При нажатии Ctrl+C здесь будет вызвана функция handle_shutdown().
 listener_d = open_listener_socket();
 bind_to_port(listener_d, 30000);     <- Создаем сокет, привязанный к порту 30000.
 if (listen(listener_d, 10) == -1)     <- Устанавливаем размер очереди – 10 клиентов.
  error("Can't listen");
 struct sockaddr_storage client_addr;
 unsigned int address_size = sizeof(client_addr);
 puts("Waiting for connection");
 char buf[255];                                  Ожидаем подключения.
 while (1) {
  int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
  if (connect_d == -1)
   error("Can't open secondary socket");         Отправляем данные клиенту.
  if (say(connect_d,
      "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ") != -1) {
   read_in(connect_d, buf, sizeof(buf));     <- Принимаем данные от клиента.
   if (strncasecmp("Who's there?", buf, 12))
    say(connect_d, "You should say 'Who's there?'!");     <- Проверяем ответ пользователя.
   else {
    if (say(connect_d, "Oscar\r\n> ") != -1) {
     read_in(connect_d, buf, sizeof(buf));
     if (strncasecmp("Oscar who?", buf, 10))
      say(connect_d, "You should say 'Oscar who?'!\r\n");
     else
      say(connect_d, "Oscar silly question, you get a silly answer\r\n");
    }
   }
  }
  close(connect_d);     <- Закрываем второй сокет, который мы использовали для диалога.
 }
 return 0;
}
```

Тест-драйв

Написав сервер «Тук-тук», вы можете его скомпилировать и запустить.

Серверная консоль

```
File Edit Window Help I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Ожидание подключения
```

Сервер ожидает соединения — откройте отдельную консоль и подключитесь к нему с помощью telnet:

Клиентская консоль

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
> Кто там?
Смерть
> Ну и что?
Ну и все
Connection closed by foreign host.
>
```

Сервер может рассказать вам шутку, но что произойдет, если вы нарушите протокол и отправите неверный ответ?

Клиентская консоль

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
>Заходите
Вы должны спросить 'Кто там?'!Connection closed by foreign host.
>
```

Сервер проверяет данные, которые вы ему присылаете, и немедленно закрывает соединение. Закончив работать с сервером, вы можете закрыть его, переключившись обратно на серверную консоль и нажав Ctrl+C. Он даже отправит вам прощальное сообщение:

Серверная консоль

```
File Edit Window Help I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Ожидание подключения
^CПока!
>
```

Отлично! Сервер делает все, что вам нужно.

Или нет?

Сервер может общаться только с одним человеком в отдельный момент времени

У серверного кода есть одна проблема. Представьте, что кто-то подключился к серверу и немного медлит с ответами:

Сервер запущен на компьютере, который находится где-то в Интернете.

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
> Кто там?
Смерть
>
```

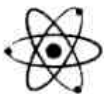
Ой, подожди! Смерть? Стоп, я помню, как нужно ответить... Это так смешно... Смерть, значит, да? Нужно сказать... Подожди, не подсказывай...

Если кто-то еще попытается подключиться к серверу, у него ничего не выйдет — сервер занят первым клиентом:

```
File Edit Window Help I'mAnotherClient
> telnet knockknockster.com 30000
Trying knockknockster.com...
Connected to localhost.
Escape character is '^]'.
```

Просто замечательно! Я не то что не могу подключиться к серверу, я даже не могу выйти из telnet с помощью Ctrl + C. Но почему?

Дело в том, что сервер все еще занят разговором с первым подключившимся клиентом. Второй пользователь будет ожидать ответа от серверного сокета до тех пор, пока сервер снова не выполнит системный вызов `accept()`. Однако пройдет некоторое время, прежде чем это случится, поскольку первый клиент уже подключился.



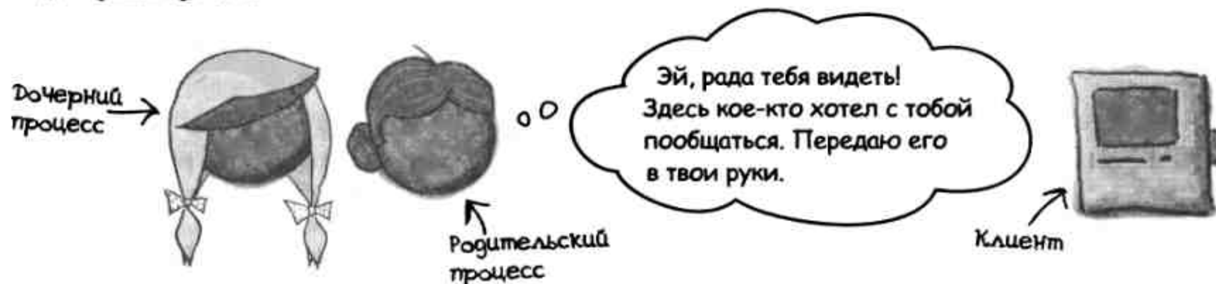
Сила мозга

Сервер не может ответить второму пользователю, потому что он в этот момент обслуживает первого. Какие из приобретенных вами знаний помогут справиться с *двумя* клиентами *одновременно*?

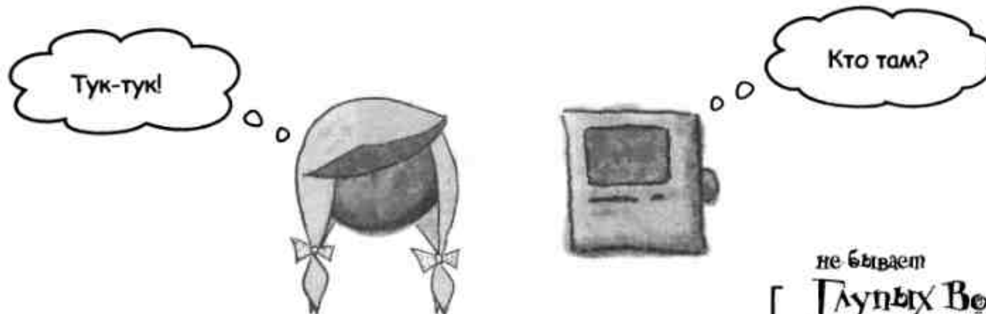
Вы можете клонировать процесс для каждого клиента

Подключаясь к серверу, клиенты начинают диалог на новом отдельном сокете. Это значит, что главный серверный сокет остается свободным и может найти себе нового клиента. Давайте так и сделаем.

Чтобы обслужить диалог между сервером и только что подключившимся клиентом, вы можете создать отдельный дочерний процесс.



Пока клиент общается с дочерним процессом, родительский процесс сервера может принимать подключения от других клиентов.



Родительский и дочерний процессы используют разные сокеты

Следует иметь в виду, что родительский процесс сервера будет работать только с главным слушающим сокетом, поскольку именно он используется для приема новых подключений. При этом дочерние процессы всегда будут иметь дело только с вторичными сокетами, которые создаются при вызове `accept()`. Таким образом, как только произошло клонирование, родительский и дочерний процессы могут закрыть вторичный и главный слушающий сокет соответственно.

После клонирования родительский процесс может закрыть этот сокет

```
close(connect_d);
close(listener_d);
```

Как только дочерний процесс будет создан, он сможет закрыть этот сокет

не бывает **Глупых Вопросов**

В: Если я выделяю новый процесс для каждого клиента, то что произойдет, когда к серверу подключатся сотни клиентов? Мой компьютер создаст сотни процессов?

О: Да. Если вы предполагаете, что у вашего сервера будет много клиентов, вам придется контролировать количество создаваемых процессов. Когда дочерний процесс закончит работу с клиентом, он может сообщить вам об этом — так вы сможете поддерживать определенное число текущих дочерних процессов.

Наточите свой карандаш

Это измененная версия серверного кода, которая создает отдельный дочерний процесс для общения с каждым клиентом... Но она не совсем готова. Попробуйте определить пропущенные фрагменты кода.

```
while (1) {
  int connect_d = accept(listener_d, (struct sockaddr *)&client_addr,
                         &address_size);
  if (connect_d == -1)
    error("Не могу открыть второй сокет");

  if (......................) {
    close(......................);
    if (say(connect_d,
          "Сервер с протоколом Тук-тук\r\nВерсия 1.0\r\Тук-тук!\r\n> ")
        != -1) {
      read_in(connect_d, buf, sizeof(buf));

      if (strncasecmp("Кто там?", buf, 12))
        say(connect_d, "Вы должны спросить 'Кто там?'!");
      else {
        if (say(connect_d, "Смерть\r\n> ") != -1) {
          read_in(connect_d, buf, sizeof(buf));

          if (strncasecmp("И что?", buf, 10))
            say(connect_d, "Вы должны спросить 'И что?'!\r\n");
          else
            say(connect_d, "И все\r\n");
        }
      }
    }
    close(......................);
    ......................  <— Что должен сделать дочерний процесс после
                               закрытия соединения?
  }
  close(......................);
}
```

Наточите свой карандаш
Решение

Это измененная версия серверного кода, которая создает отдельный дочерний процесс для общения с каждым клиентом… Но она не совсем готова. Вам нужно было определить пропущенные фрагменты кода.

```
while (1) {
  int connect_d = accept(listener_d, (struct sockaddr *)&client_addr,
                         &address_size);
  if (connect_d == -1)
    error("Не могу открыть второй сокет");

  if (!fork()) {
    close(listener_d);
    if (say(connect_d,
          "Сервер с протоколом Тук-тук\r\nВерсия 1.0\r\Тук-тук!\r\n> ")
        != -1) {
      read_in(connect_d, buf, sizeof(buf));

      if (strncasecmp("Кто там?", buf, 12))
        say(connect_d, "Вы должны спросить 'Кто там?'!");
      else {
        if (say(connect_d, "Смерть\r\n> ") != -1) {
          read_in(connect_d, buf, sizeof(buf));

          if (strncasecmp("И что?", buf, 10))
            say(connect_d, "Вы должны спросить 'И что?'!\r\n");
          else
            say(connect_d, "И все\r\n");
        }
      }
    }
    close(connect_d);
    exit(0);
  }
  close(connect_d);
}
```

Здесь создается дочерний процесс. Вы уже знаете, что текущий процесс является дочерним, если вызов fork() возвращает 0.

В дочернем процессе нужно закрыть главный слушающий сокет.

Для общения с клиентом дочерний процесс будет использовать только один сокет – connect_d.

Как только диалог закончился, дочерний процесс может закрыть соединение с клиентом.

Закончив общение, дочерний процесс должен завершиться. Это не даст ему попасть в цикл главного сервера.

Тест-драйв

Давайте опробуем модифицированную версию нашего сервера. Вы можете скомпилировать и запустить ее с помощью тех же команд:

Серверная консоль

```
File Edit Window Help I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Ожидание подключения
```

Как и прежде, вы можете подключиться к серверу, открыв отдельную консоль и запустив в ней telnet:

Клиентская консоль

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
> Кто там?
Смерть
>
```

Со стороны все выглядит так же. Чтобы увидеть изменения, оставьте клиента посреди недосказанной шутки:

Открыв третью консоль, вы увидите, что сервер теперь состоит из двух процессов — родительского и дочернего:

В Unix и Cygwin команда ps показывает запущенные процессы

Родительский процесс

```
File Edit Window Help I'mJustCurious
> ps
 PID TTY          TIME CMD
14324 ttys002    0:00.00 ./ikkp_server
14412 ttys002    0:00.00 ./ikkp_server
>
```

Дочерний процесс

Это значит, что можно подключаться, даже если первый клиент все еще общается с сервером.

Еще одна клиентская консоль

```
File Edit Window Help I'mAnotherClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Сервер с протоколом Тук-тук
Версия 1.0
Тук-тук!
>
```

Теперь, когда у вас есть интернет-сервер, давайте посмотрим, насколько сложно будет создать клиента, который сможет считывать данные из Глобальной сети.

Написание веб-клиента

Что если вам захочется написать свою собственную клиентскую программу? Так ли *сильно* она будет отличаться от сервера? Чтобы увидеть сходство и различия, попробуем создать **веб-клиента** для протокола передачи гипертекста (HTTP).

HTTP во многом похож на протокол «Тук-тук», который вы реализовали ранее. Любой протокол является *структурированным диалогом*. Каждый раз, когда клиент и сервер общаются, они «говорят» одно и то же. Откройте telnet и посмотрите, как можно загрузить страницу *http://en.wikipedia.org/wiki/O'Reilly_Media*.

Большинство веб-серверов работает на 80-м порту

Это цифровой адрес «Википедии». У вас он может быть немного другим.

Вам нужно ввести эти две строчки

Затем необходимо нажать Enter и оставить пустую строчку

Сервер начинает свой ответ с дополнительной информации о веб-странице.

```
File Edit Window Help I'mJustCurious
> telnet en.wikipedia.org 80
Trying 91.198.174.225...
Connected to wikipedia-lb.esams.wikimedia.org.
Escape character is '^]'.
GET /wiki/O'Reilly_Media http/1.1
Host: en.wikipedia.org

HTTP/1.0 200 OK
Server: Apache
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang=en" dir="ltr" class"client-nojs"
xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>O'Reilly Media - Wikipedia, the free encyclopedia</title>
...
```

Это путь, который записывается в URL после имени сервера.

В HTTP/1.1 вам нужно указать имя используемого сервера.

Это HTML-код веб-страницы

При подключении к веб-серверу ваша программа должна отправить как минимум три вещи:

На самом деле множество веб-клиентов отправляют значительно больше информации, но вы будете передавать только необходимые данные.

- **Команда GET**
  `GET /wiki/O'Reilly_Media http/1.1`
- **Имя сервера**
  `Host: en.wikipedia.org`
- **Пустая строчка**

Прежде чем отправить на сервер какие-либо данные, нужно установить соединение из клиента. Как это сделать?

Клиент всегда прав

Клиенты и серверы общаются с помощью сокетов, но используют их по-разному. Вы уже знаете, что для ***серверов*** существует последовательность:

1. **Привязаться к порту.**
2. **Прослушать порт.**
3. **Принять подключение.**
4. **Начать разговор.**

Большую часть времени сервер проводит в ожидании подключения со стороны клиента. Пока этого не произойдет, он фактически ничего не может делать. У клиентов такой проблемы нет — они могут подключаться к серверу и начинать общение, когда захотят. Это последовательность для ***клиентов***:

1. **Подключиться к удаленному порту.**
2. **Начать разговор.**

Удаленные порты и IP-адреса

Все, что должен сделать сервер при подключении к сети, — это выбрать порт, который он будет использовать. Однако помимо порта удаленного сервера клиенту необходимо знать еще и **IP-адрес (internet protocol address)**:

```
208.201.239.100
```

Адреса в формате IP четвертой версии (IPv4) состоят из четырех чисел. Когда-нибудь большинство из них вытеснит более длинный формат записи, соответствующий шестой версии.

В связи с тем что интернет-адрес довольно легко забыть, чаще используются **доменные имена**. Доменное имя — это всего лишь какой-то легко запоминающийся текст. Например:

www.oreilly.com

Несмотря на то что люди предпочитают доменные имена, при фактической передаче пакетов с информацией по Сети используются исключительно цифровые IP-адреса.

Создаем сокет для IP-адреса

Как только клиент узнает адрес сервера и номер его порта, он сможет создать **клиентский сокет**. Клиентские и серверные сокеты создаются схожим образом:

```
int s = socket(PF_INET, SOCK_STREAM, 0);
```

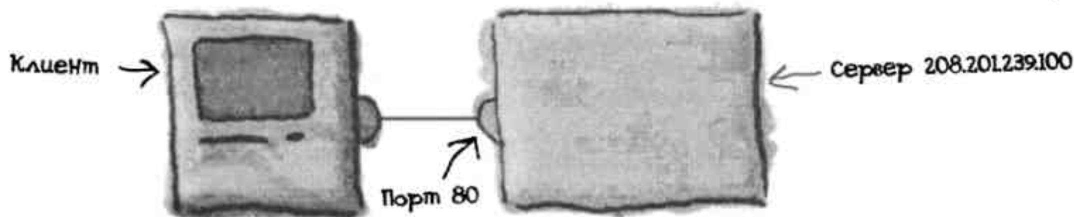
В целях экономии места данный пример не содержит проверку на ошибки. Но в вашем коде такая проверка должна присутствовать всегда.

Разница между клиентским и серверным кодом выражается в работе с сокетами, которые они создают. Сервер **привязывает** сокет к *локальному* порту, тогда как клиент **подключает** его к *удаленному*:

В этих строчках создается сокет с адресом 208.201.239.100 и портом 80.

```
struct sockaddr_in si;
memset(&si, 0, sizeof(si));
si.sin_family = PF_INET;
si.sin_addr.s_addr = inet_addr("208.201.239.100");
si.sin_port = htons(80);
connect(s, (struct sockaddr *) &si, sizeof(si));
```

В этой строчке сокет подключается к удаленному порту.



Эй, я не хочу знать, как подключать сокет к IP-адресу. Я же человек... Я хочу подключаться к настоящему доменному имени.

Код, приведенный выше, работает только для цифровых IP-адресов.

Чтобы подключить сокет с помощью удаленного доменного имени, вам понадобится функция `getaddrinfo()`.

getaddrinfo() получает адреса доменов

Система доменных имен представляет собой огромную адресную книгу. Это способ преобразования доменных имен, таких как *www.oreilly.com*, в цифровые IP-адреса, необходимые компьютеру для отправки пакетов с информацией по Сети.

DNS – это гигантская адресная книга.

Доменное имя	Адрес
en.wikipedia.org	91.198.174.225
www.oreilly.com	208.201.239.100
www.oreilly.com	208.201.239.101

У некоторых больших сайтов есть сразу несколько IP-адресов.

Для создания сетевых пакетов компьютерам нужно знать IP-адреса.

Создаем сокет для доменного имени

В большинстве случаев при создании сокетов в клиентском коде лучше использовать систему DNS — тогда вашим пользователям не придется самим вычислять IP-адреса. Чтобы воспользоваться системой DNS, вам нужно немного изменить способ создания клиентских сокетов:

```
#include <netdb.h>
...
struct addrinfo *res;
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.oreilly.com", "80", &hints, &res);
```

Чтобы использовать функцию getaddrinfo(), вам нужно подключить этот заголовок.

getaddrinfo() ожидает, что порт будет передан в виде строки.

Здесь извлекается информация для порта 80 об именном ресурсе, который находится по адресу www.oreilly.com

`getaddrinfo()` создает в **куче** новую структуру данных под названием *именной ресурс*. Эта структура описывает порт и доменное имя заданного сервера. Внутри нее скрыт IP-адрес, который затем понадобится компьютеру. У некоторых очень крупных доменов может быть несколько IP-адресов, но данный код выберет только один из них. Впоследствии вы можете использовать именные ресурсы для создания сокетов.

Теперь вы можете создать сокет с помощью именного ресурса.

```
int s = socket(res->ai_family, res->ai_socktype,
                         res->ai_protocol);
```

Наконец, вы можете подключиться к удаленному сокету. И поскольку именной ресурс был создан в куче, вам придется его очистить с помощью функции **freeaddrinfo()**:

```
connect(s, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);
```

res->ai_addr – это адрес удаленного узла и порта.

res->ai_addrlen – это размер адреса в памяти.

Здесь происходит подключение к удаленному сокету.

Подключившись, вы можете удалить данные об адресе с помощью функции freeaddrinfo().

Выполнив подключение к удаленному сокету, вы можете использовать его для чтения и записи, применяя те же функции, что и в случае с сервером, — `recv()` и `send()`. Теперь у вас достаточно информации, чтобы написать свой веб-клиент…

Магнитики с кодом

Это код для веб-клиента, который загрузит содержимое страницы из «Википедии» и выведет его на экран. Адрес веб-страницы будет передаваться программе в виде аргумента. Тщательно обдумайте, какие данные вам нужно отправить веб-серверу по HTTP.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int open_socket(char *host, char *port)
{
  struct addrinfo *res;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if (getaddrinfo(host, port, &hints, &res) == -1)
    error("Не могу определить адрес");
  int d_sock = socket(res->ai_family, res->ai_socktype,
                      res->ai_protocol);
  if (d_sock == -1)
    error("Не могу открыть сокет");
  int c = connect(d_sock, res->ai_addr, res->ai_addrlen);
  freeaddrinfo(res);
  if (c == -1)
    error("Не могу подключиться к сокету");
  return d_sock;
}
```

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "Ошибка при общении с сервером",
strerror(errno));
  return result;
}

int main(int argc, char *argv[])
{
  int d_sock;

  d_sock = ..............................................................;
  char buf[255];

  sprintf(buf, ............................................................. , argv[1]);
  say(d_sock, buf);

  say(d_sock, .............................................................);
  char rec[256];
  int bytesRcvd = recv(d_sock, rec, 255, 0);
  while (bytesRcvd) {
    if (bytesRcvd == -1)
      error("Не могу прочитать данные, отправленные сервером");

    rec[bytesRcvd] = .............................................................;
    printf("%s", rec);
    bytesRcvd = recv(d_sock, rec, 255, 0);
  }

  ............................................................. ;
  return 0;
}
```

`'\0'` `"\r\n"` `"Host: en.wikipedia.org\r\n\r\n"` `"GET /wiki/%s http/1.1\r\n"`

`open_socket("en.wikipedia.org", "80")` `"Host: en.wikipedia.org\r\n"` `close(d_sock)`

Магнитики с кодом. Решение

Это код для веб-клиента, который загрузит содержимое страницы из «Википедии» и выведет его на экран. Адрес веб-страницы будет передаваться программе в виде аргумента. Вам нужно было подумать, какие данные стоит отправить веб-серверу по HTTP.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int open_socket(char *host, char *port)
{
  struct addrinfo *res;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if (getaddrinfo(host, port, &hints, &res) == -1)
    error("Не могу определить адрес");
  int d_sock = socket(res->ai_family, res->ai_socktype,
                      res->ai_protocol);
  if (d_sock == -1)
    error("Не могу открыть сокет");
  int c = connect(d_sock, res->ai_addr, res->ai_addrlen);
  freeaddrinfo(res);
  if (c == -1)
    error("Не могу подключиться к сокету");
  return d_sock;
}
```

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "Ошибка при общении с сервером",
strerror(errno));
  return result;
}

int main(int argc, char *argv[])
{
  int d_sock;

  d_sock = open_socket("en.wikipedia.org", "80");
  char buf[255];

  sprintf(buf, "GET /wiki/%s http/1.1\r\n", argv[1]);
  say(d_sock, buf);

  say(d_sock, "Host: en.wikipedia.org\r\n\r\n");
  char rec[256];
  int bytesRcvd = recv(d_sock, rec, 255, 0);
  while (bytesRcvd) {
    if (bytesRcvd == -1)
      error("Не могу прочитать данные, отправленные сервером");

    rec[bytesRcvd] = '\0';
    printf("%s", rec);
    bytesRcvd = recv(d_sock, rec, 255, 0);
  }
  close(d_sock);
  return 0;
}
```

Создаем строку для хранения адреса нужной нам страницы.

Здесь отправляются данные о сервере и пустая строчка.

Чтобы сформировать правильную строку, добавляем в конец массива символ '\0'.

"\r\n"

"Host: en.wikipedia.org\r\n"

Тест-драйв

Если вы скомпилируете и запустите этот веб-клиент, он загрузит страницу из «Википедии»:

Вам нужно заменить все пробелы символами «_».

```
File Edit Window Help I'mTheWebClient
> gcc wiki_client.c -o wiki_client
> ./wiki_client "O'Reilly_Media"
HTTP/1.0 200 OK
Date: Fri, 06 Jan 2012 20:30:15 GMT
Server: Apache
...
Connection: close
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" dir="ltr" class="client-nojs" xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>O'Reilly Media - Wikipedia, the free encyclopedia</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
...
```

Сначала вы получите ЗАГОЛОВКИ ответа. Они содержат информацию о сервере и веб-странице.

Затем появится содержимое веб-страницы из «Википедии».

Работает!

Клиент берет адрес из командной строки, подключается к «Википедии» и загружает страницу. Поскольку здесь происходит формирование *пути* к файлу, вам нужно убедиться, что все пробелы в названии страницы заменены символами «_».

Сходим с проложенной лыжни

Почему бы не сделать так, чтобы в коде замена пробелов выполнялась автоматически? Более подробную информацию о замене символов для веб-адресов вы найдете на странице:

http://www.w3schools.com/tags/ref_urlencode.asp

не бывает
Глупых Вопросов

В: Что нужно использовать при создании сокетов — IP-адреса или доменные имена?

О: В большинстве случаев лучше использовать доменные имена. Их легче запомнить; к тому же некоторые серверы порой меняют свой цифровой адрес, сохраняя при этом доменное имя.

В: А нужно ли мне вообще знать, как подключаться к цифровому адресу?

О: Да. Если сервер не зарегистрирован в системе доменных имен (например, он находится в вашей домашней сети), то вам нужно знать, как подключиться к нему по IP.

В: Можно ли использовать `getaddrinfo()` в сочетании с цифровыми адресами?

О: Да, можно. Но если вы знаете, что это цифровой IP-адрес, то проще использовать первую версию клиентского кода для работы с сокетами.

КЛЮЧЕВЫЕ МОМЕНТЫ

- Протокол — это структурированный диалог.
- Серверы подключаются к локальным портам.
- Клиенты подключаются к удаленным портам.
- И клиент, и серверы используют для общения сокеты.
- Запись в сокет производится с помощью функции `send()`.
- Данные считываются из сокета с помощью функции `recv()`.
- HTTP — это протокол, который используется в Интернете.

Ваш инструментарий языка Си

Вы изучили главу 11, пополнив свои знания информацией о сокетах и работе в Сети. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

telnet — это просто сетевой клиент.

Сокеты создаются с помощью функции socket().

Последовательность действий для сервера:
Привязаться = bind(),
Прослушать = listen(),
Принять = accept(),
Начать диалог.

Для работы с несколькими клиентами одновременно используйте fork().

DNS = система доменных имен

getaddrinfo() Находит адрес по домену.

12 Потоки

Это параллельный мир

Программам часто приходится выполнять несколько заданий одновременно.

С помощью стандартных POSIX-потоков вы можете сделать код более отзывчивым, **разделяя его на несколько параллельно выполняемых частей**. Но будьте осторожны! Потоки являются мощным инструментом, однако вряд ли вы захотите, чтобы они столкнулись друг с другом. В этой главе вы научитесь расставлять **светофоры** и наносить **разметку** так, чтобы ***в вашем коде не образовывались пробки***. В результате вы узнаете, как создавать **POSIX-потоки** и как с помощью **механизмов синхронизации** ***сохранять целостность хрупких данных***.

Задачи выполняются последовательно... или нет...

Представьте, что вы создаете что-нибудь сложное на языке Си, например игру. Ваш код должен выполнить несколько разных задач.

Вашему коду необходимо выполнять эти задачи *одновременно*. Такое положение дел справедливо для многих программ: чат должен одновременно читать и отправлять данные по Сети, медиапроигрыватели — выводить видео на экран и реагировать на управление со стороны пользователя.

Каким образом ваш код может выполнять несколько разных задач в один и тот же момент времени?

...и разбиение на процессы не всегда подходит

Вы уже знаете, что с помощью ***процессов*** можно заставить компьютер выполнять несколько задач одновременно. В предыдущей главе вы создали сетевой сервер, способный справится сразу с несколькими клиентами. Каждый раз, когда подключался новый пользователь, сервер создавал отдельный процесс, чтобы обработать новую сессию.

Означает ли это, что всякий раз для одновременного выполнения нескольких задач вы должны создавать отдельный процесс? Не совсем. И вот почему.

Чтобы создать процесс, нужно время

На некоторых компьютерах создание новых процессов занимает определенное время. Пусть небольшое, но все же. Если на выполнение задачи уходят всего сотые доли секунды, то каждый раз создавать процесс неэффективно.

Процессы не могут обмениваться данными с легкостью

Созданный вами дочерний процесс автоматически получает абсолютную копию всех данных родительского процесса. Но это только копия. Если нужно будет отправить данные от дочернего процесса к родительскому, вам придется использовать что-то вроде канала.

Процессы достаточно сложны

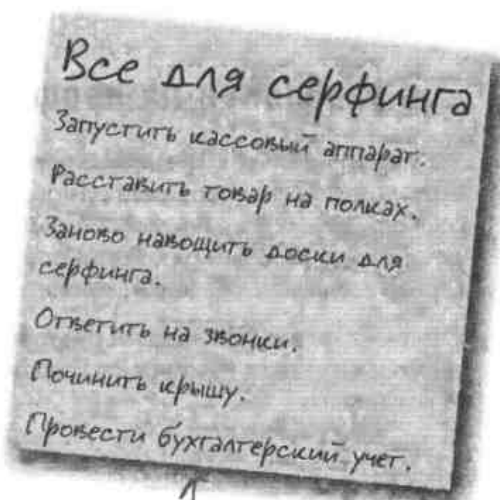
Чтобы сгенерировать процесс, нужно написать довольно много кода. Это может сделать вашу программу длинной и запутанной.

Вам нужно решение, которое сможет быстро запускать задачи, делиться всеми текущими данными и не потребует больших объемов кода.

Вам нужны *потоки*.

Простые процессы выполняют только одну задачу за один раз

Допустим, у вас есть список дел, которые необходимо выполнить.



Как вариант, просто заняться серфингом.

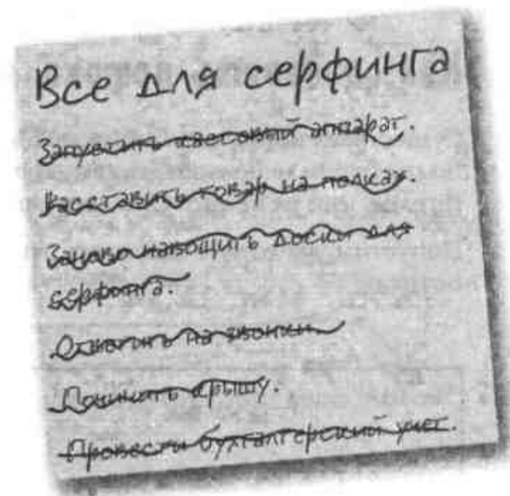
Вы не сможете сами выполнить все задачи одновременно. Если кто-то зайдет в магазин, вам придется прекратить расставлять товар на полках. Если пойдет дождь, вы, вероятно, бросите заниматься бухгалтерским учетом и полезете на крышу. Работая в одиночку, вы ведете себя как простой процесс: выполняете задания последовательно, не более одного в каждый отдельный момент времени. Конечно, вы можете переключаться от задачи к задаче, чтобы везде успевать, но что если произойдет **блокирующая операция**? Что если в то время, когда вы обслуживаете кого-то за кассой, зазвонит телефон?

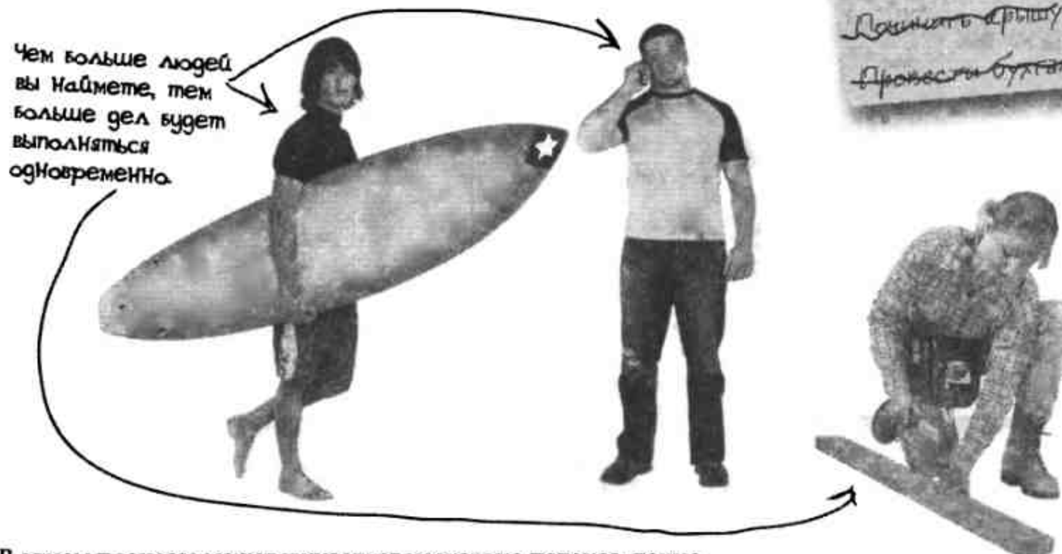
Процесс

Все программы, написанные до этого момента, имели **единственный поток выполнения**. Как будто внутри процесса работал кто-то один.

Наймите дополнительный персонал — используйте потоки

Многопоточная программа похожа на магазин, где работает сразу несколько человек. Один находится у кассы, другой расставляет товар, третий покрывает воском доски для серфинга — и все это они могут делать не отвлекаясь. Если кто-то отвечает по телефону, остальные сотрудники могут продолжать свою работу.

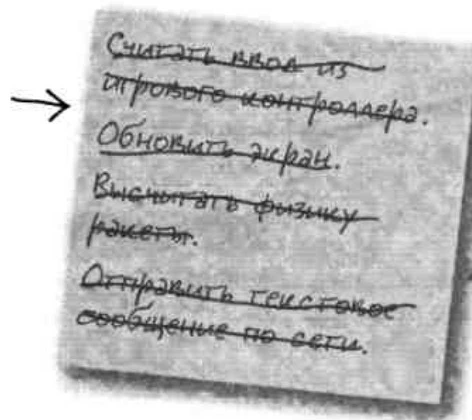




В одном процессе может уживаться несколько потоков, точно так же, как в одном магазине могут работать несколько человек. У всех потоков будет доступ к одному и тому же участку памяти в куче, они смогут работать с теми же файлами и взаимодействовать с теми же сетевыми сокетами. Если один поток изменит глобальную переменную, остальные потоки немедленно увидят эти изменения.

Это значит, что вы можете выделить каждому потоку отдельную задачу — и все они будут выполняться одновременно.

Вы можете выполнять каждую задачу в отдельном потоке.



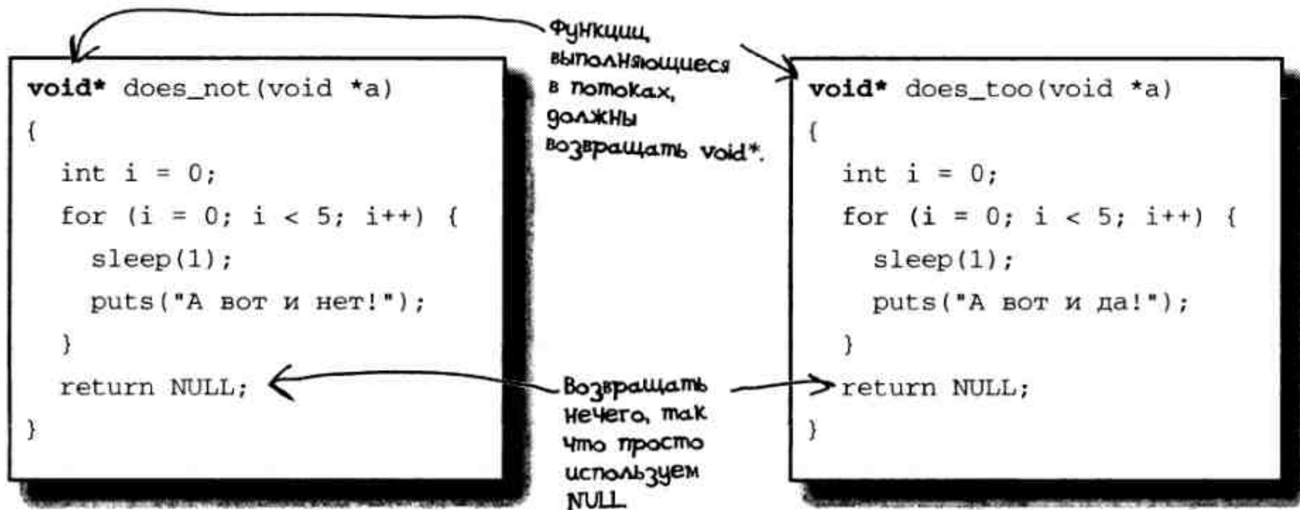
Если один поток находится в ожидании чего-то, другие могут продолжать работать.

Все потоки могут работать внутри одного процесса.

Как создавать потоки

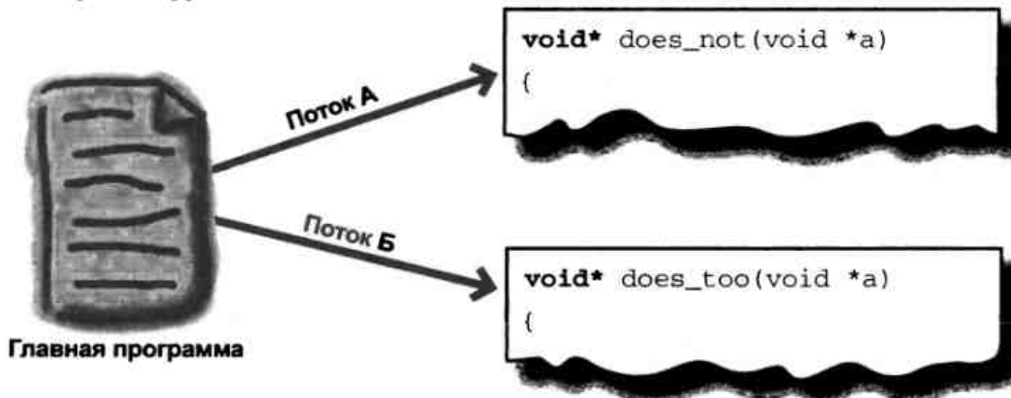
Существует несколько библиотек для работы с потоками, но мы будем использовать самую популярную из них — **POSIX threads**, или **pthread**. Она доступна в Cygwin, Linux и Mac.

Допустим, вы хотите запустить эти две функции в отдельных потоках:



Вы заметили, что обе функции возвращают *пустой указатель*? Напомним, указатель такого типа может ссылаться на любой фрагмент данных, находящихся в памяти. Вы должны следить за тем, чтобы поточные функции, всегда возвращали тип **void***.

Запустим теперь эти функции в отдельных потоках.



Эти функции нужно запустить параллельно в отдельных потоках. Давайте посмотрим, как это делается.

Создаем поток с помощью pthread_create

Для запуска этих функций вам понадобится немного подготовительного кода: несколько заголовков и, возможно, функция `error()` на случай возникновения проблем.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

Это заголовки для основной части кода.

Этот заголовок нужен библиотеке pthread.

Теперь можно начинать работу над кодом для главной функции. С помощью **`pthread_create()`** создайте и запустите два потока, предварительно сохранив информацию о них в структуре данных **`pthread_t`**.

```
pthread_t t0;
pthread_t t1;
if (pthread_create(&t0, NULL, does_not, NULL) == -1)
  error("Не могу создать поток t0");
if (pthread_create(&t1, NULL, does_too, NULL) == -1)
  error("Не могу создать поток t1");
```

Сюда записывается вся информация о потоке.

Здесь создается поток.

does_not – имя функции, которая будет выполняться в потоке.

Всегда делайте проверку на ошибки.

&t1 – это адрес структуры данных, где хранится информация о потоке.

Этот код выполнит две ваши функции в отдельных потоках. Но это еще не все. Когда программа закончит свою работу, потоки будут уничтожены вместе с ней. Поэтому вам нужно дождаться их завершения:

```
void* result;
if (pthread_join(t0, &result) == -1)
  error("Can't join thread t0");
if (pthread_join(t1, &result) == -1)
  error("Can't join thread t1");
```

Здесь будут храниться пустые указатели, возвращаемые каждой из функций.

Функция pthread_join() ждет окончания работы потока.

Значение, которое возвращает запущенная в потоке функция, `pthread_join()` сохраняет в переменную типа void. Как только оба потока закончат свою работу, ваша программа может беспрепятственно завершиться.

Давайте посмотрим, как это работает.

Тест-драйв

Поскольку вы используете библиотеку pthread, вы должны убедиться, что она участвует в компоновке при компиляции программы.

Так вы скомпонуете библиотеку pthread

```
File Edit Window Help Don'tLoseTheThread
> gcc argument.c -lpthread -o argument
```

Это ваша программа.

Запустив код, вы увидите, что обе функции работают одновременно.

```
File Edit Window Help Don'tLoseTheThread
> ./argument
А вот и да!
А вот и нет!
А вот и да!
А вот и нет!
А вот и да!
А вот и нет!
А вот и да!
А вот и нет!
А вот и нет!
А вот и да!
>
```

В вашем случае сообщения могут идти в другом порядке.

не бывает Глупых Вопросов

В: Если обе функции работают одновременно, то почему буквы в сообщениях не перемешались? Каждое сообщение занимает отдельную строчку.

О: Дело в работе стандартного вывода. Текст из функции `puts()` выводится сразу и полностью.

В: Почему после того как я убрал вызов `sleep()`, на экране появился сначала весь текст из одной функции, а потом из другой?

О: Большинство компьютеров выполняют код настолько быстро, что без вызова `sleep()` первая функция закончит свою работу до того, как запустится второй поток.

Пивные магнитики

Пришло время устроить ГРАНДИОЗНУЮ вечеринку. Этот код запускает 20 потоков, которые отсчитывают бутылки пива, начиная с 2 000 000. Попробуйте догадаться, какие фрагменты были пропущены. Если найдете правильный ответ, сможете отпраздновать бутылочкой-другой холодненького.

```
int beers = 2000000;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  return NULL;
}
int main()
{
  pthread_t threads[20];
  int t;
  printf("%i бутылок пива на стене, %i бутылок пива\n", beers, beers);
  for (t = 0; t < 20; t++) {
    ..........(.........., NULL, .........., NULL);
  }
  void* result;
  for (t = 0; t < 20; t++) {
    ..........(threads[t], &result);
  }
  printf("Теперь осталось %i бутылок пива на стене\n", beers);
  return 0;
}
```

- `int beers = 2000000;` — Начинаем с двух миллионов бутылок.
- `drink_lots` — Эта функция будет запущена в каждом из потоков.
- `beers = beers - 1;` — Функция будет уменьшать количество бутылок на 100 000.
- `for (t = 0; t < 20; t++) {` — Для запуска функции вы создадите 20 потоков.
- `..........(.........., NULL, .........., NULL);` — Для экономии места в этом примере мы пропустили проверку на ошибки. Но вы так не делайте!
- `..........(threads[t], &result);` — Здесь код ждет, пока завершатся все дополнительные потоки.

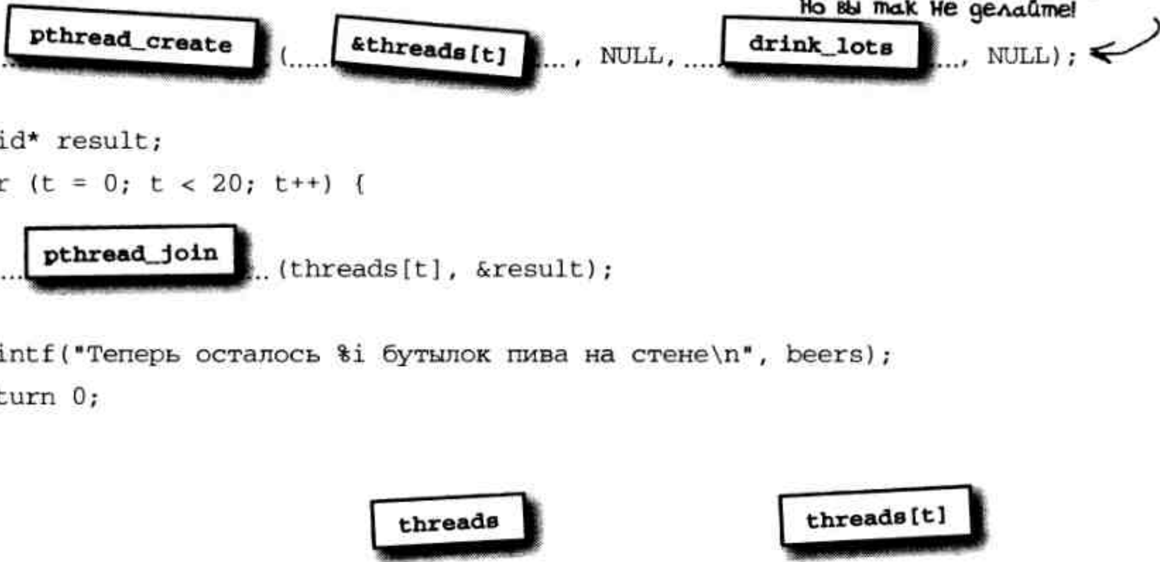
Магнитики: `pthread_join`, `pthread_create`, `threads`, `&threads[t]`, `threads[t]`, `drink_lots`

Пивные магнитики. Решение

Пришло время устроить ГРАНДИОЗНУЮ вечеринку. Этот код запускает 20 потоков, которые отсчитывают бутылки пива, начиная с 2 000 000. Вам нужно было заполнить пропущенные фрагменты.

```
int beers = 2000000;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  return NULL;
}
int main()
{
  pthread_t threads[20];
  int t;
  printf("%i бутылок пива на стене, %i бутылок пива\n", beers, beers);
  for (t = 0; t < 20; t++) {
    pthread_create(&threads[t], NULL, drink_lots, NULL);
  }
  void* result;
  for (t = 0; t < 20; t++) {
    pthread_join(threads[t], &result);
  }
  printf("Теперь осталось %i бутылок пива на стене\n", beers);
  return 0;
}
```

Для экономии места в этом примере мы пропустили проверку на ошибки. Но вы так не делайте!



Тест-драйв

Давайте подробней рассмотрим нашу последнюю программу. Если несколько раз скомпилировать и запустить код, можно увидеть следующее:

```
File Edit Window Help Don'tLoseTheThread
> ./beer
2000000 бутылок пива на стене
2000000 бутылок пива
Теперь осталось 0 бутылок пива на стене
> ./beer
2000000 бутылок пива на стене
2000000 бутылок пива
Теперь осталось 883988 бутылок пива на стене
> ./beer
2000000 бутылок пива на стене
2000000 бутылок пива
Теперь осталось 945170 бутылок пива на стене
>
```

20 потоков сократили значение переменной beers до 0.

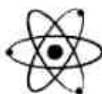
Эй, постойте...

Что за...???

А где пена?

В большинстве случаев код не уменьшает переменную beers до нуля.

Это действительно странно. Переменная `beers` изначально равна двум миллионам. Затем каждый из 20 потоков пытается уменьшить ее значение на 100 000. Разве это не означает, что переменная `beers` *всегда* должна превращаться в ноль?



Сила мозга

Еще раз внимательно посмотрите на код и попробуйте представить, что произойдет, если он будет выполняться одновременно в нескольких потоках. Почему результат непредсказуем? Почему по завершении всех потоков переменная `beers` не равна нулю? Ниже напишите свой ответ.

Код небезопасен с точки зрения многопоточности

Потоки хороши тем, что с их помощью можно одновременно выполнять множество разных задач, имея доступ к одним и тем же данным. Но в этом же заключается и их недостаток…

В отличие от первой программы, во второй происходит чтение и изменение общего участка памяти — переменной `beers`. Давайте разберемся, что происходит, когда два потока попытаются уменьшить значение переменной `beers` с помощью следующей строчки кода:

```
beers = beers - 1;
```

Представьте, что эту строчку одновременно выполняют два потока.

1 Прежде всего обоим потокам нужно прочитать текущее значение переменной beers.

2 Затем каждый поток вычтет из значения единицу.

3 И наконец, каждый поток сохраняет значение beers - 1 обратно в переменную beers.

Несмотря на то что оба потока пытаются уменьшить значение `beers`, у них ничего не получается. Значение уменьшается не на 2, а только на единицу. Вот почему переменная `beers` в конце не равна нулю — потоки постоянно мешают друг другу.

Почему мы получили такой непредсказуемый результат? Потому что потоки не всегда выполняли строчку кода точно в одно и то же время. Иногда они сталкивались, а иногда нет.

Будьте осторожны!

Тщательно выискивайте код, который не соблюдает потоковую безопасность.

Как его определить? Как правило, если два потока читают и изменяют одну и ту же переменную, то такой код небезопасен.

Вам нужно добавить светофоры

Многопоточные программы могут быть не только мощными, но и непредсказуемыми, если вы пустите их на самотек.

Представьте, что два автомобиля приближаются к одному и тому же узкому участку дороги. Чтобы избежать аварии, вы можете поставить светофоры. Благодаря им автомобили не получат одновременный доступ к общему ресурсу (дороге).

Аналогичная ситуация возникает, когда два потока и более получают доступ к общим данным. Необходимо добавить механизм регулировки, чтобы они не могли читать и изменять данные в один и тот же момент времени.

Общая переменная

Два автомобиля символизируют два потока. Они хотят получить доступ к одной и той же общей переменной.

Светофоры не позволят двум потокам получить одновременно доступ к одной и той же общей переменной.

Светофоры, которые предотвращают столкновение потоков друг с другом, называются **мьютексами** (mutex). Это самый простой способ сделать код потокобезопасным.

Мьютексы иногда называют просто блокировками.

MUT-EX = MUTually EXclusive (ВЗаИМНое ИсКЛЮчеНИе).

Используйте мьютексы как светофоры

Чтобы сделать безопасным участок кода, вам нужно создать мьютексную блокировку:

```
pthread_mutex_t a_lock = PTHREAD_MUTEX_INITIALIZER;
```

Мьютекс должен быть доступен всем потокам, которые могут столкнуться друг с другом. Поэтому, скорее всего, вам придется создавать его в виде **глобальной переменной**.

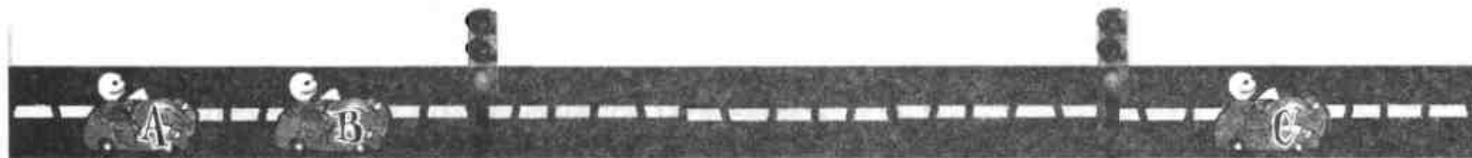
`PTHREAD_MUTEX_INITIALIZER` на самом деле является макросом. Когда компилятор с ним сталкивается, он вставляет на его место весь код, необходимый для правильного создания мьютексной блокировки.

1. **Красный означает «Стой!»**
Вам нужно поместить свой первый светофор в начало опасного участка кода. Функция `pthread_mutex_lock()` даст пройти только **одному потоку**. Остальные должны будут ждать своей очереди.

```
pthread_mutex_lock(&a_lock);
/* Здесь начинается опасный код… */
```

← Здесь сможет пройти только один поток

2. **Зеленый означает «Езжай!»**
В конце опасного участка кода вызывается функция `pthread_mutex_unlock()`. Она снова включает зеленый свет, и опасный код может выполняться другими потоками.




```
/* …Конец опасного кода */
pthread_mutex_unlock(&a_lock);
```

Научившись создавать в своем коде блокировки, вы можете гораздо лучше контролировать работу ваших потоков.

Подробнее о передаче в потоковые функции значений типа long

Функции, работающие в потоках, могут принимать и возвращать одиночные пустые указатели. Вам часто придется передавать в поток целочисленные значения и получать их обратно. Для этого хорошо подходит тип `long`. Значения такого типа можно сохранять внутри пустых указателей, поскольку у них один и тот же размер:

```
void* do_stuff(void* param)
{
  long thread_no = (long)param;
  printf("Поток №%ld\n", thread_no);
  return (void*)(thread_no + 1);
}

int main()
{
  pthread_t threads[20];
  long t;
  for (t = 0; t < 3; t++) {
    pthread_create(&threads[t], NULL, do_stuff, (void*)t);
  }
  void* result;
  for (t = 0; t < 3; t++) {
    pthread_join(threads[t], &result);
    printf("Поток №%ld вернул %ld\n", t, (long)result);
  }
  return 0;
}
```

Потоковая функция может принимать в качестве единственного параметра пустой указатель.

Приводим его обратно к типу long.

При возвращении снова трансформируем его в пустой указатель.

Преобразуем значение long t в пустой указатель.

Прежде чем использовать возвращаемое значение, приводим его к типу long.

File Edit Window Help Don'tLoseTheThread

```
> ./param_test
Поток № 0
Поток № 0 вернул 1
Поток № 1
Поток № 2
Поток № 1 вернул 2
Поток № 2 вернул 3
>
```

Каждый поток получает свой номер.

Каждый поток возвращает свой номер, к которому добавлена единица.

Большое упражнение

Не существует простого способа определить, в какой именно участок кода нужно вставлять блокировки. От расстановки мьютексов зависит, как будет выполняться ваша программа. Ниже представлены две версии функции `drink_lots()`, которые блокируют код по-разному.

Версия А

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  pthread_mutex_lock(&beers_lock);
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  pthread_mutex_unlock(&beers_lock);
  printf("beers = %i\n", beers);
  return NULL;
}
```

Версия Б

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    pthread_mutex_lock(&beers_lock);
    beers = beers - 1;
    pthread_mutex_unlock(&beers_lock);
  }
  printf("beers = %i\n", beers);
  return NULL;
}
```

Обе функции используют мьютексы для защиты переменной `beers`, и каждая из них перед завершением выводит значение этой переменной. Поскольку блокировки кода находятся на разных участках этих функций, они выводят на экран разный текст.

Можете ли вы определить, к какой версии принадлежит каждый из этих программных выводов?

```
File Edit Window Help Don'tLoseTheThread
> ./beer
2000000 бутылок пива на стене
2000000 бутылок пива
beers = 1900000
beers = 1800000
beers = 1700000
beers = 1600000
beers = 1500000
beers = 1400000
beers = 1300000
beers = 1200000
beers = 1100000
beers = 1000000
beers = 900000
beers = 800000
beers = 700000
beers = 600000
beers = 500000
beers = 400000
beers = 300000
beers = 200000
beers = 100000
beers = 0
Теперь осталось 0 бутылок пива на стене
>
```

Сопоставьте код с программным выводом.

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_2
2000000 бутылок пива на стене
2000000 бутылок пива
beers = 63082
beers = 123
beers = 104
beers = 102
beers = 96
beers = 75
beers = 67
beers = 66
beers = 65
beers = 62
beers = 58
beers = 56
beers = 51
beers = 41
beers = 36
beers = 30
beers = 28
beers = 15
beers = 14
beers = 0
Теперь осталось 0 бутылок пива на стене
>
```

Большое упражнение

Решение

Не существует простого способа определить, в какой именно участок кода нужно вставлять блокировки. От расстановки мьютексов зависит, как будет выполняться ваша программа. Ниже представлены две версии функции `drink_lots()`, которые блокируют код по-разному.

Версия А

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  pthread_mutex_lock(&beers_lock);
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  pthread_mutex_unlock(&beers_lock);
  printf("beers = %i\n", beers);
  return NULL;
}
```

Версия Б

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    pthread_mutex_lock(&beers_lock);
    beers = beers - 1;
    pthread_mutex_unlock(&beers_lock);
  }
  printf("beers = %i\n", beers);
  return NULL;
}
```

Обе функции используют мьютексы для защиты переменной `beers`, и каждая из них перед завершением выводит значение этой переменной. Поскольку блокировки кода находятся на разных участках этих функций, они выводят на экран разный текст.

Вам нужно было определить, к какой версии принадлежит каждый из этих программных выводов?

```
File Edit Window Help Don'tLoseTheThread
> ./beer
2000000 бутылок пива на стене
2000000 бутылок пива
beers = 1900000
beers = 1800000
beers = 1700000
beers = 1600000
beers = 1500000
beers = 1400000
beers = 1300000
beers = 1200000
beers = 1100000
beers = 1000000
beers = 900000
beers = 800000
beers = 700000
beers = 600000
beers = 500000
beers = 400000
beers = 300000
beers = 200000
beers = 100000
beers = 0
Теперь осталось 0 бутылок пива на стене
>
```

Сопоставьте код с программным выводом.

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_2
2000000 бутылок пива на стене
2000000 бутылок пива
beers = 63082
beers = 123
beers = 104
beers = 102
beers = 96
beers = 75
beers = 67
beers = 66
beers = 65
beers = 62
beers = 58
beers = 56
beers = 51
beers = 41
beers = 36
beers = 30
beers = 28
beers = 15
beers = 14
beers = 0
Теперь осталось 0 бутылок пива на стене
>
```

Поздравляем! Вы (почти) добрались до конца этой книги. Теперь самое время открыть одну из тех 2 000 000 бутылок и как следует отпраздновать!

Теперь вы вправе выбирать, *какого рода* программистом вы хотите быть. **Linux-хакером**, который пишет на чистом Си? Программировать небольшие **встраиваемые устройства**, такие как Arduino? А может быть, вы хотите стать **разработчиком игр** и перейти на Си++? Или **писать код под Mac и iOS** на Objective-C?

Каким бы ни был ваш выбор, теперь вы являетесь частью сообщества, в котором используют и любят язык Си. Язык, с помощью которого создано больше всего программ. Язык, на котором основан Интернет и практически любая операционная система. Язык, который использовался *для создания почти всех других языков*. И наконец, язык, совместимый с чуть ли не любым существующим устройством — от часов и телефонов до самолетов и спутников.

Приветствуем тебя, о новый хакер!

не бывает Глупых Вопросов

В: Нужно ли компьютеру несколько процессоров, чтобы поддерживать потоки?

О: Нет. У большинство процессоров есть несколько **ядер**, благодаря которым они могут выполнять несколько задач одновременно. Но даже если вы запустите свой код на одноядерном процессоре, то все равно сможете запускать потоки.

В: Как?

О: Операционная система будет быстро переключаться между потоками, чтобы все выглядело так, будто она делает несколько вещей одновременно.

В: Станет ли моя программа быстрее благодаря потокам?

О: Не обязательно. С помощью потоков вы можете использовать больше процессоров и ядер, но нужно тщательно следить за количеством блокировок, выполняемых вашим кодом. При слишком частых блокировках потоки будут работать не быстрее однопоточного кода.

В: Как ускорить работу многопоточного кода?

О: Попробуйте сократить количество данных, к которым нужно получать доступ из потоков. Если их будет не так много, потокам не придется блокировать друг друга слишком часто, что значительно повысит их эффективность.

В: Потоки быстрее отдельных процессов?

О: Как правило, да. Просто потому, что на создание потоков уходит меньше времени, чем на создание процессов.

В: Я слышал, что мьютексы могут стать причиной взаимной блокировки. Так ли это?

О: Допустим, у вас есть два потока и оба они хотят получить мьютексы А и Б. Если в первом потоке сработал мьютекс А, а во втором — Б, то они заблокируют друг друга. Все потому, что первый поток не может получить мьютекс Б, а второй — мьютекс А. Они оба зашли в тупик.

Ваш инструментарий языка Си

Вы изучили главу 12, пополнив свои знания информацией о потоках. Чтобы ознакомиться с полным перечнем подсказок, содержащихся в этой книге, перейдите к приложению II.

С помощью потоков процесс может выполнять несколько задач одновременно.

Простые процессы выполняют только одну задачу за один раз.

Pthread – это библиотека POSIX-потоков.

Потоки – это «легковесные процессы».

pthread_create() создает поток для запуска функции.

Потоки используют одни и те же глобальные переменные.

pthread_join() будет ждать завершения потока.

Если два потока читают и изменяют одну и ту же переменную, код становится непредсказуемым.

Мьютексы создаются с помощью pthread_mutex_lock().

Мьютекс – это защитная блокировка общих данных.

Мьютексы удаляются с помощью pthread_mutex_unlock().

Имя: **Дата:**

Лабораторная работа 3

Бластероиды

В этой лабораторной работе содержится описание программы, которую вы должны изучить и создать, применяя знания, приобретенные при изучении последних нескольких глав.

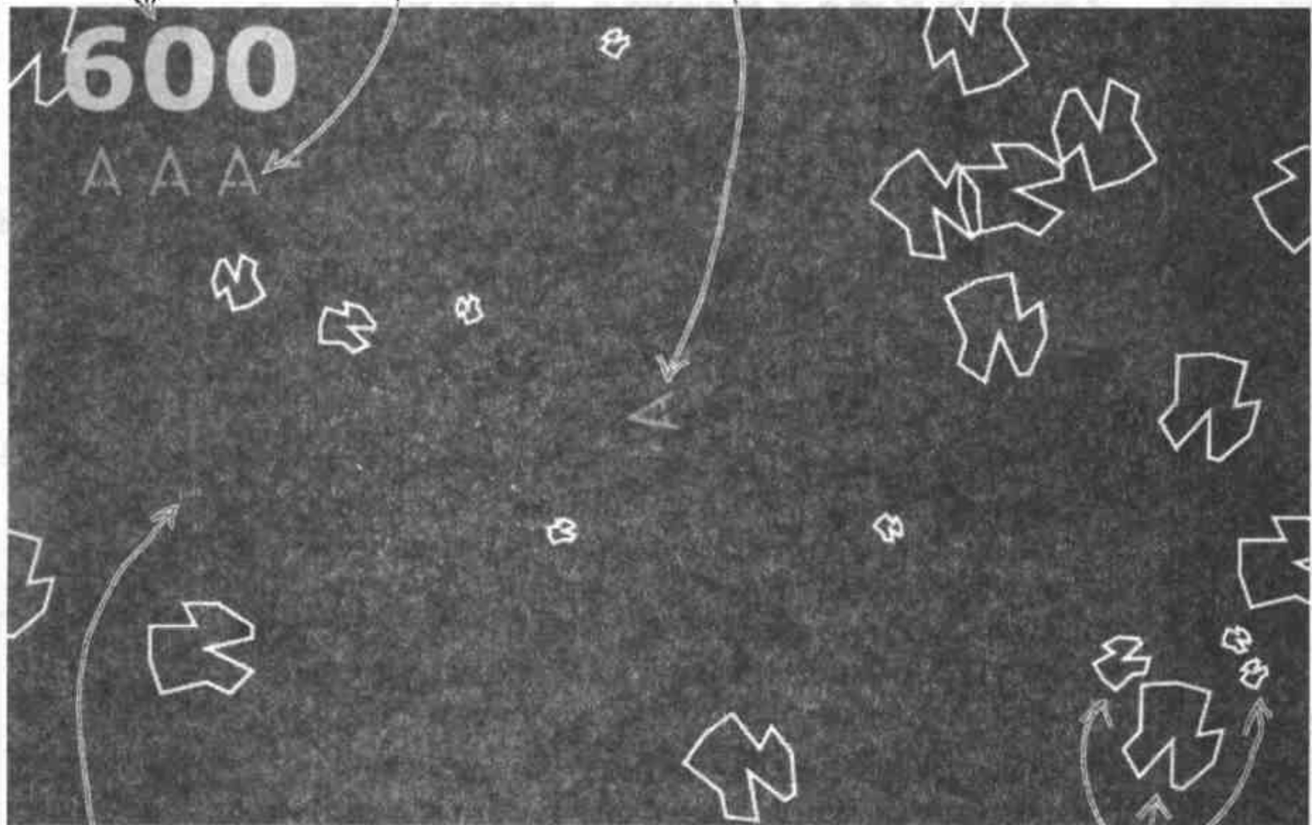
Этот проект больше, чем те, с которыми вы сталкивались до этого момента. Поэтому, прежде чем начинать, не торопитесь и внимательно все прочтите. Не переживайте, если у вас что-то не получится, – здесь нет никаких новых сведений о языке Си. Если необходимо, вы можете продолжить чтение книги и вернуться к этой лабораторной работе позже.

Мы предоставили вам некоторые подробности о структуре программы, сделав так, чтобы у вас было все необходимое для написания кода.

Выполнение этой задачи целиком и полностью зависит от вас, и кода для ее решения у вас не будет.

Пишем аркадную игру «Бластероиды»

Естественно, одна из *истинных* причин изучения языка Си заключается в возможности создавать **игры**. В этой лабораторной работе вы отдадите дань уважения одной из самых популярных и долговечных видеоигр. **Пришло время написать «*Бластероиды*»!**

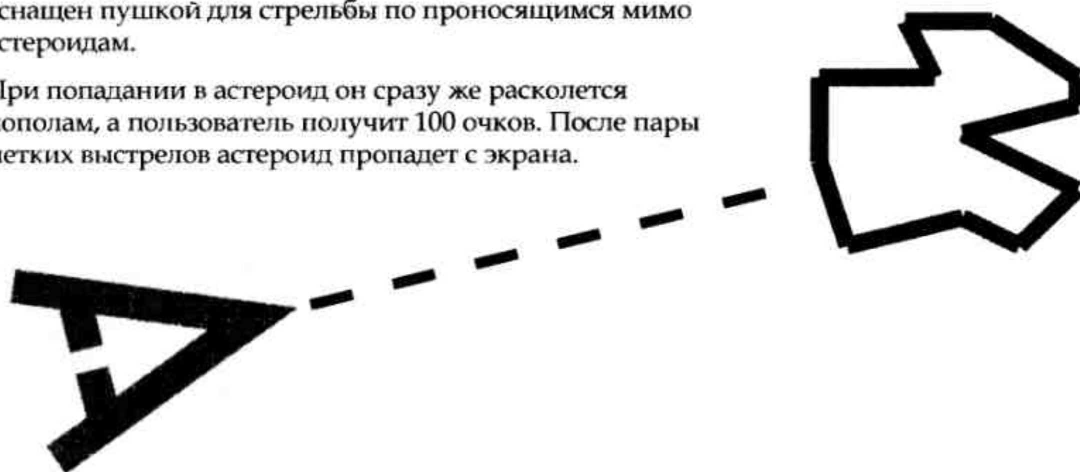


Ваша задача — уничтожить астероиды, избегая столкновения с ними

Зловещие, кровожадные и странным образом похожие друг на друга, астероиды — это то, чего в данной игре стоит опасаться. Они медленно вращаются и проплывают по экрану, грозя мгновенной смертью любому космическому путешественнику, который попадется на их пути.

Добро пожаловать на борт космического корабля «*Вектрорайз*»! Вы будете летать на нем, используя клавиатуру для перемещения по экрану. Корабль оснащен пушкой для стрельбы по проносящимся мимо астероидам.

При попадании в астероид он сразу же расколется пополам, а пользователь получит 100 очков. После пары метких выстрелов астероид пропадет с экрана.



Если корабль столкнется с астероидом, вы потеряете жизнь. У вас всего три жизни. При потере последней игра закончится.

Allegro

Allegro — это библиотека с открытыми исходниками, с помощью которой можно разрабатывать, компилировать и запускать игровой код на разных операционных системах. Она работает в Windows, Linux, Mac OS и даже в телефонах.

Библиотека Allegro довольно проста в использовании, но при этом не обделена возможностями. Она способна работать со звуком, анимацией, различными устройствами и даже 3D-графикой (при условии, что компьютер поддерживает OpenGL).

OpenGL — это открытый стандарт для графических процессоров. Вы описываете свой трехмерный объект, а OpenGL выполняет большинство вычислений за вас.

Установка Allegro

Исходный код библиотеки Allegro можно получить на одноименной странице сайта SourceForge:

http://alleg.sourceforge.net/

Информация в Интернете обновляется чаще, чем в книгах, поэтому данный адрес может измениться. Воспользуйтесь своей любимой поисковой системой для проверки.

Вы можете загрузить, собрать и установить последнюю версию из хранилища. Инструкции, приведенные на сайте, помогут вам сделать это с учетом вашей операционной системы.

Вам может понадобиться CMake

Для сборки кода вам, вероятно, придется установить еще один инструмент — **CMake**. Это утилита, которая упрощает сборку программ на языке Си в разных операционных системах. Вы можете найти ее по адресу *http://www.cmake.org*.

Будьте осторожны!

Код, который мы подготовили для этой лабораторной работы, рассчитан на пятую версию Allegro.

Если вы загрузите и установите более новую версию, скорее всего, вам придется внести несколько изменений.

Какая польза от Allegro?

Библиотека Allegro занимается несколькими вещами.

- **Оконный интерфейс**
  Allegro создаст простое окно с игрой. На первый взгляд, это пустяк. Однако в разных операционных системах окна создаются *совершенно* по-разному. Взаимодействие с клавиатурой и мышью также сильно отличается.

- **События**
  Каждый раз, когда вы нажимаете клавишу, двигаете мышью или щелкаете на чем-то, ваша система генерирует **событие**. События — это просто фрагменты данных, которые сообщают о том, что произошло. Как правило, они выстраиваются в очередь, а затем отправляются в приложения. Allegro помогает реагировать на события. Например, вы с легкостью можете написать код, который будет отвечать за выстрелы из пушки и запускаться по нажатию пробела.

- **Таймеры**
  Вы уже сталкивались с таймерами на системном уровне. Allegro предоставляет простой способ задать игре **ритм**. В том или ином виде ритм присутствует в любой игре, позволяя делать непрерывные и очень частые обновления изображения. К примеру, с помощью таймера вы можете создать игру, которая будет выводить на экран актуальное изображение 60 раз в секунду.

- **Буферизация графики**
  Чтобы игра работала плавно, Allegro использует **двойную буферизацию**. В игровой разработке этот прием позволяет отрисовывать графику в невидимый буфер перед тем, как вывести ее на экран. Поскольку анимационные кадры будут отображаться сразу и целиком, ваша игра сможет работать более плавно.

- **Графика и трансформации**
  Allegro содержит набор встроенных графических **примитивов**, которые позволяют рисовать линии, кривые, текст, фигуры и изображения. Если у вашего графического адаптера есть драйвер с поддержкой OpenGL, то вы сможете работать и в 3D. Кроме того, Allegro поддерживает **трансформации**, с помощью которых можно поворачивать, перемещать и масштабировать графику на экране. Таким образом, вы можете легко создавать анимированные звездолеты и летающие каменные глыбы, способные вращаться и двигаться на экране.

- **Звуки**
  Библиотека для работы со звуком, входящая в состав Allegro, позволит вам добавить в свою игру звуковые эффекты.

Создание игры

Для начала стоит определиться с тем, как вы будете структурировать исходный код. Большинство программистов на Си, вероятно, предпочли бы разбить его на отдельные файлы. Помимо быстрой перекомпиляции это даст вам менее объемные фрагменты кода, что сильно упростит рабочий процесс.

Исходный код можно разделить на части многими способами. Один из вариантов — создать отдельный файл для каждого элемента, который будет отображаться в игре.

asteroid.c

Файл со всем исходным кодом для отслеживания и вывода последнего местоположения астероида.

blast.c

Космический корабль должен стрелять из пушки по пролетающим мимо астероидам, поэтому вам нужен код, чтобы отрисовывать и двигать пушечные заряды на экране.

spaceship.c

Главный герой вашей игры — маленький отважный звездолет. В отличие от многочисленных астероидов, вам, скорее всего, придется управлять только одним космическим кораблем в конкретный момент времени.

blasteroids.c

Для работы с ядром игры всегда хорошо иметь отдельный исходный файл. Этот код будет отслеживать нажатия клавиш на клавиатуре, запускать таймер и сообщать всем другим звездолетам, астероидам и пушечным выстрелам о том, что им нужно нарисовать себя на экране.

Космический корабль

Когда вам приходится управлять на экране множеством объектов, полезно создавать для каждого из них отдельную структуру. Используйте этот код для космического корабля:

```
typedef struct {
  float sx;
  float sy;
  float heading;
  float speed;
  int gone;
  ALLEGRO_COLOR color;
} Spaceship;
```

(sx, sy) — В какой точке экрана он находится.

heading — Куда он направлен.

gone — Он разбился?

Как выглядит космический корабль

Если ваш код примет за точку отсчета **начало координат** (о том, как это сделать, мы поговорим позже), то для отрисовки корабля вы сможете использовать нижеприведенный код.

Переменная s — это указатель на структуру Spaceship. Корабль будет зеленого цвета.

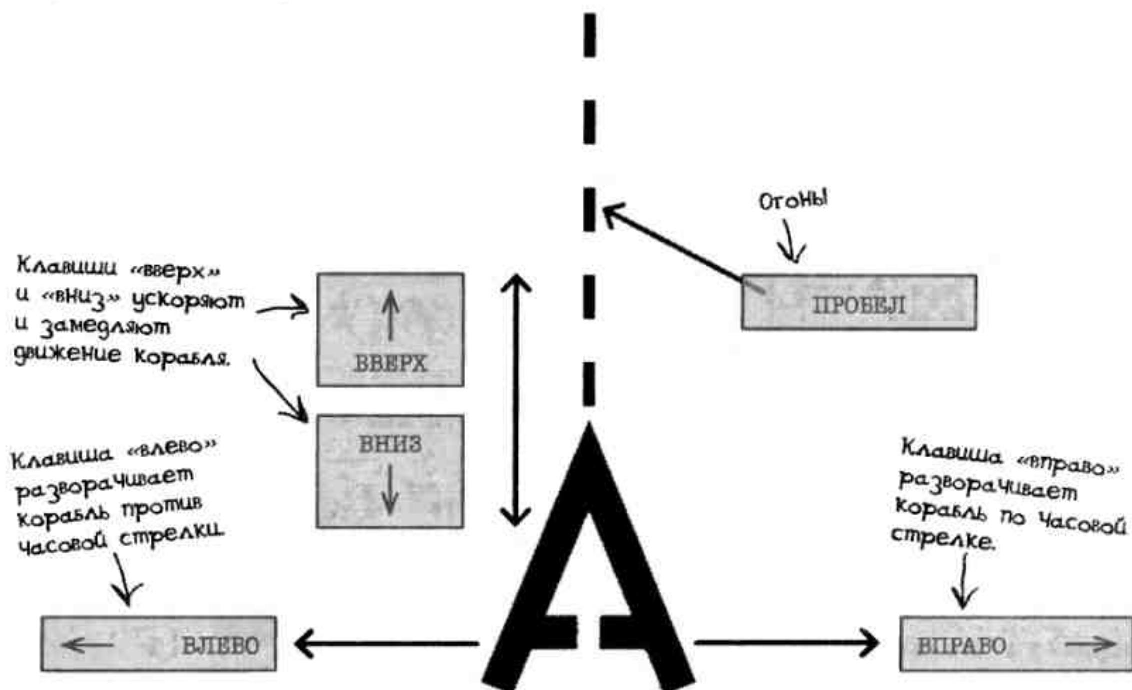
```
al_draw_line(-8, 9, 0, -11, s->color, 3.0f);
al_draw_line(0, -11, 8, 9, s->color, 3.0f);
al_draw_line(-6, 4, -1, 4, s->color, 3.0f);
al_draw_line(6, 4, 1, 4, s->color, 3.0f);
```

Столкновения

Если космический корабль столкнется с астероидом, он потерпит крушение, а игрок потеряет жизнь. В центре экрана появится вновь созданный корабль, и первые пять секунд проверка на столкновения выполняться не будет.

Поведение космического корабля

Космический корабль начинает игру, оставаясь неподвижным в центре экрана. Чтобы он начал летать, вам нужно обеспечить реакцию на нажатие клавиш.



Следите, чтобы космический корабль не слишком сильно ускорялся. Вероятно, вам не захочется, чтобы его скорость превышала несколько сотен пикселов в секунду. Кроме того, он никогда не должен лететь в обратном направлении.

Считывание клавиатурных нажатий

На Си пишется код практически для любого компьютерного оборудования в мире. Но, как ни странно, в этом языке нет общепринятого способа распознать нажатие клавиши. Все стандартные функции, такие как `fgets()`, считывают клавиатурный набор только после того, как будет нажата клавиша «ввод» (Enter). Вот здесь вам и *поможет* Allegro. Все события, которые поступают в игру, написанную с помощью этой библиотеки, проходят через *очередь*. Это обычный список данных, в котором указано, какая клавиша была нажата, где находится указатель мыши и т. д. Чтобы ожидать появления событий в очереди, вам понадобится цикл.

Даже функции вроде getchar() стараются помещать все набранные символы в буфер, пока не будет нажата клавиша «ввод» (Enter).

```
ALLEGRO_EVENT_QUEUE *queue;
queue = al_create_event_queue();   <- Так создается очередь событий
ALLEGRO_EVENT event;
al_wait_for_event(queue, &event);  <- Здесь мы ждем поступления события из буфера.
```

Как только вы получите событие, нужно определить, описывает оно нажатие клавиши или нет. Вы можете сделать это, прочитав его тип.

```
if (event.type == ALLEGRO_EVENT_KEY_DOWN) {
  switch(event.keyboard.keycode) {
  case ALLEGRO_KEY_LEFT:   <- Поворачиваем корабль влево.

    break;
  case ALLEGRO_KEY_RIGHT:
                           <- Поворачиваем вправо.
    break;
  case ALLEGRO_KEY_SPACE:

    break;                 <- Огонь!
  }
}
```

Выстрелы

Получай, булыжник! Ваша задача заключается в том, чтобы обеспечить движение на экране пушечных зарядов космического корабля. Это структура для заряда:

```
typedef struct {
  float sx;
  float sy;
  float heading;
  float speed;
  int gone;
  ALLEGRO_COLOR color;
} Blast;
```

Внешний вид заряда

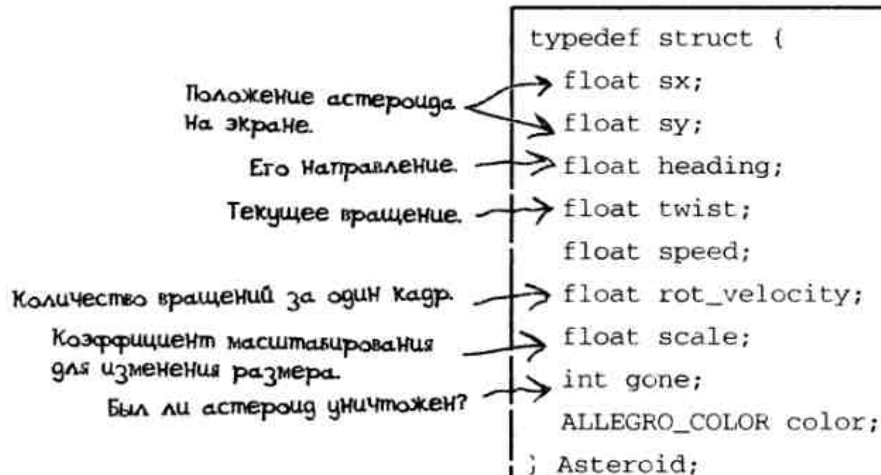
Заряд — это пунктирная линия. Если пользователь будет быстро нажимать клавишу пробела для стрельбы по астероидам, то заряды перекроют друг друга и линия станет больше похожа на сплошную. Таким образом, быстрая стрельба создаст впечатление увеличивающейся огневой мощи.

Поведение зарядов

В отличие от других анимированных объектов, заряды, выходящие за пределы экрана, исчезают навсегда. Это значит, что вам нужно написать код, который сможет легко создавать и уничтожать заряды. Направление выстрелов всегда совпадает с направлением корабля. Заряды всегда следуют по прямой и с постоянной скоростью (пусть она будет в три раза больше, чем скорость корабля). При столкновении заряда с астероидом последний разделится на две части.

Астероид

Используйте эту структуру для каждого астероида:



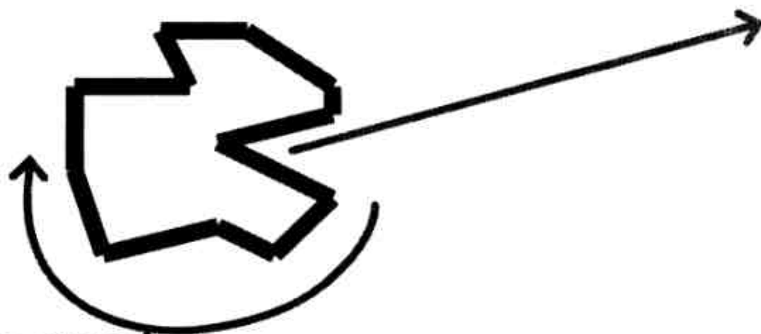
Внешний вид астероида

Это код для отрисовки астероида относительно начала координат:

```
al_draw_line(-20, 20, -25, 5, a->color, 2.0f);
al_draw_line(-25, 5, -25, -10, a->color, 2.0f);
al_draw_line(-25, -10, -5, -10, a->color, 2.0f);
al_draw_line(-5, -10, -10, -20, a->color, 2.0f);
al_draw_line(-10, -20, 5, -20, a->color, 2.0f);
al_draw_line(5, -20, 20, -10, a->color, 2.0f);
al_draw_line(20, -10, 20, -5, a->color, 2.0f);
al_draw_line(20, -5, 0, 0, a->color, 2.0f);
al_draw_line(0, 0, 20, 10, a->color, 2.0f);
al_draw_line(20, 10, 10, 20, a->color, 2.0f);
al_draw_line(10, 20, 0, 15, a->color, 2.0f);
al_draw_line(0, 15, -20, 20, a->color, 2.0f);
```

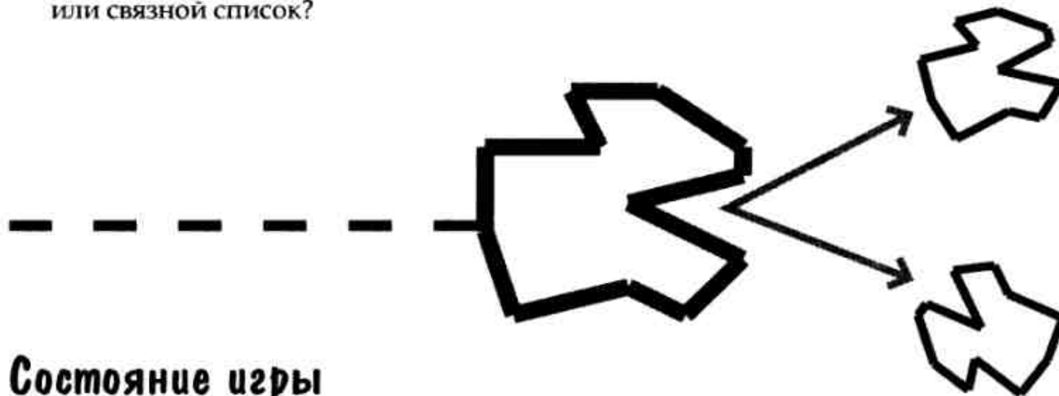
Движения астероида

Астероиды двигаются по прямой, пересекая экран. Кроме того, они постоянно вращаются вокруг своей оси. Вылетев за пределы экрана с одной стороны, астероид немедленно появится с другой.



Попадание заряда в астероид

При попадании пушечного заряда астероид мгновенно распадется на две части, которые будут в два раза меньше его самого. Если астероид будет подбит/разделен пару раз, он исчезнет с экрана. С каждым метким выстрелом очки пользователя увеличиваются на 100. Вам нужно определиться с тем, как записывать набор астероидов, выводящихся на экран. Это будет один большой массив или связной список?



Состояние игры

Помимо всего прочего, вам нужно вывести на экран количество оставшихся жизней и текущий счет. Когда жизни закончатся, в центре экрана должна появиться надпись «Игра окончена», набранная большим приятным шрифтом.

Используйте трансформации для перемещения объектов

Объекты на экране нужно анимировать: космический корабль должен летать, а астероиды вертеться, дрейфовать и даже изменять свой размер. Повороты, перемещения и масштабирования требуют довольно больших математических расчетов, однако в Allegro изначально встроен целый набор *трансформаций*.

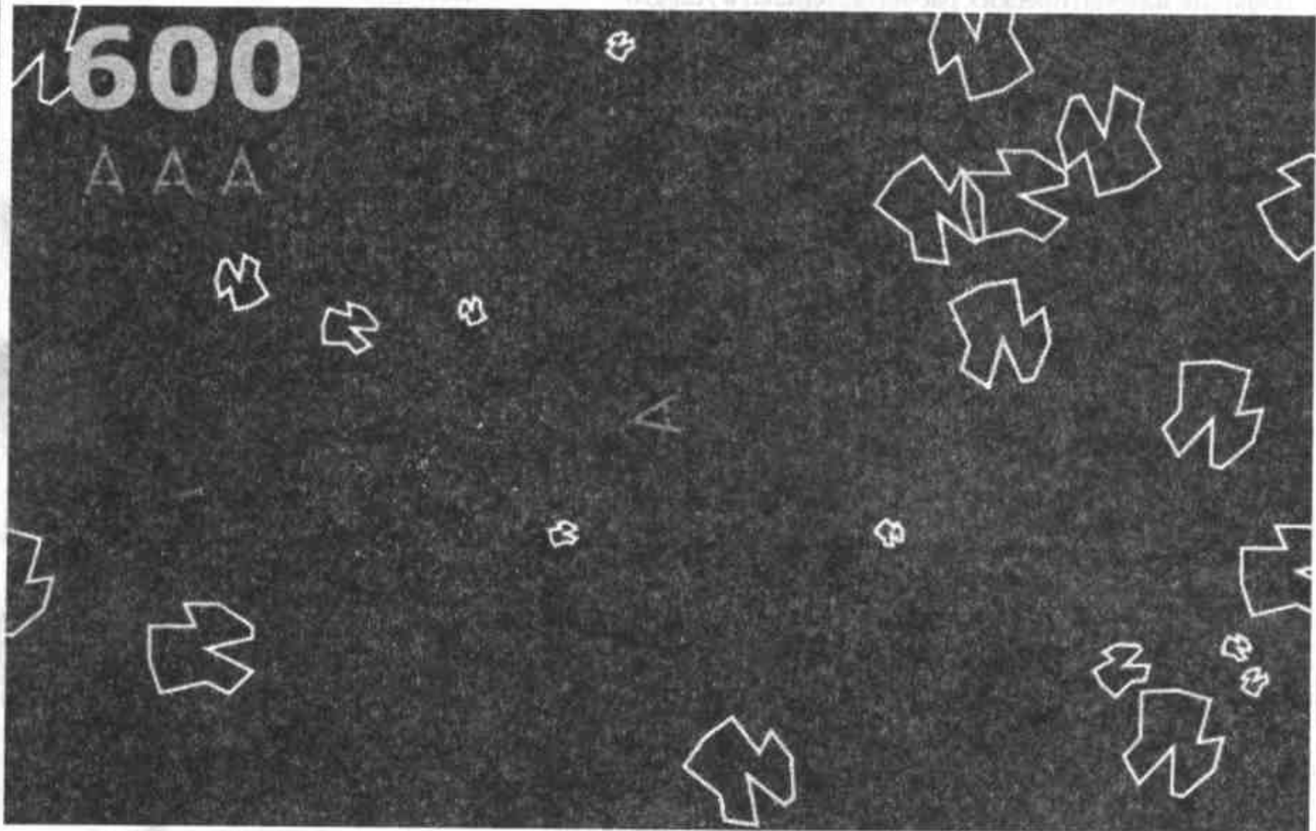
Рисуя объект, например космический корабль, вам стоит побеспокоиться о том, чтобы отобразить его относительно **начала координат**, за которое принимается левый верхний угол экрана с координатами (0; 0). Ось X располагается горизонтально, а ось Y — вертикально. С помощью трансформаций вы можете переместить начало координат в ту точку экрана, где должен находиться объект, а затем повернуть оси так, чтобы они указывали в верном направлении. После того как вы это сделаете, останется только нарисовать объект в начале координат, и все окажется там, где и должно быть.

Это примерный вариант того, как нарисовать на экране звездолет:

```
void draw_ship(Spaceship* s)
{
  ALLEGRO_TRANSFORM transform;
  al_identity_transform(&transform);
  al_rotate_transform(&transform, DEGREES(s->heading));
  al_translate_transform(&transform, s->sx, s->sy);
  al_use_transform(&transform);
  al_draw_line(-8, 9, 0, -11, s->color, 3.0f);
  al_draw_line(0, -11, 8, 9, s->color, 3.0f);
  al_draw_line(-6, 4, -1, 4, s->color, 3.0f);
  al_draw_line(6, 4, 1, 4, s->color, 3.0f);
}
```

Конечный продукт

Закончили? Тогда пришло время сыграть в «*Бластероиды*»!



Вы можете сделать еще много всего, чтобы улучшить эту игру. Почему бы, к примеру, не добавить в нее поддержку OpenCV? Сообщите нам о своих успехах.

Были рады видеть вас в нашем Си-вилле!

Жаль, что приходится прощаться, но нет ничего лучше, чем применить полученные знания на практике. В конце книги вы найдете указатель и еще немного полезной информации. Теперь вам остается взять на вооружение все новые идеи и найти им достойное применение. Мы с нетерпением ждем рассказа о ваших успехах. Не забудьте ***написать нам пару строк*** на сайте ***www.headfirstlabs.com*** и рассказать о том, как **ВАМ** пригодился язык Си.

Приложение I На закуску

Топ-10 фактов (которым мы не уделили внимание)

Даже после всего сказанного еще кое-что осталось.

Есть несколько фактов, о которых, как нам кажется, вы должны знать. С одной стороны, нам не хотелось делать эту книгу объемной настолько, чтобы ее нельзя было поднять без специальной физической подготовки. Но, с другой стороны, мы считаем, что кое-какие моменты заслуживают хотя бы краткого упоминания и игнорировать их было бы неправильно. Поэтому, прежде чем окончательно отложить книгу в сторону, **ознакомьтесь со следующими темами**.

№ 1. Операторы

Помимо основных *арифметических операторов*, используемых в этой книге (+, –, * и /), в языке Си существует еще несколько, которые могли бы упростить вам жизнь.

Инкременты и декременты

Инкремент и *декремент* используются для увеличения и уменьшения числа на единицу соответственно. Это довольно распространенная операция, часто применяемая в циклах при работе со счетчиком. Для инкрементов и декрементов в языке Си есть четыре простых выражения:

Увеличивает i на 1 и возвращает новое значение. → `++i`

Увеличивает i на 1 и возвращает старое значение. → `i++`

Уменьшает i на 1 и возвращает новое значение. → `--i`

Уменьшает i на 1 и возвращает старое значение. → `i--`

Каждое из этих выражений меняет значение `i`. Расположение знаков `++` и `--` указывает на то, какое значение будет возвращаться — новое или старое. Например:

```
int i = 3;
int j = i++;
```

← После этой строчки j==3 и i==4.

Тернарный оператор

Как получать разные значения в зависимости от истинности условия?

```
if (x == 1)
  return 2;
else
  return 3;
```

В Си есть *тернарный оператор*, который позволяет ужать вышеприведенный код до следующей строчки:

```
return (x == 1) ? 2 : 3;
```

Сначала идет условие.

Затем идет значение для случая, когда условие истинно.

В конце указывается значение для случая, когда условие ложно.

Битовые операции

Язык Си можно использовать для низкоуровневого программирования. Он содержит набор операторов, которые позволяют вычислять новые последовательности битов.

Оператор	Описание
`~a`	Все биты в a будут зеркально отражены
`a&b`	Операция И для битов a и b
`a | b`	Операция ИЛИ для битов a и b
`a^b`	Исключающее ИЛИ для битов a и b
`<<`	Сдвиг битов влево (увеличение)
`>>`	Сдвиг битов вправо (уменьшение)

Оператор << можно использовать для быстрого умножения целых чисел на 2. Только следите за тем, чтобы не произошло переполнение.

Запятые для разделения выражений

Вы уже видели циклы `for`, выполняющие код в конце каждого прохода:

```
for (i = 0; i < 10; i++)
```

Данный инкремент будет срабатывать в конце каждой итерации.

Но что если вы хотите выполнять после каждого прохода не одну, а несколько операций? Для этого можно использовать оператор «запятая»:

```
for (i = 0; i < 10; i++, j++)
```

Наращивает i и j

Этот оператор существует для тех случаев, когда вы не хотите разделять выражения точкой с запятой.

№ 2. Директивы препроцессора

Вы используете директивы препроцессора каждый раз, когда компилируете программу, содержащую заголовочные файлы:

```
#include <stdio.h>
```
← Это директива препроцессора

Препроцессор сканирует файл с исходным кодом и генерирует его модифицированную версию, которая и будет компилироваться. В случае с директивой `#include` препроцессор вставляет содержимое файла *stdio.h*. Директивы всегда указываются в начале строки и предваряются символом `#`. Еще одна наиболее часто используемая директива — `#define`:

```
#define DAYS_OF_THE_WEEK 7
...
printf("В неделе %i дней\n", DAYS_OF_THE_WEEK);
```

Директива `#define` создает *макрос*. Препроцессор просканирует исходный код и заменит имя макроса его значением. Макросы отличаются от переменных тем, что не могут меняться во время выполнения программы. Их замена происходит *до* самой компиляции. Вы даже можете создавать макросы, схожие с функциями по поведению:

```
#define ADD_ONE(x) ((x) + 1)
...
printf("Ответ: %i\n", ADD_ONE(3));
```
x – это параметр макроса.

← Будьте осторожны при использовании скобок в макросах.

← Этот код выведет: «Ответ: 4».

Перед тем как программа будет скомпилирована, препроцессор заменит `ADD_ONE(3)` на `((3) + 1)`.

Условия

Вы также можете использовать препроцессор для **условной компиляции**, включая или исключая отдельные части исходного кода:

```
#ifdef SPANISH
char *greeting = "Hola";
#else
char *greeting = "Hello";
#endif
```
← Если макрос SPANISH существует...

← ...подключаем этот код.

← Если нет – подключаем этот код.

Данный код будет компилироваться по-разному в зависимости от того, определен ли макрос `SPANISH`.

№ 3. Ключевое слово static

Допустим, вы хотите создать функцию, которая работает, как счетчик. Вы можете написать ее следующим образом:

```
int count = 0;  <- Считаем количество вызовов.
int counter()
{
  return ++count;  <- Каждый раз наращиваем count.
}
```

Что не так с этим кодом? Он использует глобальную переменную count. Ее значение может изменить любая функция, поскольку count находится в глобальной области видимости. Когда вы начнете писать серьезные программы, стоит следить за тем, чтобы в вашем коде не использовалось слишком много глобальных переменных, — это может привести к ошибкам. К счастью, Си позволяет создавать *глобальные* переменные, доступные только *внутри* функции:

```
int counter()
{
  static int count = 0;
  return ++count;
}
```

Ключевое слово static означает, что эта переменная будет сохранять свое значение между вызовами функции counter().

count остается глобальной переменной, но доступ к ней можно получить только внутри функции.

Ключевое слово static помещает переменную в участок памяти для глобальных значений, и если какая-нибудь другая функция попытается получить доступ к count, то компилятор выдаст ошибку.

static также может делать вещи приватными

Помимо всего прочего, ключевое слово static можно использовать за пределами функций. В таком случае оно будет означать «сюда может получить доступ только код из этого *.c*-файла». Например:

```
static int days = 365;  <- Вы можете использовать эту переменную только внутри текущего исходного файла.

static void update_account(int x) {
...
}
```

Вы можете вызывать данную функцию только из текущего исходного файла.

Ключевое слово static меняет **область видимости**. Оно не позволит получить доступ к вашим данным и функциям из мест, которые не были для этого предусмотрены.

№ 4. Определение размера

Вы уже знаете, что оператор sizeof может сообщить, сколько памяти занимает фрагмент данных. Но что если вам захочется узнать **диапазон значений**, которые он будет хранить? Допустим, вы знаете, что int занимает на вашем компьютере 4 байта. Какое максимальное положительное или отрицательное число вы можете в него записать? Теоретически это можно узнать по количеству байтов, которые занимает тип, но иногда это достаточно сложно сделать.

Вместо этого вы можете использовать макросы, объявленные в заголовке ***limits.h***. Хотите узнать максимальное значение типа long? Для этого есть макрос LONG_MAX. Как насчет самого большого отрицательного числа типа short? Воспользуйтесь SHRT_MIN. В качестве примера приведем программу, которая выводит диапазоны значений для int и short:

```
#include <stdio.h>
#include <limits.h>

int main()
{
  printf("На этом компьютере int занимает %lu байта\n", sizeof(int));
  printf("Тип int может хранить значения от %i до %i\n", INT_MIN, INT_MAX);
  printf("Тип short может хранить значения от %i до %i\n", SHRT_MIN, SHRT_MAX);
  return 0;
}
```

```
File Edit Window Help HowBigIsBig
На этом компьютере int занимает 4 байта
Тип int может хранить значения от -2147483648 до 2147483647
Тип short может хранить значения от -32768 до 32767
```

Имена макросов происходят от названий типов: INT (int), SHRT (short), LONG (long), CHAR (char), FLT (float), DBL (double). Затем нужно добавить либо _MAX (для максимальных положительных значений), либо _MIN (для максимальных отрицательных значений). Если вас интересуют более специфические типы данных, то можно добавить префиксы U (unsigned — без знака), S (signed — со знаком) или L (long — длинный).

№ 5. Автоматизированное тестирование

Тестировать код очень важно. А если этот процесс *автоматизировать*, то жизнь становится куда проще. В наши дни автоматизированное тестирование используют практически все разработчики. Для этих целей существует весьма много фреймворков, но в лаборатории Head First наиболее популярен **AceUnit**:

http://aceunit.sourceforge.net/

AceUnit очень похож на фреймворки для юнит-тестирования из других языков (например, nUnit и jUnit).

Если вы пишете утилиту для командной строки и у вас под рукой Unix-подобная консоль, то обратите внимание на еще один замечательный инструмент под названием **`shunit2`**.

http://code.google.com/p/shunit2/

`shunit2` позволяет создавать shell-скрипты для тестирования других скриптов и команд.

№ 6. Еще немного о gcc

На протяжении всей книги вы использовали компилятор gcc (GNU Compiler Collection), но мы затронули только малую часть того, на что он способен. gcc — это как многофункциональный складной армейский нож. Его возможности практически безграничны, что позволяет получить полный контроль над кодом, который он производит.

Оптимизация

gcc может сделать много всего для улучшения производительности вашего кода. Если он видит, что вы присваиваете одно и то же значение переменной в цикле, он может вынести эту операцию за пределы цикла. Если у вас есть небольшая функция, которая используется всего в нескольких местах, то gcc может *подставить* вместо нее код, который она содержит.

Компилятор gcc поддерживает множество вариантов оптимизации, но большинство из них по умолчанию выключено. Почему? Потому что для их выполнения нужно время, а в процессе написания кода желательно, чтобы компиляция проходила *быстро*. Как только ваш код будет готов, вы, вероятно, захотите включить дополнительную оптимизацию. Существует четыре уровня:

Уровень	Описание
-O	Если вы добавите в команду gcc ключ -O (буква O), то получите первый уровень оптимизации.
-O2	Чтобы повысить оптимизацию и замедлить компиляцию, выбирайте -O2.
-O3	Для еще большей оптимизации выбирайте -O3. При этом будут задействованы предыдущие уровни (-O и -O2) и дополнительная оптимизация.
-Ofast	Максимальная оптимизация выполняется с ключом -Ofast. При этом компиляция происходит наиболее медленно. Будьте осторожны с этим ключом: он снижает вероятность того, что произведенный код будет соответствовать стандартам языка Си.

Предупреждения

Предупреждения выводятся в том случае, когда с формальной точки зрения код грамотен, но вызывает некоторые подозрения, например переменной присваивается значение не того типа. Вы можете увеличить количество проверок, выдающих предупреждения, с помощью **-Wall**:

```
gcc fred.c -Wall -o fred
```

Параметр `-Wall` обозначает «Все предупреждения» (All warnings), но по историческим причинам он *на самом деле* выводит *не все* предупреждения. Чтобы получить полный их список, нужно добавить ключ **-Wextra**:

```
gcc fred.c -Wall -Wextra -o fred
```

И если вы хотите провести *по-настоящему серьезную* компиляцию, то с помощью **-Werror** можно сделать так, чтобы компилятор останавливался при любом предупреждении:

```
gcc fred.c -Werror -o fred
```

Это означает «относись к предупреждениям, как к ошибкам».

Ключ `-Werror` помогает поддерживать качество кода и может пригодиться, если над одним и тем же приложением работают несколько человек одновременно.

Еще больше параметров для `gcc` можно найти по адресу:

http://gcc.gnu.org/onlinedocs/gcc

№ 7. Еще немного о make

make — невероятно мощный инструмент для сборки приложений на Си, который мы только начали рассматривать в этой книге. Более подробно о том, какие поразительные вещи можно делать с помощью make, ищите в книге Роберта Мекленбурга (Robert Mecklenburg) *Managing Projects with GNU Make*:

http://shop.oreilly.com/product/9780596006105.do

Вот лишь некоторые из его возможностей.

Переменные

Использование переменных — отличный способ сократить ваши make-файлы. Например, у вас есть стандартный набор параметров командной строки, который вы хотите передать gcc. Эти параметры можно определить в виде переменной:

```
CFLAGS = -Wall -Wextra -v


fred: fred.c
  gcc fred.c $(CFLAGS) -o fred
```

Вы описываете переменную с помощью знака равенства (=), а затем получаете ее значение, используя запись вида $(...).

Использование %, ^ и @

Довольно часто ваши команды компиляции будут выглядеть подобным образом:

```
fred: fred.c
gcc fred.c -Wall -o fred
```

В этом случае можно использовать символ % для написания наиболее общих целей/рецептов:

```
%: %.c
gcc $^ -Wall -o $@
```

Если вы создаете <файл>, то нужно искать <файл>.c.

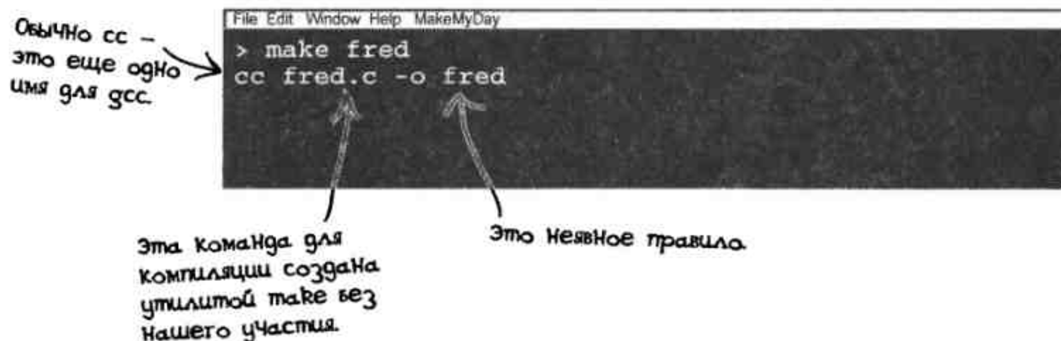
$^ – это значение, хранящее зависимость (.c-файл).

$@ – это имя цели.

Такая запись выглядит странно из-за всех этих символов. Если вы хотите создать файл с именем *fred*, это правило заставит make искать файл с именем *fred.c*. Затем рецепт запустит команду gcc, чтобы создать цель fred (описанную в виде специальных символов), используя заданную зависимость (обозначенную $@).

Неявные правила

Утилита `make` знает о компиляции довольно много и может использовать для сборки файлов *неявные правила*, не посвящая вас в подробности своей работы. Например, если у вас есть файл под названием *fred.c*, вы можете скомпилировать его **без makefile**, набрав:



Все благодаря тому, что `make` поставляется с набором встроенных рецептов. Подробности ищите по адресу:

http://www.gnu.org/software/make/

№ 8. Средства разработки

Создавая код на Си, вы, вероятно, будете весьма обеспокоены его производительностью и стабильностью. И если вы используете компилятор `gcc`, то вас должны заинтересовать некоторые другие инструменты проекта *GNU*.

gdb

Отладчик `gdb` (*GNU Project Debugger*) позволяет исследовать скомпилированную вами программу в процессе ее работы. Он незаменим в тех случаях, когда нужно отловить какую-то досадную ошибку. Этим отладчиком можно воспользоваться как в командной строке, так и в составе *интегрированных сред разработки*, например *Xcode* или *Guile*.

http://sourceware.org/gdb/download/onlinedocs/gdb/index.html

gprof

Если ваш код не так быстр, как вам того хотелось бы, возможно, стоит выполнить *профилирование*. Профайлер `gprof` (*GNU Profiler*) подскажет, какие компоненты вашей программы работают наиболее медленно, чтобы вы могли максимально улучшить свой код. `gprof` позволяет скомпилировать модифицированную версию приложения, которая, завершив работу, выдаст отчет о производительности. Затем с помощью той же утилиты `gprof` вы сможете проанализировать данный отчет и найти самые медленные участки кода.

http://sourceware.org/binutils/docs-2.22/gprof/index.html

gcov

`gcov` (*GNU Coverage*) — еще один инструмент для профилирования. Если профайлер `gprof`, как правило, используется для проверки производительности приложения, то `gcov` — для проверки выполнения участков кода. Это важно для написания автоматизированных тестов, поскольку позволяет понять, выполняются ли проверки для всего кода, который вы хотели бы протестировать.

http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

№ 9. Создание пользовательских интерфейсов

Ни в одной из основных глав этой книги вы не занимались созданием *графических пользовательских интерфейсов* (GUI). В лабораторных работах для написания программ мы использовали библиотеки *Allegro* и *OpenCV*, которые могли выводить на экран простейшие окна. Однако на разных операционных системах процесс проектирования пользовательских интерфейсов имеет кардинальные отличия.

Linux — GTK

В Linux есть библиотеки, которые используются для создания оконных приложений. Одной из самых популярных является GTK+ (*GIMP toolkit*):

http://www.gtk.org/

И хотя библиотека GTK+ используется в основном для создания Linux-приложений, она также доступна в Windows и Mac.

Windows

В Windows есть очень продвинутые встроенные библиотеки для GUI. Программирование под эту операционную систему довольно специфично. Скорее всего, вам придется потратить некоторое время на изучение ее *программных интерфейсов* (WinAPI), прежде чем вы с легкостью сможете создавать оконные приложения. Все больше и больше программ для Windows пишется на языках, основанных на Си, например Си# или Си++. Основы программирования под эту операционную систему можно найти по адресу:

http://www.winprog.org/tutorial/

Mac – Carbon

На компьютерах Macintosh используется графическая система под названием *Aqua*. Вы можете создавать оконные приложения для Mac на Си с помощью набора библиотек **Carbon**. Более современный способ программирования для этой платформы заключается в использовании фреймворка Cocoa, написанного на *Objective-C* — еще одном потомке языка Си. Теперь, когда вы дошли до последних страниц этой книги, у вас есть все необходимое, чтобы приступить к изучению *Objective-C*. В лаборатории Head First просто *обожают* книги и курсы по программированию для Mac, они доступны на сайте *Big Nerd Ranch*:

http://www.bignerdranch.com/

№ 10. Справочные материалы

Вот список некоторых англоязычных книг и сайтов, посвященных программированию на Си.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language* (Prentice Hall; ISBN 978-0-131-10362-7)

Это книга, которая изначально определила язык программирования Си. Почти у каждого разработчика в мире есть ее экземпляр.

Samuel P. Harbison and Guy L. Steele Jr., *C: A Reference Manual* (Prentice Hall; ISBN 978-0-130-89592-9)

Это отличный справочник по Си, который лучше держать под рукой, когда вы пишете код.

Peter van der Linden, *Expert C Programming* (Prentice Hall; ISBN 978-0-131-77429-2)

Эта книга для тех, кого интересует более продвинутое программирование.

Steve Oualline, *Practical C Programming* (O'Reilly; ISBN 978-1-565-92306-5)

Данное издание освещает практические вопросы разработки на Си.

Сайты

Информация по стандартам:
http://pubs.opengroup.org/onlinepubs/9699919799/

Дополнительные учебные пособия по программированию на Си:
http://www.cprogramming.com/

Общая справочная информация:
http://www.cprogrammingreference.com/

Общие учебные пособия по программированию на Си:
http://www.crasseux.com/books/ctutorial/

Приложение II Все темы

Вспомнить все

Всегда хотелось собрать все замечательные факты о Си в одном месте?

Здесь мы собрали все темы и правила, которые были затронуты в этой книге. Взгляните на них и постарайтесь вспомнить все, о чем мы вам рассказали. Для каждого факта указана глава, где он был рассмотрен, в случае чего вы можете запросто вернуться назад. Возможно, вы даже захотите вырвать эти страницы и приклеить их на стену.

Основы

Глава 1

Простые инструкции являются командами.

Глава 1

Блочные инструкции заключены в { и }.

Глава 1

Инструкция if выполняет код, если выражение истинно.

Глава 1

Инструкции switch целесообразно использовать для проверки одной переменной на несколько значений.

Глава 1

Вы можете объединять условия с помощью && и ||.

Глава 1

Любая программа должна иметь функцию main().

Глава 1

#include подключает внешний код, например для ввода или вывода.

Глава 1

Имена ваших исходных файлов должны заканчиваться на .c.

Глава 1

Прежде чем запускать свою программу на Си, вам необходимо ее скомпилировать.

Глава 1

Вы можете использовать оператор && в командной строке, чтобы запускать программу только в том случае, если она проходит компиляцию.

Глава 1

count++ добавляет 1 к count.

Глава 1

Цикл while повторяет код до тех пор, пока условие истинно.

Глава 1

Цикл for — более компактный способ написания циклов.

Глава 1

gcc — самый популярный компилятор для языка Си.

Глава 1

В ключе -o указывается итоговый файл.

Глава 1

count-- вычитает 1 из count.

Глава 1

Циклы do-while выполняют код по меньшей мере один раз.

Указатели и память

Глава 2

С помощью scanf("%i", &x) пользователь может вводить числа напрямую.

Глава 2

Присваивая строку новому массиву, вы копируете ее.

Глава 2

&x возвращает адрес x.

Глава 2

Считывайте содержимое адреса с помощью *a.

Глава 2

Локальные переменные хранятся в стеке.

Глава 2

Переменная-указатель x типа char объявляется как char *x.

Глава 2

&x называется указателем на x.

Глава 2

Переменные массивов можно использовать как указатели.

Глава 2

fgets(buf, size, stdin) – более простой способ ввода текста.

Строки

Глава 2

Строковые литералы хранятся в памяти, предназначенной только для чтения.

Глава 2.5

Заголовок string.h содержит полезные функции для работы со строками.

Глава 2.5

Массив строк — это массив массивов.

Глава 2.5

Массив массивов создается с помощью char strings [...][...].

Глава 2.5

strstr(a, b) вернет адрес строки b внутри строки a.

Глава 2.5

strcmp() сравнивает две строки.

Глава 2.5

strcat() объединяет две строки.

Глава 2.5

strchr() находит местоположение символа внутри строки.

Глава 2.5

strcpy() копирует одну строку в другую.

Глава 2.5

strlen() находит длину строки.

Потоки данных

Глава 3

В Си такие функции, как printf() и scanf(), используют для взаимодействия стандартных ввода и вывода.

Глава 3

По умолчанию стандартный вывод направлен на экран.

Глава 3

По умолчанию стандартный ввод считывает данные с клавиатуры.

Глава 3

С помощью перенаправления вы можете изменить подключение стандартных ввода, вывода и потока ошибок.

Глава 3

Стандартный поток ошибок — это отдельный выходной поток данных, предназначенный для сообщений об ошибках.

Глава 3

Вы можете производить печать в стандартный поток для ошибок при помощи fprintf(stderr, ...).

Глава 3

Вы можете создавать собственные потоки данных с помощью fopen("имя_файла", mode).

Глава 3

Существует три режима:
«w» – для записи,
«r» – для чтения
и «a» – для добавления.

Глава 3

Аргументы командной строки передаются в функцию main() в виде массива строковых указателей.

Глава 3

Функция getopt() упрощает чтение параметров командной строки.

Типы данных

Глава 4

Тип char хранит числа.

Глава 4

Используйте тип long для по-настоящему больших целых чисел.

Глава 4

Используйте тип short для небольших целых чисел.

Глава 4

Используйте тип int для большинства целых чисел.

Глава 2

Тип int имеет разный размер на разных компьютерах.

Глава 4

Используйте тип float для большинства чисел с плавающей точкой.

Глава 4

Используйте тип double для дробных чисел с высокой точностью.

Множественные файлы

Глава 4

Отделяйте объявление функции от ее определения.

Глава 4

Выносите объявления в заголовочный файл.

Глава 4

Директива #include<> нужна для подключения библиотечных заголовков.

Глава 4

Директива #include" " нужна для подключения локальных заголовков.

Глава 4

Сохраняйте объектный код в файлы, чтобы повысить скорость сборки.

Глава 4

Используйте утилиту make для управления процессом сборки.

Структуры

Глава 5

Структура сочетает в себе разные типы данных.

Глава 5

Вы можете считывать поля структуры с помощью оператора «точка».

Глава 5

Вы можете инициализировать структуры так же, как вы это {делали, с, массивами}.

Глава 5

С помощью записи -> вы можете легко обновлять поля, используя указатель на структуру.

Глава 5

typedef позволяет создавать псевдонимы для типов данных.

Глава 5

Назначаемые инициализаторы позволяют присваивать значения полям структур и объединений по их именам.

Объединения и битовые поля

Глава 5

Объединения могут хранить разные типы данных в одном и том же месте.

Глава 5

Перечисления позволяют создавать набор обозначений.

Глава 5

Битовые поля позволяют выбирать, сколько именно битов будет сохранено внутри структуры.

Структуры данных

Глава 6

Динамические структуры данных основаны на рекурсивных структурах.

Глава 6

Рекурсивные структуры содержат одну или несколько ссылок на похожие данные.

Глава 6

Связной список является динамической структурой данных.

Глава 6

В связной список можно легко вставлять данные.

Глава 6

Связной список более гибкий по сравнению с массивом.

Динамическая память

Глава 6

Стек используется для локальных переменных.

Глава 6

В отличие от стека, память в куче не освобождается автоматически.

Глава 6

malloc() выделяет память в куче.

Глава 6

free() освобождает память в куче.

Глава 6

strdup() создает копию строки в куче.

Глава 6

Утечка памяти означает, что вы больше не можете получить доступ к данным, место для которых выделили.

Глава 6

valgrind поможет вам отследить утечки памяти.

Продвинутые функции

Глава 7

Указатели на функции позволяют обмениваться кодом так же, как данными.

Глава 7

Массивы указателей на функции могут помочь запускать разные функции для разных типов данных.

Глава 7

Имя любой функции является указателем на нее.

Глава 7

Только указателям на функции не требуются операторы * и &.

Глава 7

Каждой сортировочной функции нужен указатель на функцию сравнения.

Глава 7

qsort() отсортирует массив..

Глава 7

Функция сравнения определяет порядок, в котором должны располагаться два фрагмента данных.

Глава 7

stdarg.h позволяет создавать функции с переменным количеством параметров.

Статические и динамические библиотеки

Глава 8

#include <> производит поиск в стандартных директориях, таких как /usr/include.

Глава 8

-L<имя> добавляет каталог к списку стандартных директорий.

Глава 8

-l<имя> выполняет компоновку с файлом, который находится в стандартной директории. Например, в /usr/lib.

Глава 8

-I<имя> добавляет каталог к списку стандартных заголовочных директорий.

Глава 8

Команда ar создает библиотечный архив из объектных файлов.

Глава 8

Библиотечные архивы имеют названия вида lib<что-то>.a.

Глава 8

Библиотечные архивы имеют названия вида lib<что-то>.a.

Глава 8

gcc -shared преобразует объектные файлы в динамические библиотеки.

Глава 8

Динамические библиотеки компонуются во время выполнения программы.

Глава 8

В разных операционных системах динамические библиотеки называются по-разному.

Глава 8

Динамические библиотеки имеют расширения .so, .dylib, .dll или .dll.a.

Процессы и взаимодействие

Глава 9

system() запускает строку как консольную команду.

Глава 9

fork() дублирует текущий процесс.

Глава 9

Комбинация fork() и exec() создает дочерний процесс.

Глава 9

execl() = список аргументов.

execle() = список аргументов + переменные среды.

execlp() = список аргументов + поиск по пути.

execv() = массив аргументов.

execve() = массив аргументов + переменные среды.

execvp() = массив аргументов + поиск по пути.

Глава 10

Процессы могут взаимодействовать с помощью каналов.

Глава 10

pipe() создает канал связи.

Глава 10

exit() немедленно останавливает программу.

Глава 10

waitpid() ждет, пока завершится процесс.

Глава 10

fileno() находит дескриптор.

Глава 10

dup2() дублирует поток данных.

Глава 10

Сигналы — это сообщения операционной системы.

Глава 10

sigaction() позволяет обрабатывать сигналы.

Глава 10

С помощью raise() программа может посылать сигналы сама себе.

Глава 10

alarm() посылает сигнал SIGALARM с задержкой в несколько секунд.

Глава 10

Команда kill посылает сигнал.

Глава 12

Простые процессы выполняют только одну задачу за один раз.

Сокеты и работа в Сети

Глава 11

telnet — это просто сетевой клиент.

Глава 11

Сокеты создаются с помощью функции socket().

Глава 11

Последовательность действий для сервера:
Привязаться = bind(),
Прослушать = listen(),
Принять = accept(),
Начать диалог.

Глава 11

Для работы с несколькими клиентами одновременно используйте fork().

Глава 11

DNS = система доменных имен

Глава 11

getaddrinfo() находит адрес по домену.

Потоки

Глава 12

С помощью потоков процесс может выполнять несколько задач одновременно.

Глава 12

Потоки — это «легковесные процессы».

Глава 12

Pthread — это библиотека POSIX-потоков.

Глава 12

pthread_create() создает поток для запуска функции.

Глава 12

pthread_join() будет ждать завершения потока.

Глава 12

Потоки используют одни и те же глобальные переменные.

Глава 12

Если два потока читают и изменяют одну и ту же переменную, код становится непредсказуемым.

Глава 12

Мьютекс — это защитная блокировка общих данных.

Глава 12

Мьютексы создаются с помощью pthread_mutex_lock().

Глава 12

Мьютексы удаляются с помощью pthread_mutex_unlock().

Алфавитный указатель

Символы

$ в командах компиляции для makefile 584

\0 *см.* Нуль-символ

& (амперсанд):

&&, оператор логического И 54, 56

оператор И 56, 577

ссылочный оператор 79, 84

( ) скобки при использовании структур 276

< > (угловые скобки):

<<, >>, оператор побитового сдвига 577

< для перенаправления стандартного ввода 147

> — — стандартного вывода 148, 466

2> — — стандартного потока ошибок 468

в заголовочных файлах 216, 390

{ } (фигурные скобки):

заключение тела функции 42

— выражений 50

[ ] (квадратные скобки):

в массивах 97

в объявлениях переменных 74

создание массивов и доступ к элементам 96

* (звездочка):

доступ к элементам массива 97

объявление переменных 74

оператор разыменования 84

^, оператор исключающего ИЛИ 577

, (запятая):

разделение выражений 577

значений в перечислениях 291

; (точка с запятой), разделитель 255

. (оператор «точка»):

присваивание значения объединениям 284

чтение полей структуры 258

… (многоточие) 381

!, оператор отрицания НЕ 54

?, тернарный оператор 576

(решетка) в директивах препроцессора 578

= (знак равенства):

оператор присваивания 49

— равенства 49

- (минус):
-- (декремент) 49, 576
-= (уменьшение переменной) 49
в командной строке 191
+ (плюс):
++ (инкремент) 49, 576
+= (увеличение переменной) 49
%, символ-спецификатор 42, 84, 88
|, оператор ИЛИ 56, 577
соединение ввода и вывода 167
||, оператор логического ИЛИ 54, 56
" " (двойные кавычки):
в строках 49
в заголовочных файлах 216, 390
' ' (одинарные кавычки) в строках 49
/ (слеш) в комментариях 44
~ (тильда), оператор отрицания 577

A

accept(), функция 507

AceUnit *см.* Автоматизированное тестирование

alarm(), функция 494–495

Allegro, библиотека 562–563

ANSI C, стандарт 38, 55, 311

ar, команда 395, 399

Arduino 245

  функции 250

Arduino IDE 245

autoconf 238

B

break, оператор 62, 67

bus error 49

C

C11, стандарт 38, 44, 200, 311

C89, стандарт 284

C99, стандарт 38, 44, 55, 68, 200, 284, 301, 311

Carbon, библиотека 587

case, оператор 62

char, тип данных 198, 322

  арифметические действия 218

char** 356, 369

checksum(), функция 388

CMake 562

const, инструкция 112, 115

continue, инструкция 67

COW 462

CreateProcess(), функция 462

Cygwin, компилятор 32, 115, 145, 154, 395, 411, 413, 422, 443, 456, 462, 501, 504, 508

D

define, директива 578

do…while, цикл 65

double, тип данных 198

dup2(), функция 469

E

enum, тип данных 291

errno, переменная 444

exec(), функция 440, 447, 456, 462

  с поддержкой массивов 442

  — — списков 441

exit(), функция 470, 477

extern, ключевое слово 222

F

fclose 174

fgets(), функция 103–104

FILE 174–175

fileno(), функция 469

float, тип данных 198
  приведение к целому числу 200
for, цикл 66
fopen(), функция 174, 183
fork(), функция 456, 459, 462
fprintf(), функция 158
free(), функция 315–316
freeaddrinfo(), функция 529
fscanf(), функция 158, 174

G

gcc 32, 45, 58, 75, 582–583
getaddrinfo(), функция 529, 535
getopt(), функция 185
GNU Compiler Collection *см.* gcc

H

head, фильтр 145
HTTP, протокол 505

I

if, инструкция 50
IEEE 204
include, директива 210, 390
int, тип данных 198
IP, протокол 505
IP-адрес 527

K

kill, команда 493

L

listen(), команда 507
long:
  ключевое слово 200
  тип данных 198, 551

M

main() 41–42, 177
malloc(), функция 314, 316
make, утилита 234, 238, 584–585
  принцип работы 235–236
  разновидности 235
  руководство 240
makefile 236, 238
MinGW, компилятор 32, 235, 391, 411, 413, 422, 441
mkfifo(), функция 486

N

nm, команда 394, 396

O

OpenCV, библиотека 427

P

PATH, переменная 46, 413, 422, 441–443

PID 440

pid_status, параметр 475, 477

pipe(), функция 480

POSIX 185, 542

printf() 42, 146, 158

pthread_create(), функция 543

Q

qsort(), функция 362–363, 369

R

raise(), команда 493

recv(), функция 514

return, инструкция 68–69

RSS-поток 452

RSS Gossip, скрипт 452

S

scanf(), функция 101–104, 115, 146, 158, 275

sed, фильтр 145

send(), функция 508

setitimer(), функция 495

short, тип данных 198

sigaction, структура 488

sigaction (), функция 489

SIGALRM, сигнал 494

sizeof, оператор 89, 92, 95

  в сочетании с fgets() 103

  — — — malloc() 316

sleep(), функция 494

static, ключевое слово 579

stderr 158, 160

stdin 158

stdio, библиотека 41, 216

stdout 158, 160

strdup(), функция 321

strerror(), функция 444

string, библиотека 122, 131

strstr(), функция 125

struct, ключевое слово 256

system(), функция 434, 437–439

switch, инструкция 62, 68

T

tail, фильтр 145

telnet, утилита 504

typedef, спецификатор 268, 307

U

unsigned, ключевое слово 200

V

valgrind, утилита 338, 344

void, тип функции 69

W

WEXITSTATUS(), макрос 475, 477

while, цикл 65

waitpid(), функция 474–475

X

XOR-шифрование 218

А

Автоматизированное тестирование 581

Аргумент:

командной строки 177

функции 68, 92

Ассемблер, трансляция кода 220

Б

Библиотека:

внешняя 41

динамическая 409, 411, 422

имена библиотек в MinGW и Cygwin 411

путь на разных платформах 412–413

стандартная 122

Битовые поля 298, 301, 331, 599

Битовые операции 577

В

Веб-камера 428–429

Г

Глобальная область памяти 79, 116

Д

Двоичное дерево 332, 336

Двоичный вид числа 199

Двусвязной список 332

Декремент 576

Дескриптор:

соединения 507

файловый *см.* Файловый дескриптор

Динамическое хранилище 312

Директивы препроцессора 578

Доменное имя 527, 529, 535

Дочерний процесс 456, 462

завершение 474

запуск 457–461

общение с родительским процессом напрямую 478

перенаправление стандартного вывода в файл 471–476

работа с сокетом 522

связывание каналом 479

З

Зависимость 234

Заголовочный файл 122, 209

разделение 392

расположение 216, 391

создание собственного 210, 222

Замена символов для веб-адресов 534

Зарезервированные слова 217

И

Именной ресурс 529

Индекс 49, 97

Инкремент 576

Инструкция:
- блочная 50
- простая 50

Исполняемый файл 38

Исходный файл 38, 41

К

Канал 167, 172, 486
- соединение процессов 479
- создание 480

Ключ 184–185, 396

Код 116
- объектный 38, 221
- позиционно-независимый 410
- сохранение копии 226–227

Комментарий 41, 44

Компилятор 38, 45 *см. также* gcc

Компиляция 38, 75, 225
- автоматизация 234 *см. также* autoconf
- принцип работы 220–221
- условная 578
- частичная 227

Константа 116

Куча 116, 314–315, 330

Л

Линковка 221

Логические выражения 54–56

М

Массивы 47
- адрес 95
- ассоциативный 332
- длина 48–49
- массив массивов 121
- — символов 47, 322
- — указателей 134
- — — на функции 374, 377–378
- переменная *см.* Переменная массивов
- распад 95

Макрос 380, 382

Межпроцессное взаимодействие 476

Множественное присваивание 69

Мьютекс 549–550, 556

Н

Нуль-символ 48–49, 103

О

Обработчик сигнала 487–488

Объединения 283
- использование в структурах 285
- присваивание значения 284

Объектный файл:
- архив 394–396, 399
- разделение 393

Операторы:
- битовые 577
- логические 54

Оптический поток Фарнбага 429

Ошибка со статусом 2 154

П

Память компьютерная 88, 109, 115–116
распределение 410
Параметр:
командной строки 191 *см. также* Ключ
функции *см.* Аргумент функции
Переменная 87
глобальная 83, 87
локальная 83, 87
массивов 90, 95, 97
отличие от указателей 95
разделяемые 222
среды 443
ссылочная 84, 92, 95
Переполнение буфера памяти 102
Перечисление 291
Пользовательский интерфейс 587
Порт 506
выбор номера 508
привязка 506
Поток данных *см. также* Сокет:
для ошибок 156–157
перенаправление 158, 468
создание собственного 174–175
соединение *см.* Канал
стандартный входной 146
перенаправление 147
стандартный выходной 146
перенаправление 148, 468
Потоки 541, 556
запуск 543
Прекомпиляция 216
Препроцессор 382
Присваивание 49 *см. также* Множественное присваивание
Простой инструмент 140
гибкость 160
советы по разработке 165
Протокол 505
Процесс 440, 539
дочерний *см.* Дочерний процесс
идентификатор *см.* PID
перенаправление 469
прерывание 487
принцип работы 467
родительский 456

Р

Рецепт 234

С

Сборка мусора 330
Светофор *см.* Мьютекс
Связной список 305, 310
Си:
основы 590
принцип работы 38, 75
справочные материалы 588

Си++ 75, 88, 284, 301
Сигнал 487–488, 492
порядок принятия 501
сброс и игнорирование 495
эскалация *см.* Эскалация сигнала
Система счисления:
двоичная 199, 297
шестнадцатеричная 297
Системный вызов 434
неудачное выполнение 444
Сокет 506
клиентский 528
Сравнение целых чисел 363
Средства разработки 586
Стек 79, 83, 109, 116, 314–315
Строки 48 *см. также* Строковый литерал
изменение 110
копирование 321
передача в функцию 89
поиск подстроки 125
Строковые функции 124
Строковый литерал 108, 115–116
Структура 256–257, 261 *см. также* Указатель на структуру
вложенная 263
изменение полей 272
имя 271
место в памяти 262
псевдоним 268, 271
рекурсивная 307
считывание полей 258
Считывание клавиатурных нажатий 567

Т

Таблица дескрипторов 469, 477
Табуляция 236, 238
Таймер процесса 494
сброс сигнала 495
Тернарный оператор 576
Типы данных 198
объем занимаемой памяти 199
размер 203–204
структурированные *см.* Структура
Трансформация объектов 571

У

Указатель 78–79, 88
арифметические действия 97–98, 100
на строковый литерал 112
на структуру 275, 277
на функцию 354, 360
преобразование в число 92
разделение памяти 83–84
типы 98
Утилита *см.* Простой инструмент

Ф

Файловый дескриптор 467 *см. также* Таблица дескрипторов
Фильтр 145

Функция:
главная *см.* main()
написание 68
объявление 209
очередность вызова в коде 208
с переменным количеством параметров 380–381, 385
с поддержкой массивов 442
сравнения *см.* qsort()
тело 42, 68

Ц

Целевой файл 234
Цикл:
бесконечный 75
возобновление *см.* continue
прерывание *см.* break
структура 66

Э

Эскалация сигнала 493

Я

Ядро операционной системы 439

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Дэвид Гриффитс
Дон Гриффитс

ИЗУЧАЕМ ПРОГРАММИРОВАНИЕ НА С

Директор редакции *Е. Капьёв*
Ответственный редактор *В. Обручев*
Художественный редактор *Г. Федотов*

ООО «Издательство «Эксмо»
127299, Москва, ул. Клары Цеткин, д. 18/5. Тел. 411-68-86, 956-39-21.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 127299, Мәскеу, Клара Цеткин көшесі, 18/5 үй.
Тел. 8 (495) 411-68-86, 8 (495) 956-39-21.
Home page: www.eksmo.ru . E-mail: info@eksmo.ru.
Қазақстан Республикасындағы Өкілдігі: «РДЦ-Алматы» ЖШС, Алматы қаласы,
Домбровский көшесі, 3«а», Б литері, 1 кеңсе. Тел.: 8(727) 2 51 59 89,90,91,92,
факс: 8 (727) 251 58 12 ішкі 107; E-mail: RDC-Almaty@eksmo.kz
Қазақстан Республикасының аумағында өнімдер бойынша шағымды Қазақстан
Республикасындағы Өкілдігі қабылдайды: «РДЦ-Алматы» ЖШС,
Алматы қаласы, Домбровский көшесі, 3«а», Б литері, 1 кеңсе.
Өнімдердің жарамдылық мерзімі шектелмеген.

Подписано в печать 19.12.2012. Формат 84х108 1/16.
Печать офсетная. Усл. печ. л. 65,52.
Тираж 2 000 экз. Заказ 4314

Отпечатано с электронных носителей издательства.
ОАО «Тверской полиграфический комбинат». 170024, г. Тверь, пр-т Ленина, 5.
Телефон: (4822) 44-52-03, 44-50-34, Телефон/факс: (4822) 44-42-15.
Home page – www.tverpk.ru Электронная почта (E-mail) sales@tverpk.ru

ISBN 978-5-699-60233-9

Оптовая торговля книгами «Эксмо»:
ООО «ТД «Эксмо». 142700, Московская обл., Ленинский р-н, г. Видное, Белокаменное ш., д. 1, многоканальный тел. 411-50-74.
E-mail: **reception@eksmo-sale.ru**

По вопросам приобретения книг «Эксмо» зарубежными оптовыми покупателями *обращаться в отдел зарубежных продаж ТД «Эксмо»*
E-mail: **international@eksmo-sale.ru**

International Sales: *International wholesale customers should contact Foreign Sales Department of Trading House «Eksmo» for their orders.*
international@eksmo-sale.ru

По вопросам заказа книг корпоративным клиентам, в том числе в специальном оформлении,
обращаться по тел. 411-68-59, доб. 2299, 2205, 2239, 1251.
E-mail: **vipzakaz@eksmo.ru**

Оптовая торговля бумажно-беловыми и канцелярскими товарами для школы и офиса «Канц-Эксмо»:
Компания «Канц-Эксмо»: 142702, Московская обл., Ленинский р-н, г. Видное-2, Белокаменное ш., д. 1, а/я 5. Тел./факс +7 (495) 745-28-87 (многоканальный).
e-mail: **kanc@eksmo-sale.ru**, сайт: **www.kanc-eksmo.ru**

Полный ассортимент книг издательства «Эксмо» для оптовых покупателей:
В Санкт-Петербурге: ООО СЗКО, пр-т Обуховской Обороны, д. 84Е. Тел. (812) 365-46-03/04.
В Нижнем Новгороде: Филиал ООО «Торговый Дом «Эксмо» в Нижнем Новгороде, ул. Маршала Воронова, д. 3. Тел. (8312) 72-36-70.
В Ростове-на-Дону: Филиал ООО «Издательство «Эксмо» в г. Ростове-на-Дону, пр-т Стачки, 243 «А». Тел. +7 (863) 305-09-12/13/14.
В Самаре: ООО «РДЦ-Самара», пр-т Кирова, д. 75/1, литера «Е». Тел. (846) 269-66-70.
В Екатеринбурге: ООО «РДЦ-Екатеринбург», ул. Прибалтийская, д. 24а. Тел. +7 (343) 272-72-01/02/03/04/05/06/07/08.
В Новосибирске: ООО «РДЦ-Новосибирск», Комбинатский пер., д. 3. Тел. +7 (383) 289-91-42. E-mail: **eksmo-nsk@yandex.ru**
В Киеве: ООО «РДЦ Эксмо-Украина», Московский пр-т, д. 6. Тел./факс: (044) 498-15-70/71.
В Донецке: ул. Артема, д. 160. Тел. +38 (062) 381-81-05.
В Харькове: ул. Гвардейцев Железнодорожников, д. 8. Тел. +38 (057) 724-11-56.
Во Львове: ул. Бузкова, д. 2. Тел. +38 (032) 245-01-71.
Интернет-магазин: www.knigka.ua. Тел. +38 (044) 228-78-24.
В Казахстане: ТОО «РДЦ-Алматы», ул. Домбровского, д. 3а. Тел./факс (727) 251-59-90/91. RDC-Almaty@eksmo.kz

Полный ассортимент продукции издательства «Эксмо»
можно приобрести в магазинах «Новый книжный» и «Читай-город».
Телефон единой справочной: 8 (800) 444-8-444.
Звонок по России бесплатный.

В Санкт-Петербурге в сети магазинов «Буквоед»:
«Парк культуры и чтения», Невский пр-т, д. 46. Тел. (812) 601-0-601
www.bookvoed.ru

По вопросам размещения рекламы в книгах издательства «Эксмо» обращаться в рекламный отдел. Тел. 411-68-74.

Интернет-магазин ООО «Издательство «Эксмо»
www.fiction.eksmo.ru
Розничная продажа книг с доставкой по всему миру.
Тел.: +7 (495) 745-89-14 . E-mail: **imarket@eksmo-sale.ru**

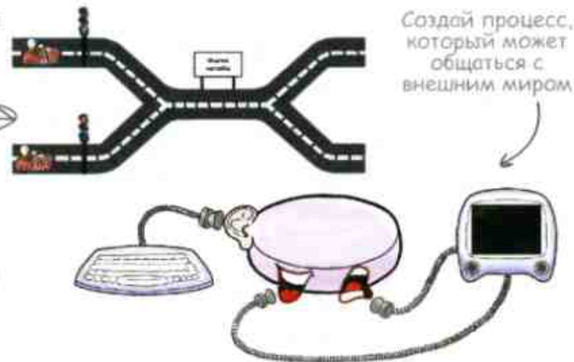
ИЗУЧАЕМ ПРОГРАММИРОВАНИЕ НА C

Языки программирования C

О чем эта книга?

Вы всегда мечтали о том, чтобы найти более легкий способ изучения программирования на C? «Изучаем C» предлагает методику, с помощью которой вы научитесь создавать программы на этом языке. В книге используется уникальный подход, который выходит за рамки синтаксиса и пошаговых руководств и поможет вам стать отличным программистом. Вы изучите ключевые моменты, в том числе основы языка, динамическое управление памятью, указатели и арифметические операции с ними. А благодаря более продвинутым темам, таким как многопоточность и сетевое программирование, «Изучаем C» может рассматриваться в качестве учебника для студентов.

В книге есть практические задания — проекты, которые должны усовершенствовать ваши способности, проверить приобретенные вами навыки и сделать вас более уверенным в себе. Изучив основы языка, вы научитесь пользоваться компилятором, утилитой make и упаковщиком, чтобы решать реальные проблемы.



Почему эта книга столь не похожа на другие?

Мы уверены, что ваше время слишком ценно, чтобы тратить его на попытки понять сложные концепции. Книга основана на самых последних достижениях в теории обучения и восприятия. В отличие от скучных текстовых книг, она задействует в процессе обучения сразу несколько органов чувств, сделав усвоение материала максимально эффективным.

«Изучаем C» вполне может стать лучшей книгой о C на все времена. Я легко могу представить ее в качестве стандартного учебника для любого университетского курса по C. Большинство книг о программировании выполнено в довольно предсказуемом стиле. В этой же книге совершенно иной подход. Она сделает из вас настоящего программиста на языке C.

— ***Дейв Китабян***, *руководитель отдела разработки программного обеспечения NetCarrier Telecom*

«Изучаем C» — это доступное и увлекательное введение в программирование на C. Иллюстрации, шутки, упражнения и практические задания — все это незаметно, но настойчиво знакомит читателя с основами языка C... после чего окунает его в более продвинутые темы на примере системного программирования для Linux и Posix.

— ***Винс Милнер***, *разработчик программного обеспечения*

ISBN 978-5-699-60233-9

ЭКСМО

O'REILLY®