Основы разработки на C++: жёлтый пояс

Неделя 3

Разделение кода по файлам

Оглавление

Разделение кода по файлам

Распределение кода по файлам

Введение в разработку в нескольких файлах на примере задачи «Синонимы»

До этого мы весь код хранили в одном файле. Но в общем случае это приводит к проблемам:

1. Для использования одного и того же кода в нескольких программах его приходится копировать;
2. Даже самое маленькое изменение программы приводит к её полной перекомпиляции;

Как говорит автор языка C++ Бьёрн Страуструп в своей книге «Язык программирования C++»: «Разбиение программы на модули помогает подчеркнуть ее логическую структуру и облегчает понимание». Рассмотрим это всё на примере кода нашей программы из прошлой недели. Здесь у нас есть логически не связанные друг с другом вещи. Первый кусок – само решение задачи «Синонимы»:

```
using Synonyms = map <string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
  const string& second_word) {
  synonyms[second_word].insert(first_word);
  synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms,
  const string& first_word) {
  return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
```

```
  const string& second_word) {
  return synonyms[first_word].count(second_word) == 1;
}
```

Далее идёт наш юнит-тест фреймворк:

```
template <class T>
ostream& operator<<(ostream& os, const set<T>& s) {
  os << "{";
  bool first = true;
  for (const auto& x : s) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << x;
  }
  return os << "}";
}
template <class K, class V>
ostream& operator<<(ostream& os, const map<K, V>& m) {
  os << "{";
  bool first = true;
  for (const auto & kv : m) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << kv.first << ": " << kv.second;
  }
  return os << "}";
}
template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
    os << "Assertion failed: " << t << " != " << u <<
      " hint: " << hint;
    throw runtime_error(os.str());
  }
}
```

```
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}
class TestRunner {
  public:
    template<class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {
      try {
        func();
        cerr << test_name << " OK" << endl;
      } catch (runtime_error & e) {
        ++fail_count;
        cerr << test_name << " fail: " << e.what() << endl;
      }
    }
    ~TestRunner() {
      if (fail_count > 0) {
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1);
      }
    }
  private:
    int fail_count = 0;
};
```

Весь юнит-тест фреймворк логически не зависит от функций, которые решают нашу задачу.

Следующая логически независимая часть программы – это сами юнит-тесты:

```
void TestAddSynonyms() {
  {
    Synonyms empty;
    AddSynonyms(empty, "a", "b");

    const Synonyms expected = {
      {"a", {"b"}},
      {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Empty");
  }
  {
```

```
    Synonyms synonyms = {
      {"a", {"b"}},
      {"b", {"a", "c"}},
      {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");

    const Synonyms expected = {
      {"a", {"b", "c"}},
      {"b", {"a", "c"}},
      {"c", {"b", "a"}}
    };
    AssertEqual(synonyms, expected, "Nonempty");
  }
}

void TestCount() {
  {
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty,"a"), 0u, "Syn. count for empty dict");
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms,"a"), 2u, "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms,"b"), 1u, "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms,"z"), 0u, "Nonempty dict, count z");
  }
}

void TestAreSynonyms() {
  {
    Synonyms empty;
    Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
    Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
  }
  {
    Synonyms synonyms = {
```

```
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
    Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
    Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
    Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
    Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
    Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
  }
}
void TestAll() {  // объединяем запуск всех юнит-тестов
  TestRunner tr;
  tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
  tr.RunTest(TestCount, "TestCount");
  tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}
```

Ещё у нас есть `main`:

```
int main() {
  TestAll();

  int q;
  cin >> q;
  Synonyms synonyms;

  for (int i = 0; i < q; ++i) {
    string operation_code;
    cin >> operation_code;

    if (operation_code == "ADD") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      AddSynonyms(synonyms, first_word, second_word);
    } else if (operation_code == "COUNT") {
      string word;
      cin >> word;
      cout << GetSynonymCount(synonyms, word) << endl;
    } else if (operation_code == "CHECK") {
```

```
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
      } else {
        cout << "NO" << endl;
      }
    }
  }
  return 0;
}
```

Теперь рассмотрим вынесение в отдельные файлы в Eclipse. Итак, у нас в программе есть 4 логически обособленных компонента:

1. Функции решения нашей задачи;
2. Юнит-тест фреймворк;
3. Сами юнит-тесты;
4. Решение нашей задачи в main.

И довольно логично отделить эти части друг от друга, поместив их в отдельные файлы. Открываем наш проект Coursera: *Window* → *Show View* → *C/C++ Projects* (как это сделано на рис. 2.1). Нажимаем на него **project name** → *New* → *Header File*. (см. 2.2) Вводим имя заголовочному файлу (test_runner.h) и у нас создаётся пустой файл test_runner.h.

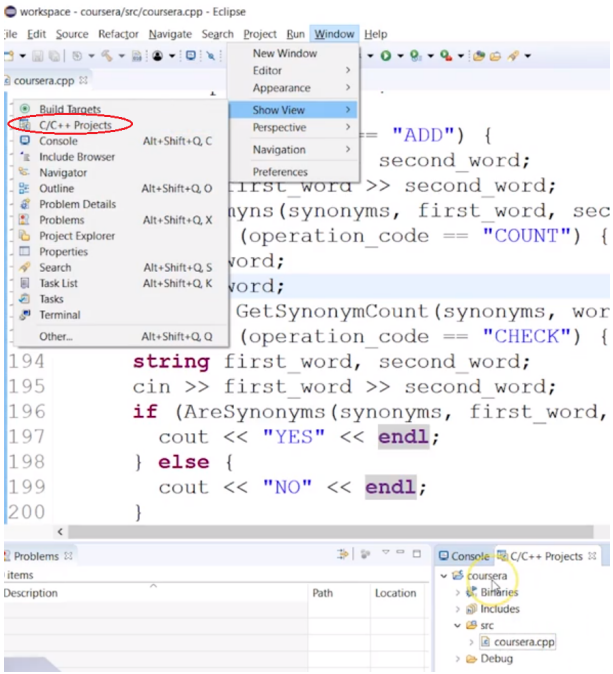


Рис. 2.1: Открытие C/C++ projects

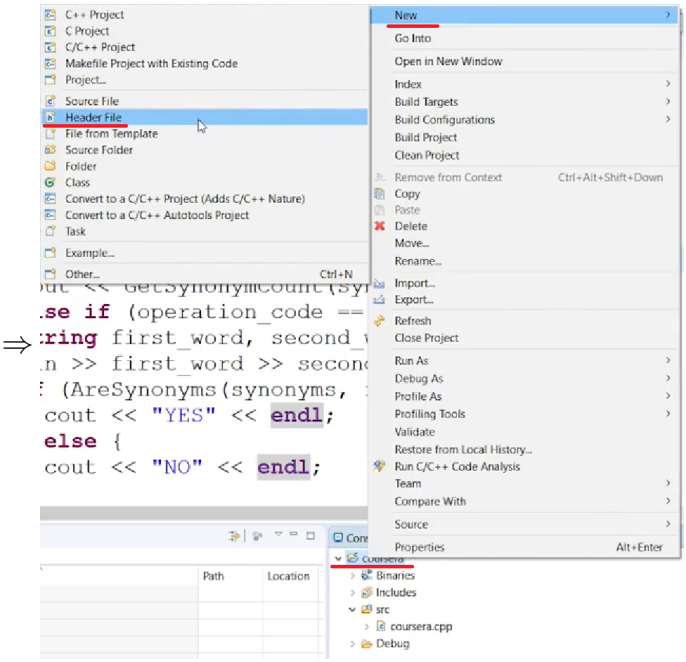


Рис. 2.2: New → Header File

Теперь из основного монолитного файла решения задачи «Синонимы» вырезаем сами юнит-тесты в test_runner.h. Теперь запустим нашу программу и она не скомпилируется, потому что мы, как минимум, не знаем, что такое `Assert`. Нам надо дописать в начало нашей программы

```
#include "test_runner.h"  // подключаем файл с юнит-тестами
```

Теперь всё компилируется и работает. Аналогичным образом в файл synonyms.h вынесем функции самого решения задачи, а в файл tests.h вынесем все юнит-тесты и допишем:

```
#include "synonyms.h"
#include "tests.h"
```

Программа компилируется и тесты выполняются. Таким образом мы смогли разбить исходную программу на 4 файла, в каждом из которых лежат независимые блоки.

Механизм работы директивы `#include`

Несмотря на кажущуюся корректность в выполнении этих операций, у нас есть немало проблем. И давайте посмотрим, какие это проблемы. Для примера закомментируем `#include <set>` в начале нашей программы:

```
#include <cassert>
#include <sstream>
#include <exception>
#include <iostream>
#include <string>
#include <map>
#include <vector>
// #include <set> // закомментировали
using namespace std;

#include "test_runner.h"
#include "synonyms.h"
#include "tests.h"

int main() {
  ...
}
// 'AddSynonyms' was not declared in this scope...
```

И ещё несколько ошибок. Компилятор пишет, что мы не объявили фукцию `AddSynonyms`, хотя мы её объявляли в test_runner.h. Перед нами встаёт проблема: мы не можем понять, где именно возникает ошибка.

Теперь посмотрим на другую. Поменять инклюды из начала мы можем без проблем. А вот если сделать подключение `tests` не третьим, а вторым, программа снова выдаст нам ошибку.

Третья демонстрация: если мы перенесём подключения в начало программы (например, между подключениями `sstream` и `exception`), снова появится куча ошибок о необъявленных переменных.

Разберёмся, как работает `#include`:

- Директива `#include "file.h"` вставляет содержимое файла file.h в месте использования;
- Файл, полученный после всех включений, подаётся на вход компилятору.

Разберёмся на примере маленького проектика Sum. У нас есть два файла: how_include_works.cpp с самой программой...

```
#include "sum.h"
int main() {
  int k = Sum(3, 4);
  return 0;
}
```

...в которой подключается sum.h с функцией суммирования:

```
int Sum(int a, int b) {
  return a + b;
}
```

Переключимся в консоль операционной системы. Зайдём в нашу директорию и увидим там два файла: how_include_works.cpp и sum.h. (рис. 2.3)

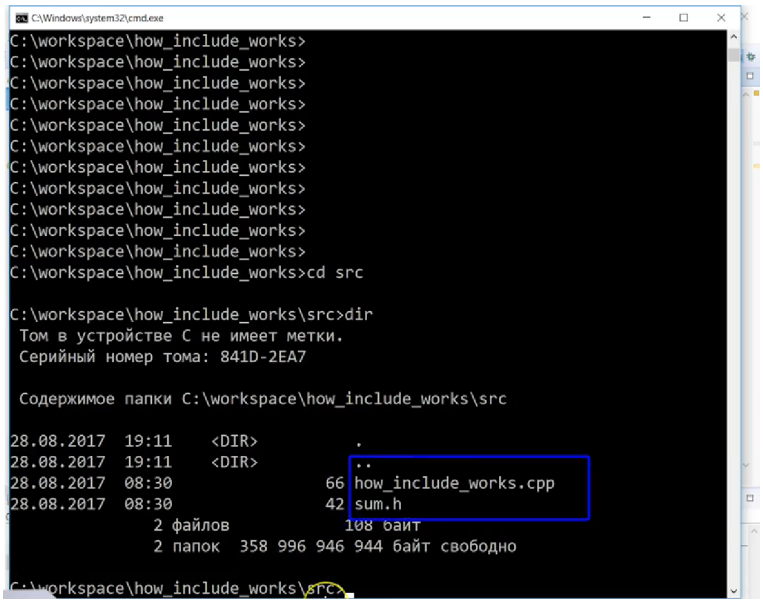


Рис. 2.3: Консоль `cmd`

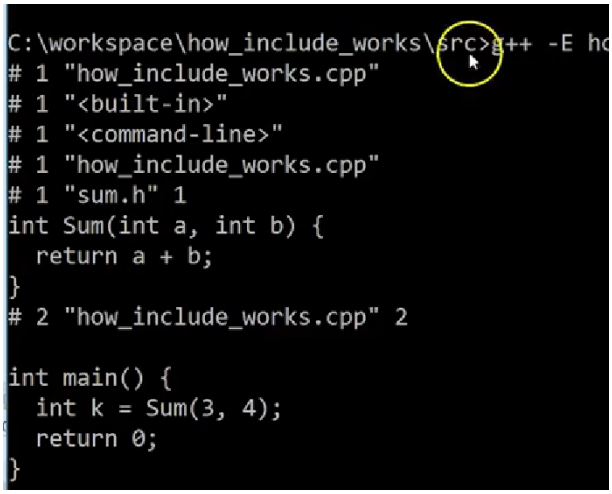


Рис. 2.4: Препроцессинг проекта

Вызовем команду компилятора:

```
g++ -E how_include_works.cpp
```

Вызов компилятора с флагом `-E` значит, что мы просим компилятор не выполнять полную

сборку проекта, а просто выполнить стадию препроцессинга (стадию выполнения директив `#include`). В итоге мы видим, что в файле есть функция `main`, а выше вставлен sum.h (рисунок 2.4). За символом `#` – уже служебные символы компилятора.

Теперь вернёмся к нашему большому проекту и посмотрим, как препроцессинг работает на нашем проекте тем же образом: в терминале `cmd.exe` переходим в директорию проекта и вводим:

```
g++ -E coursera.cpp > coursera.i
```

(чтобы результат препроцессинга вывелся в файл coursera.i)

Размер файла оказался 37980 строк после отрабатывания директив `include`. Содержимое каждого модуля было вставлено в файл с исходником. И само наше решение (`main` и все файлы, в которые мы до этого выносили части кода) начинается только с 37780 строки. А всё до этого – модули стандартных библиотек.

Отсюда и ответ на все те проблемы, которые мы получали: если мы убирали какую-то стандартную библиотеку, например `#include <set>`, нигде не было написано, что `set` – это множество, какие у него есть операции. И поэтому у нас возникала ошибка компиляции. При переносе тоже была ошибка, потому что в заголовочных файлах мы использовали функции и структуры, которые включались позже, и компилятор не мог их найти.

Обеспечение независимости заголовочных файлов

Избавляемся от одной из проблем, описанных выше. Нам надо, чтобы наши файлы были независимыми и порядок включения не влиял на компилируемость программы.

Решение: включим в каждый файл проекта те заголовочные файлы, которые ему нужны. Начнём с test_runner.h. Ему нужны `set`, `map`, `ostream` и `string`. Просто перенесём эти включения из основного файла в test_runner.h:

```
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std;  // попытаемся добавить, хотя так делать не стоит
```

Программа компилируется. Тогда попробуем поставить `#include "test_runner.h"` самым первым в нашем основном файле. Но программа не компилируется, потому что файлу synonyms.h нужно знать `map`, `string`, `set`, а он об этом сейчас не знает, ведь мы подключаем файлы в данном порядке:

```
#include "synonyms.h"
#include "test_runner.h"

#include <exception>
#include <iostream>
#include <vector>

using namespace std;

#include "tests.h"
```

Раньше synonyms.h стоял после test_runner.h и получал из него все нужные `include`'ы. Теперь поставим и его (synonyms.h) вперёд и добавим все необходимые `include`'ы:

```
#include <map>
#include <set>
#include <string>
using namespace std;  // попытаемся добавить, хотя так делать не стоит
```

Теперь всё компилируется.

Рассмотрим функцию `main()`. Он состоит из функции `TestAll()` и кода, который решает задачу. В этом конкретном файле мы нигде не используем наш фреймворк. Значит, test_runner.h нам в этом файле не нужен. Он нужен в tests.h, потому что именно они используют тестовый фреймворк. Таким образом мы сделали test_runner.h и synonyms.h независимыми, и подключать их можно в любом порядке до функции `main()`.

Проблема двойного включения

Функция `AddSynonyms()` в tests.h определена в synonyms.h, и если мы поставим tests.h перед synonyms.h, наш проект не скомпилируется. Тогда добавим в tests.h все зависимости, в частности, synonyms.h и скомпилируем:

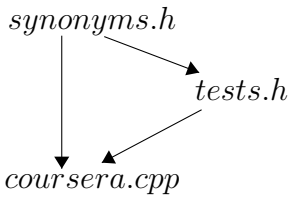
```
#include "test_runner.h"
```

```
#include "synonyms.h"
...
// redefinition of "bool AreSynonyms...."
```

Программа не компилируется, причём со странными ошибками о переопределении наших функций. Для того, чтобы понять, что произошло, вернёмся к маленькой задачке. Продублируем строчку `#include "sum.h"`.

```
#include "sum.h"
#include "sum.h"
int main() {
  int k = Sum(3, 4);
  return 0;
}
// redefinition of "int Sum...."
```

После компиляции увидим ту же ошибку. Теперь получим препроцессинг проекта, как на рисунках 2.3 и 2.4. Заметим, что в файле получилось две функции `sum`. Когда компилятор это видит, он выкидывает ошибку компиляции `Redefinition`, т.е. повторное определение. Точно та же ситуация у нас в большом проекте: в coursera.cpp включается tests.h, который подключает synonyms.h, который так же включается в coursera.cpp. Таким образом у нас получается переопределение всего synonyms.h.



Избежать двойного включения очень просто: добавляем в начало каждого заголовочного файла `"#pragma once"`. В нашем случае дописываем в synonyms.h (и уже сейчас всё заработает), tests.h и test_runner.h. Эта директива говорит компилятору игнорировать все повторные включения.

Также добавим в sum.h эту строчку и проверим, что всё работает. Выполним его препроцессинг и увидим, что функция `sum` там встречается только один раз. Здесь мог возникнуть вопрос: «Почему препроцессор не отслеживает, что заголовочные файлы включаются несколько раз, и почему препроцессор по умолчанию не выкидывает повторные включения?» Потому что C++ делался обратно совместимым с C, и вообще C++ развивается так, чтобы не терять обратную совместимость. Но мы можем забывать каждый раз прописывать эту строчку в каждом заголовочном файле, и тут нам на помощь приходит IDE. В Eclipse оно работает так: *Window* → *Preferences* → *C/C++* → *Code Style* → *Code Templates* → *Files* → *C++ Header File* → *Default C++ header template*, там нажимаем Edit и у нас открывается окно ввода шаблона, который будет вставляться во все заголовочные файлы, которые мы создаём. Сюда можно добавлять специальные макропеременные, которые вставляют ваше имя, дату создания файла, имя проекта и так далее. Но вот мы сюда прямо и напишем `#pragma once`, перевод строки. OK, Apply,

Apply and Close. Снова создадим новый header-файл, как на рисунке 2.2. Как только мы его создали, он по умолчанию сразу идёт с вставленным шаблоном.

Понятия объявления и определения

Когда у нас есть большой проект, в котором много файлов, то мы, естественно, не можем помнить досконально, в каком файле какие функции есть. И очень часто хочется, открыв файл, понять интерфейс этого файла, то есть понять, какие функции и классы в этом файле есть. Т. е. зайти в файл и сразу увидеть его интерфейс. Нам придётся пролистать весь файл, чтобы понять, что за функции в нём есть. Хотелось бы короткий список функций. В Eclipse можно нажать Ctrl+O и получить краткий список с названиями и типами функций.

Иногда хочется видеть только интерфейс – список функций и классов, которые там есть. Нас не будет интересовать, как это работает (допускаем, что оно работает). Нас интересует только, что мы с ним можем делать. Введём два новых определения:

- **Объявление функции (function declaration)** – сигнатура функции (возвращаемый тип, имя функции и список параметров с типами). Оно говорит, что где-то в программе есть функция с заданными параметрами;

```
int GreatestCommonDivisor(int a, int b);
```

- **Определение функции (function definition)** – сигнатура + реализация функции.

```
int GreatestCommonDivisor(int a, int b) {
  while (a > 0 && b > 0) {
    if (a > b) {
       a %= b;
    } else {
       b %= a;
    }
  }
  return a + b;
}
```

Функция может быть объявлена несколько раз, но определена должна быть только в одном месте. Ещё важно, чтобы все объявления функции были одинаковыми.

На простом примере разберёмся, как это работает. Итак, у нас есть

```
void foo() {
  bar();
}
void bar() {
}
int main() {
  return 0;
}
// "bar" was not declared in this scope
```

Функция `bar` не объявлена и файл не компилируется. Теперь в самом начале файла объявим функцию `bar`:

```
void bar();  // добавили в самое начало программы
```

Теперь всё заработало. И даже если объявлений будет много, программа будет компилироваться. А вот если мы продублируем определение, то всё сломается и мы получим redefinition error. Это было насчёт определения и объявления функций. Аналогично у нас будет и для классов:

- **Объявление класса (class declaration)** – объявление класса, его поля и методы. Но методы не реализованы:

```
class Rectangle {
public:
  Rectangle(int width, int height);
  int Area() const;
  int Perimeter() const;

private:
  int width, height;
};
```

- **Определение методов класса (class methods definition)**

```
Rectangle::Rectangle(int w, int h) {  // по принципу: имя класса::имя метода
  width = w;
  height = h;
}
int Rectangle::Area() const {
```

```
  return width * height;
}
int Rectangle::Perimeter() const{
  return 2 * (width + height);
}
```

Теперь вспомним, а зачем оно нам: мы хотели в начале файла видеть объявления всех функций и классов, которые есть в файле. Сделаем это для нашего большого проекта. Допишем в tests.h:

```
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();
```

Теперь аналогично сделаем для synonyms.h и test_runner.h. Причём во второй у нас есть шаблоны функций, которые точно так же стоит объявить в начале:

```
void AddSynomyns(Synonyms& synonyms,
  const string& first_word, const string& second_word);
bool AreSynonyms(Synonyms& synonyms,
  const string& first_word, const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);
```

```
template <class T> // копируем объявление шаблонов
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
  public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name);
    ~TestRunner();

  private:
```

```
    int fail_count = 0;
};
```

Итоги:

- Объявление в начале файла сообщает компилятору, что функция/класс/шаблон где-то определены;
- Объявлений может быть несколько. Определение – только одно;
- Группировка объявлений в начале файла позволяет узнать, какие функции и классы в нём есть, не вникая в их реализацию.

Механизм сборки проектов, состоящих из нескольких файлов

Когда мы начинали разговор о разделении кода на несколько файлов, то в качестве одного из недостатков хранения всего кода в одном файле мы называли то, что при минимальном изменении программы у нас она пересобирается вся, в случае, если весь код лежит в одном файле. Сейчас мы разделили код нашего проекта на целых четыре файла. Но при этом каждый раз, когда мы меняем что угодно в нашем проекте, он все равно пересобирается целиком. Почему это происходит? Потому что у нас есть файл coursera.cpp, в который так или иначе включаются с помощью директивы `#include` три других наших файла. Соответственно, если мы в них что-нибудь меняем, то они вставляются в наш coursera.cpp, и вся программа перекомпилируется целиком. Но давайте подумаем. Например, есть у нас функции в `"synonyms.h"`, которые умеют работать со словарём синонимов – добавлять в него, проверять количество синонимов. Есть эти функции и есть тесты на них. Если мы внесем изменения в тесты, например, добавим какой-нибудь ещё тестовый случай, например, в тест на `TestCount`, давайте там проверим, что при пустом словаре и для строки `b` у нас тоже вернётся ноль. Если мы поменяли тесты, то нам нет никакой необходимости перекомпилировать сами функции. Но мы все равно перекомпилируем всё.

Нам надо не пересобирать проект целиком при изменении в конкретном месте. Разберёмся в механике сборки проектов в C++. Посмотрим на расширения файлов в нашем проекте:

- **tests.h**
- **synonyms.h**

- **test_runner.h**
- **coursera.cpp**

Пока у нас 3 файла **.h** и только один файл **.cpp**. Добавим ещё один, как на картинке 2.2: *project name* → *New* → *Source File* и назовём его one_more.cpp. Очистим результаты сборки (*project name* → *Clean Project*) и соберём проект с нуля. Запустим сборку и после её завершения посмотрим, какие команды выполнял Eclipse в процессе сборки проекта:

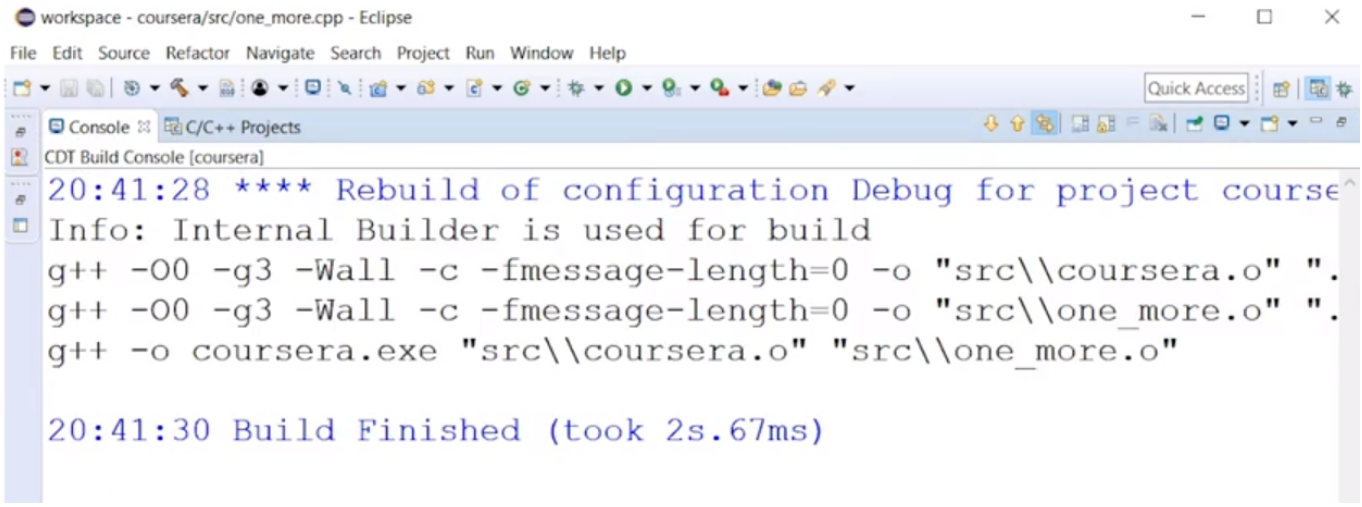


Рис. 2.5: Команды по сборке проекта

Первый раз он запускался для файла coursera.cpp. Второй раз он запускался для нашего только что добавленного файла one_more.cpp. В результате было получено два файла с расширением .o. Вот этот параметр -o задает имя выходного файла, поэтому по значению параметра -o мы можем понимать, какие выходные файлы формировались в этой стадии. И потом была третья стадия, в которой на вход были поданы вот эти файлы с расширением .o, а на выходе получился исполняемый файл coursera.exe. Этот пример демонстрирует, каким образом выполняется сборка проектов на C++, состоящих из нескольких файлов.

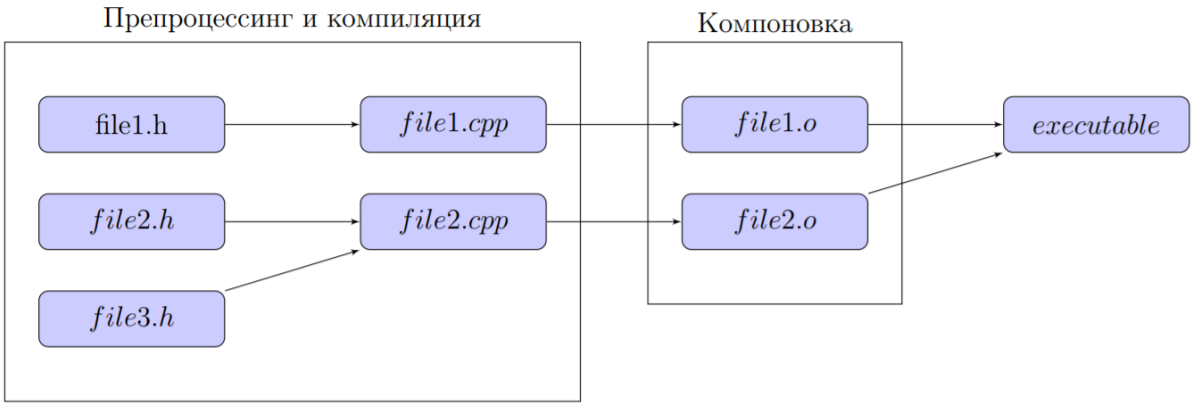


Рис. 2.6: Компиляция нескольких файлов

Как мы уже видели, первая стадия – это препроцессинг, когда выполняются все директивы `include`. Дальше, после того как препроцессинг выполнен, берётся каждый отдельный .cpp файл и компилируется. В результате компиляции каждого .cpp файла получается так называемый объектный файл. Вот на схеме (рисунок 2.6) у нас объектные файлы изображены как файлы с расширением .o. И затем начинается третья стадия – это стадия компоновки, когда берутся все объектные файлы, которые у нас получились, и компонуются в один исполняемый файл.

Теперь, если мы в наш one_more.cpp добавим какое-нибудь изменение, например, комментарий, запустим сборку и посмотрим на команды консоли, видим, что теперь вместо всего проекта перекомпилировался только one_more.cpp и потом был собран исходный файл coursera.exe. Аналогично внесём изменения в coursera.cpp и запустим сборку. В консоли увидим, что one_more.cpp не был тронут, и перекомпилировался только coursera.cpp. Если мы изменим любой из .h файлов, подключаемых в coursera.cpp, увидим то же самое.

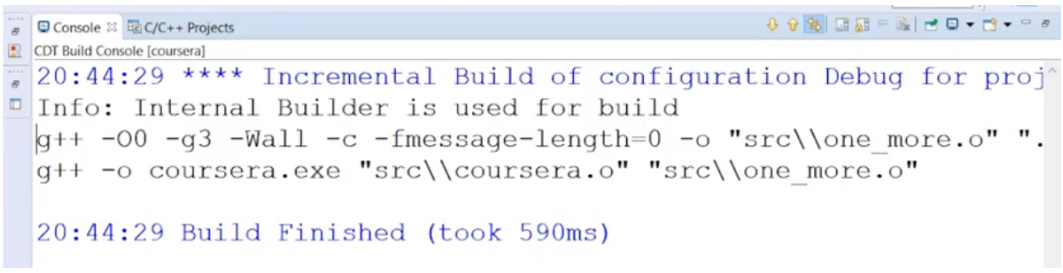


Рис. 2.7: Сообщения в консоли

Вывод:

1. При сборке проекта компилируются только изменённые **.cpp**-файлы;
2. Внесённые изменения в **.h** файл приводит к перекомпиляции всех **.cpp**-файлов, в которые он включён;
3. Если перенести определения функций и методов классов в **.cpp**-файлы, то они будут пересобираться только после изменений.

Теперь используем эти знания на нашем проекте, чтобы при небольшом изменении наш код реже пересобирался. Определения функций и методов классов переносим в .cpp файлы, а в заголовочных файлах оставляем только объявления. Мы логически не связанные друг с другом определения разнесём в разные файлы. Когда мы меняем, например, определения тестов, то определения функций, которые эти тесты покрывают, не меняются, и соответственно, они не будут перекомпилироваться. Таким образом мы минимизируем количество .cpp-файлов, которые нужно перекомпилировать при каждом изменении программы.

Давайте выполним такое преобразование с нашим проектом, то есть вынесем определение в .cpp-файл. И начнем вот, например, с test_runner.h. Добавим в наш проект файл test_runner.cpp. И вынесем в него определения. Здесь есть нюанс (далее в курсе мы это разберём) с шаблонами, так что пока их переносить в .cpp-файл мы не будем.

```
#include "test_runner.h"
void Assert(bool b, const string& hint) {
  AssertEqual(b, true, hint);
}
TestRunner::~TestRunner() {
  if (fail_count > 0) {
    cerr << fail_count << " unit tests failed. Terminate" << endl;
    exit(1);
  }
}
```

Таким же образом с synonyms.cpp и tests.cpp:

```
#include "synonyms.h"
void AddSynonyms(Synonyms& synonyms, const string& first_word,
  const string& second_word) {
  synonyms[second_word].insert(first_word);
```

```
  synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word) {
  return synonyms[first_word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
  const string& second_word) {
  return synonyms[first_word].count(second_word) == 1;
}
```

```
#include "tests.h"
void TestAddSynonyms() {
  {
    Synonyms empty;
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
      {"a", {"b"}},
      {"b", {"a"}},
    };
    AssertEqual(empty, expected, "Empty");
  }
  {
    Synonyms synonyms = {
      {"a", {"b"}},
      {"b", {"a", "c"}},
      {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
      {"a", {"b", "c"}},
      {"b", {"a", "c"}},
      {"c", {"b", "a"}}
    };
    AssertEqual(synonyms, expected, "Nonempty");
  }
}

void TestCount() {
  {
    Synonyms empty;
    AssertEqual(GetSynonymCount(empty, "a"), 0u,
```

```
      "Syn. count for empty dict");
    AssertEqual(GetSynonymCount(empty, "b"), 0u,
      "Syn. count for empty dict b");
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    AssertEqual(GetSynonymCount(synonyms, "a"), 2u,
      "Nonempty dict, count a");
    AssertEqual(GetSynonymCount(synonyms, "b"), 1u,
      "Nonempty dict, count b");
    AssertEqual(GetSynonymCount(synonyms, "z"), 0u,
      "Nonempty dict, count z");
  }
}

void TestAreSynonyms() {
  {
    Synonyms empty;
    Assert(!AreSynonyms(empty, "a", "b"), "AreSynonyms empty a b");
    Assert(!AreSynonyms(empty, "b", "a"), "AreSynonyms empty b a");
  }
  {
    Synonyms synonyms = {
      {"a", {"b", "c"}},
      {"b", {"a"}},
      {"c", {"a"}}
    };
    Assert(AreSynonyms(synonyms, "a", "b"), "AreSynonyms nonempty a b");
    Assert(AreSynonyms(synonyms, "b", "a"), "AreSynonyms nonempty b a");
    Assert(AreSynonyms(synonyms, "a", "c"), "AreSynonyms nonempty a c");
    Assert(AreSynonyms(synonyms, "c", "a"), "AreSynonyms nonempty c a");
    Assert(!AreSynonyms(synonyms, "b", "c"), "AreSynonyms nonempty b c");
    Assert(!AreSynonyms(synonyms, "c", "b"), "AreSynonyms c b");
  }
}

void TestAll() {
```

```
  TestRunner tr;
  tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
  tr.RunTest(TestCount, "TestCount");
  tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}
```

А в самих .h файлах у нас остаётся:

```
#pragma once
#include "test_runner.h"
#include "synonyms.h"
void TestAddSynonyms();
void TestAreSynonyms();
void TestCount();
void TestAll();
```

```
#pragma once
#include <map>
#include <set>
#include <string>
using namespace std;
using Synonyms = map<string, set<string>>;
void AddSynonyms(Synonyms& synonyms, const string& first_word,
  const string& second_word);
bool AreSynonyms(Synonyms& synonyms, const string& first_word,
  const string& second_word);
size_t GetSynonymCount(Synonyms& synonyms, const string& first_word);
```

```
#pragma once
#include <string>
#include <set>
#include <map>
#include <iostream>
#include <sstream>
using namespace std;

template <class T>
ostream& operator << (ostream& os, const set<T>& s);

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m);
```

```
template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint);

void Assert(bool b, const string& hint);

class TestRunner {
public:
  template <class TestFunc>
  void RunTest (TestFunc func, const string & test_name);
  ~TestRunner();
private:
  int fail_count = 0;
};

template <class T>
ostream& operator << (ostream& os, const set<T>& s) {
  os << "{";
  bool first = true;
  for (const auto& x : s) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << x;
  }
  return os << "}";
}

template <class K, class V>
ostream& operator << (ostream& os, const map<K, V>& m) {
  os << "{";
  bool first = true;
  for (const auto & kv:m) {
    if (!first) {
      os << ", ";
    }
    first = false;
    os << kv.first << ": " << kv.second;
  }
  return os << "}";
```

```
}

template <class T, class U>
void AssertEqual(const T& t, const U& u, const string& hint) {
  if (t != u) {
    ostringstream os;
    os << "Assertion failed: " << t << " != " << u << " hint: " << hint;
    throw runtime_error(os.str());
  }
}

template <class TestFunc>
void TestRunner::RunTest (TestFunc func, const string& test_name) {
  try {
    func ();
    cerr << test_name << " OK" << endl;
  } catch (runtime_error & e) {
    ++fail_count;
    cerr << test_name << " fail: " << e.what () << endl;
  }
}
```

Вспомним, что у нас есть и основной файл с решением:

```
#include "tests.h"
#include "synonyms.h"
#include <exception>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  TestAll();
  int q;
  cin >> q;
  Synonyms synonyms;
  for (int i = 0; i < q; ++i) {
    string operation_code;
    cin >> operation_code;

    if (operation_code == "ADD") {
      string first_word, second_word;
```

```
      cin >> first_word >> second_word;
      AddSynonyms(synonyms, first_word, second_word);
    } else if (operation_code == "COUNT") {
      string word;
      cin >> word;
      cout << GetSynonymCount(synonyms, word) << endl;
    } else if (operation_code == "CHECK") {
      string first_word, second_word;
      cin >> first_word >> second_word;
      if (AreSynonyms(synonyms, first_word, second_word)) {
        cout << "YES" << endl;
      } else {
        cout << "NO" << endl;
      }
    }
  }
  return 0;
}
```

Таким образом, весь наш проект представляет собой 7 файлов:

1. **coursera.cpp** – главный файл с `main()`, в котором лежит решение нашей задачи;
2. **synonyms.h** – объявления функций, решающих нашу задачу и **synonyms.cpp** – определения этих самых функций;
3. **test_runner.h** и **test_runner.cpp** – определения и объявления функций и классов, связанных с юнит-тестированием;
4. **tests.h** – объявления тестирующих функций и **test.cpp** – их определение.

Если внести изменения в test_runner.h, то у нас пересоберётся всё: test_runner.cpp, tests.cpp, coursera.cpp. Потому что coursera.cpp включает в себя test.h, который включает в себя test_runner.h. Таким образом:

1. Сборка проектов состоит из трёх стадий: препроцессинг, компиляция и компоновка;
2. При повторной сборке проекта компилируются только изменённые .cpp-файлы;

3. Внесение определений в .cpp-файлы позволяет при каждой сборке компилировать только изменённые файлы;

4. Это сильно ускоряет пересборку проекта.

Правило одного определения

Мы ранее говорили, что объявлений может быть сколько угодно, а определение обязательно должно быть ровно одно. И давайте мы ещё раз это продемонстрируем: вот у нас есть функция, например, `GetSynonymCount`, и у неё есть определение в файле synonyms.cpp. Если мы просто возьмём и скопируем это определение, а потом запустим компиляцию, то мы получим знакомую ошибку redefinition. Однако в больших проектах бывают ситуации, когда в вашем проекте, вроде бы, есть всего одно определение функции, но при этом компилятор сообщает вам, что у вас одна и та же функция определена несколько раз. И давайте посмотрим, как это выглядит и по какой причине случается. Давайте, например, возьмём нашу функцию `GetSynonymCount` и перенесём её определение обратно в заголовочный файл, как было у нас несколькими видео ранее. И скомпилируем наш проект. Давайте мы его соберём. И что-то пошло не так. Нам компилятор написал `first defined here`. А в консоли увидим `"multiple definition of GetSynonymsCount"`. Вроде определение функции одно, но ошибка возникает. Посмотрим на строки запуска компилятора:

```
21:09:19 **** Incremental Build of configuration Debug for proj
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\coursera.o" ".
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\tests.o" "..\\
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\\synonyms.o" ".
g++ -o coursera.exe "src\\coursera.o" "src\\synonyms.o" "src\\t
src\synonyms.o: In function `std::_Rb_tree<std::__cxx11::basic_
c:/dev/mingw-w64/mingw64/lib/gcc/x86_64-w64-mingw32/7.1.0/inclu
src\coursera.o:C:\workspace\coursera\Debug/../src/synonyms.h:18
src\tests.o: In function `std::_Rb_tree<std::__cxx11::basic_str
C:\workspace\coursera\Debug/../src/synonyms.h:18: multiple defi
src\coursera.o:C:\workspace\coursera\Debug/../src/synonyms.h:18
collect2.exe: error: ld returned 1 exit status

21:09:25 Build Finished (took 5s.944ms)
```

Рис. 2.8: Настройка компилятора

Каждый .cpp файл успешно скомпилировался. На этапе компоновки возникает ошибка.

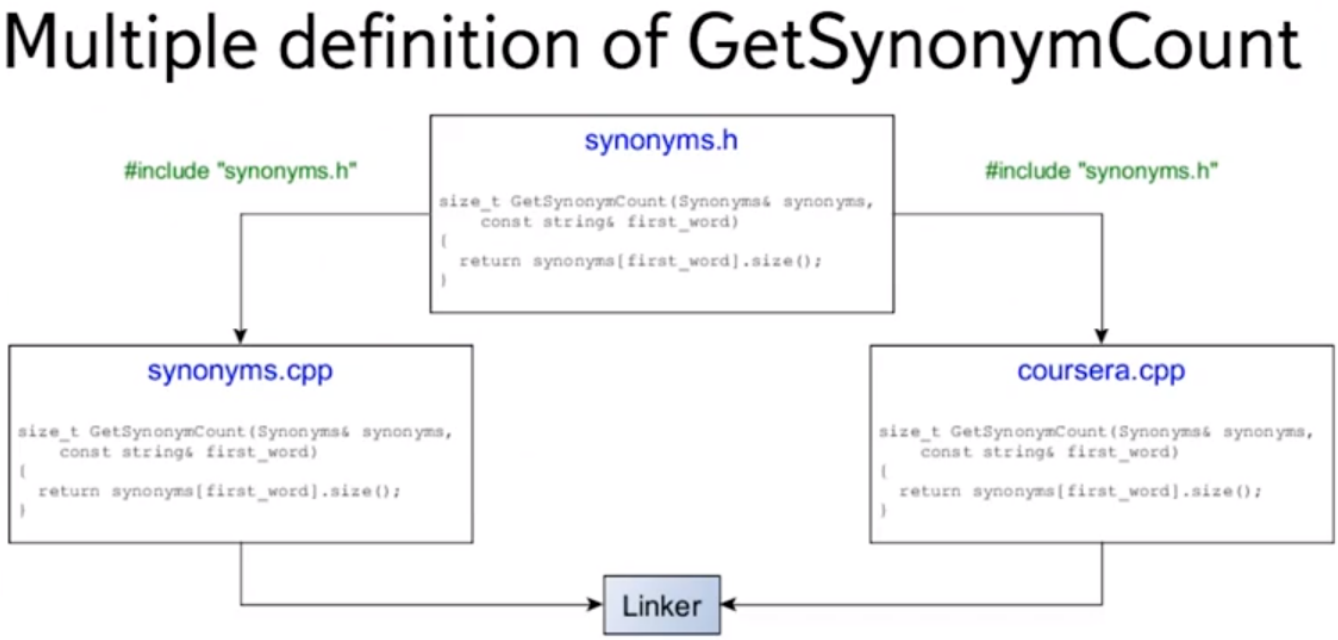


Рис. 2.9: Схема компиляции и сборки проекта

Ошибка происходит, когда компоновщик видит два определения одной и той же `GetSynonymCount` уже на этапе компоновки двух разных .cpp-файлов. Он видит, что одна и та же функция определена в двух объектных файлах и сообщает об ошибке. Вспомним, что основной причиной разделения на файлы было ускорение сборки. Теперь у нас есть ещё одна причина помещать определения в .cpp-файлы – это позволяет избежать ошибки Multiple definitions. Таким образом:

1. В C++ есть One Definition Rule (ODR);
2. Если функция определена в .h-файле, который включается в несколько .cpp-файлов, то нарушается ODR;
3. Чтобы не нарушать ODR, все определения надо помещать в .cpp-файлы.

Итоги

1. Разбиение программы на файлы упрощает её понимание и переиспользование кода, а также ускоряет перекомпиляцию;
2. В C++ есть два типа файлов: заголовочные (чаще .h) и файлы реализации .cpp;
3. Включение одного файла в другой осуществляется с помощью директивы `#include`;
4. Чтобы избежать двойного включения, надо добавлять `#pragma once`;
5. Знаем, что такое объявления и определения. Объявлений может быть сколько угодно, а определение только одно (ODR);
6. В .h-файлы обычно помещают объявления, а в .cpp – определения;
7. Если помещать определения в .h-файлы, то возможно нарушение ODR на этапе компоновки.