2.3. Контейнеры

2.3.1. Контейнер vector

Тип vector представляет собой набор элементов одного типа. Тип элементов вектора указывается в угловых скобках. Классический сценарий использования вектора — сохранение последовательности элементов.

Создание вектора требуемой длины. Ввод и вывод с консоли

Напишем программу, которая считывает из консоли последовательность строк, например, имен лекторов. Сначала на вход подается число элементов последовательности:

```
int n;
cin >> n;
```

Поскольку известно количество элементов последовательности, его можно указать в конструкторе вектора (то есть в круглых скобках после названия переменной):

```
vector<string> v(n);
```

После этого можно с помощью цикла for перебрать все элементы вектора по ссылке:

```
for (string& s : v) {
  cin >> s;
}
```

Каждый очередной элемент `s` — ссылка на очередной элемент вектора. С помощью этой ссылки считывается очередная строчка.

Теперь остается вывести вектор на экран, чтобы проверить, что все было считано правильно. Для этого удобно написать специальную функцию, которая выводит все значения вектора. Вызываем функцию следующим образом:

```
PrintVector(v);
```

А само определение функции PrintVector располагаем над функцией main:

```
void PrintVector(const vector<string>& v) {
  for (string s : v) {
    cout << s << endl;
  }
}
```

Запустим программу и проверим, что она работает:

```
> 2
> Anton
> Ilia
Anton
Ilia
```

Отлично: мы успешно считали элементы вектора и успешно их вывели.

Добавление элементов в вектор. Методы push_back и size

Можно реализовать эту программу несколько иначе с помощью цикла while. Также считаем число элементов вектора `n`, но создадим пустой вектор `v`.

```
int n;
cin >> n;
vector<string> v;
```

Создадим переменную `i`, в которой будет храниться индекс считываемой на данной итерации строчки.

```
int i = 0;
```

В цикле while считываем строку из консоли в локальную вспомогательную переменную `s`, которая добавляется к вектору с помощью метода push_back:

```
while (i < n) {
  string s;
  cin >> s;
  v.push_back(s);
  cout << "Current size = " << v.size() << endl;
  ++i;
}
```

В конце каждой итерации значение `i` увеличивается на 1. Чтобы продемонстрировать, что размер вектора меняется, на каждой итерации его текущий размер выводится на экран.

После завершения цикла чтения, как и в предыдущем примере, выводим значения вектора на экран с помощью функции PrintVector:

```
PrintVector(v);
```

```
> 2
> first
Current size = 1
> second
Current size = 2
first
second
```

Как и ожидалось, после ввода первой строки текущий размер стал равным 1, а после ввода второй — равным 2.

Вернемся к прошлой программе, в которой размер вектора задавался через конструктор в самом начале, и добавим туда также вывод размера на каждой итерации цикла:

```
int n;
cin >> n;
vector<string> v(n);
for (string& s: v) {
  cin >> s;
  cout << "Current size = " << v.size() << endl;
}
PrintVector(v);
```

Запустим программу и убедимся, что в таком случае размер вектора постоянен:

```
> 2
> first
Current size = 2
> second
Current size = 2
first
second
```

Так происходит, потому что в самом начале программы вектор создается сразу нужного размера.

Задание элементов вектора при его создании

Бывают случаи, когда содержимое вектора заранее известно. В этом случае указать заранее известные значения при создании вектора можно с помощью фигурных скобок. Например, числовой вектор, содержащий количество дней в каждом месяце (для краткости: в первых 5 месяцах), можно создать так:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};
```

Такой вектор можно распечатать:

```
PrintVector(days_in_months);
```

Правда, сперва следует подправить функцию PrintVector так, чтобы она принимала числовой вектор, а не вектор строк:

```
void PrintVector(const vector<int>& v) {
  for (auto s : v) {
    cout << s << endl;
  }
}
```

Запустим программу, убеждаемся, что она работает как надо.

Иногда бывает необходимым изменить значения вектора после его создания. Например, в високосных годах количество дней в феврале — 29, и чтобы это учесть, слегка допишем нашу программу:

```
vector<int> days_in_months = {31, 28, 31, 30, 31};
if (true) { // if year is leap
  days_in_months[1]++;
}
PrintVector(days_in_months);
```

Здесь для простоты проверка на високосность опущена. Замечу, что в C++ элементы вектора нумеруются с нуля, поэтому количество дней в феврале хранится в первом элементе вектора.

Из этого примера можно сделать вывод, что вектор также можно использовать для хранения элементов в привязке к их индексам.

Создание вектора, заполненного значением по умолчанию

Допустим, нужно создать вектор, который для каждого дня в феврале хранит, является ли данный день праздничным. В этом случае следует использовать вектор булевых значений. Поскольку большинство дней праздничными не являются, хотелось бы, чтобы при создании вектора все его значения по умолчанию были false.

Значение по умолчанию можно указать, передав его в качестве второго аргумента конструктора:

```
vector<bool> is_holiday(28, false);
```

В качестве первого аргумента конструктора указывается длина вектора, как и в первом примере. После этого заполним элементы вектора. Например, известно, что 23 февраля — праздничный день:

```
is_holiday[22] = true;
```

Вывести вектор в консоль можно с помощью функции PrintVector:

```
PrintVector(is_holiday);
```

Функцию PrintVector все же предстоит сперва доработать, чтобы она принимала вектор булевых значений.

```
void PrintVector(const vector<bool>& v) {
  for (auto s : v) {
    cout << s << endl;
  }
}
```

Заметим, что изменилось только определение типа при задании параметра функции, а ее тело осталось неизменным. В будущем это позволит обобщить эту функцию для вывода векторов разных типов. Но пока мы не обсудили этот вопрос, приходится довольствоваться только функциями, каждая из которых работает с векторами определенного типа.

Изменение длины вектора

Для удобства сперва доработаем функцию вывода, чтобы кроме значений выводились также и индексы элементов.

```
void PrintVector(const vector<bool>& v) {
  int i = 0;
  for (auto s : v) {
    cout << i << ": "<< s << endl;
    ++i;
  }
}
```

Иногда бывает необходимым изменить длину вектора. Например, если необходимо (по тем или иным причинам) созданный в предыдущей программе вектор использовать для хранения праздничных мартовских дней, его нужно сперва расширить и заполнить значением по умолчанию.

Попытаемся сделать это с помощью функции resize, которая может выполнить то, что надо. Попробуем это сделать:

```
is_holiday.resize(31);
PrintVector(is_holiday);
```

Метод resize сделал не то, что мы хотели, потому что старые значения остались и 23 марта оказалось праздничным. Если мы хотим переиспользовать этот вектор и сделать его нужной длины, нам понадобится метод assign:

```
is_holiday.assign(31, false);
```

В качестве первого аргумента передается желаемый размер вектора, а в качестве второго — какими элементами проинициализировать его элементы. Теперь можно указать, что 8 марта — праздничный день:

```
is_holiday[7] = true;
```

Запустив код, убеждаемся, что «упоминание о 23 марта» пропало, как и хотелось.

```
PrintVector(is_holiday);
```

Очистить вектор можно с помощью метода clear:

```
is_holiday.clear();
```

2.3.2. Контейнер map

Создание словаря. Добавление элементов

Допустим, требуется хранить важные события в привязке к годам, в которые они произошли. Для решения этой задачи лучше всего подходит такой контейнер как словарь. Словарь состоит из пар ключ-значение, причем ключи не могут повторяться. Для работы со словарями нужно подключить соответствующий заголовочный файл:

```
#include <map>
```

Создадим словарь с ключами типа int и строковыми значениями:

```
map<int, string> events;
events[1950] = "Bjarne Stroustrup's birth";
events[1941] = "Dennis Ritchie's birth";
events[1970] = "UNIX epoch start";
```

Напишем функцию, которая позволяет вывести словарь на экран:

```
void PrintMap(const map<int, string>& m) {
  cout << "Size = " << m.size() << endl;
  for (auto item: m) {
    cout << item.first << ": " << item.second << endl;
  }
}
```

Обратиться к ключу очередного элемента item при итерировании можно как item.first, а к значению — как к item.second. Также добавим в функцию вывода словаря вывод его размера (используя метод size).

Итерирование по элементам словаря

Выведем получившийся словарь на экран с помощью написанной функции:

```
PrintMap(events);
```

На экран будут выведены три элемента:

```
Size = 3
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
1970: UNIX epoch start
```

Словарь не просто вывелся на экран в формате ключ-значение. В выводе ключи оказались отсортированными в порядке возрастания целых чисел.

Этот пример демонстрирует одно из важных свойств словаря: элементы в нем хранятся отсортированными по ключам, а также выводятся отсортированными в цикле for.

Обращение по ключу к элементам словаря

Кроме того, можно обращаться к конкретным значениям из словаря по ключу. Например, можно узнать событие, которое произошло в 1950 году:

```
cout << events[1950] << endl;
```

```
Bjarne Stroustrup's birth
```

Отдельно отметим, что такой синтаксис очень напоминает синтаксис для получения значения элемента вектора по индексу. В некотором смысле, словарь позволил расширить функционал вектора: теперь в качестве ключей можно указывать сколь угодно большие целые числа.

Удаление по ключу элементов словаря

Элементы словаря можно не только добавлять в него, но и удалять. Для удаления элемента словаря по ключу используется метод erase:

```
events.erase(1970);
PrintMap(events);
```

```
Size = 2
1941: Dennis Ritchie's birth
1950: Bjarne Stroustrup's birth
```

Построение «обратного» словаря

Ключи словаря могут иметь тип string. Продемонстрируем это, обратив построенный нами словарь. Словарь, который получится в результате, позволит получать по названию события год, когда это событие произошло.

Для построения такого словаря, напишем функцию BuildReversedMap:

```
map<string, int> BuildReversedMap(
    const map<int, string>& m) {
  map<string, int> result;
  for (auto item: m) {
    result[item.second] = item.first;
  }
  return result;
}
```

Реализация этой функции достаточно проста. Сперва нужно приготовить итоговый словарь, типы ключей и значений в котором переставлены по сравнению с исходным словарем. Затем в цикле for нужно пробежаться по всем элементам исходного словаря и записать в итоговый, используя в качестве ключа бывшее значение, а в качестве значения — ключ. После цикла нужно вернуть получившийся словарь с помощью return.

Для вывода на экран получившегося словаря необходимо написать функцию PrintReversedMap, поскольку мы пока не научились писать функцию, выводящую на печать словарь любого типа:

```
void PrintReversedMap(const map<string, int>& m) {
  cout << "Size = " << m.size() << endl;
  for (auto item: m) {
    cout << item.first << ": " << item.second << endl;
  }
}
```

Еще раз отметим, что тело функции уже довольно общее и в нем нигде не содержатся типы ключей и значений.

Теперь можно запустить следующий код:

```
map<string, int> event_for_year = BuildReversedMap(events);
PrintReversedMap(event_for_year);
```

```
Size = 2
Bjarne Stroustrup's birth: 1950
Dennis Ritchie's birth: 1941
```

Также по названиям событий можно получить год, в котором они произошли:

```
cout << event_for_year["Bjarne Stroustrup's birth"];
```

```
1950
```

Создание словаря по заранее известным данным

Создание словаря по заранее известному набору пар ключ-значение можно произвести следующим образом с помощью фигурных скобок:

```
map<string, int> m = {{"one", 1}, {"two" ,2}, {"three", 3}};
```

Выведем словарь на экран и убедимся, что он создался правильно:

```
PrintMap(m);
```

```
one: 1
three: 3
two: 2
```

Все ключи здесь отсортировались лексикографически, то есть в алфавитном порядке.

Следует также отметить, что функцию печати словаря можно улучшить, итерируясь по нему по константной ссылке:

```
void PrintMap(const map<string, int>& m) {
  for (const auto& item: m) {
    cout << item.first << ": " << item.second << endl;
  }
}
```

В таком случае получается избежать лишнего копирования элементов словаря.

Еще раз отметим, как удалять значения из словаря, например для ключа «three»:

```
map<string, int> m = {{"one", 1}, {"two", 2}, {"three", 3}};
m.erase("three");
PrintMap(m);
```

```
one: 1
two: 2
```

Подсчет количества различных элементов последовательности

Словари могут быть полезными, если необходимо подсчитать, сколько раз встречаются элементы в некоторой последовательности.

Допустим, дана последовательность слов:

```
vector<string> words = {"one", "two", "one"};
```

Строки могут повторяться. Необходимо подсчитать, сколько раз встретилась каждое слово из этой последовательности. Для этого создадим словарь:

```
map<string, int> counters;
```

После этого пробежимся по всем элементам последовательности. Случай, когда слово еще не встречалось, нужно будет рассматривать отдельно, например так:

```
for (const string& word : words) {
  if (counters.count(word) == 0) {
    counters[word] = 1;
  } else {
    ++counters[word];
  }
}
```

Проверка на то, содержится ли элемент в словаре, может быть произведена с помощью метода count, как показано в коде. Такой код, безусловно, работает, но оказывается, что он избыточен. Достаточно написать так:

```
for (const string& word : words) {
  PrintMap(counters);
  ++counters[word];
}
PrintMap(counters);
```

Дело в том, что как только происходит обращение к конкретному элементу словаря с помощью квадратных скобок, компилятор уже создает пару для этого ключа со значением по умолчанию (для целого числа значение по умолчанию — 0).

Здесь мы сразу добавили вывод всего словаря для того, чтобы продемонстрировать как меняется размер словаря (в функцию PrintMap также добавлен вывод размера словаря):

```
Size = 0
Size = 1
one: 1
```

```
Size = 2
one: 1
two: 1
Size = 2
one: 2
two: 1
```

Продемонстрируем, что от простого обращения к элементу словаря происходит добавление к нему пары с этим ключом и значением по умолчанию:

```
map<string, int> counters;
counters["a"];
PrintMap(counters);
```

```
Size = 1
a: 0
```

Группировка слов по первой букве

Приведем еще один пример, показывающий, как можно использовать свойство изменения размера словаря при обращении к несуществующему ключу. Предположим, что необходимо сгруппировать слова из некоторой последовательности по первой букве. Решение данной задачи может выглядеть следующим образом:

```
vector<string> words = {"one", "two", "three"};
map<char, vector<string>> grouped_words;
for (const string& word : words) {
  grouped_words[word[0]].push_back(word);
}
```

В цикле for сначала идет обращение к несуществующему ключу (первой букве каждого слова). При этом ключ добавляется в словарь вместе с пустым вектором в качестве значения. Далее, с помощью метода push_back текущее слово присваивается в качестве значения текущего ключа. Выведем словарь на экран и убедимся, что слова были сгруппированы по первой букве:

```
for (const auto& item: grouped_words) {
  cout << item.first << endl;
  for (const string& word : item.second) {
    cout << word << " ";
  }
  cout << endl;
}
```

```
o
one
t
two three
```

Стандарт C++17

Недавно комитет по стандартизации языка C++ утвердил новый стандарт C++17. Говоря простым языком, были утверждены новые возможности языка. Но, к сожалению, изменения в стандарте только спустя некоторое время отражаются в свежих версиях компиляторов. Также свежие версии компиляторов не всегда просто использовать, так как они еще не появились в дистрибутивах для разработки. Тем не менее, имеет смысл рассказывать о свежих возможностях языка, даже если пока они не поддерживаются компиляторами и их еще нельзя использовать.

Чтобы попробовать новые возможности компиляторов, существуют различные ресурсы, например gcc.goldbolt.org. Он представляет собой окно ввода кода на C++ и панель для выбора версии компилятора. На данной панели можно выбрать еще не вышедшую версию компилятора gcc 7. Чтобы сказать компилятору, что код будет соответствовать новому стандарту, нужно указать флаг компиляции `|--std=c++17|`.

Среди новых возможностей — новый синтаксис для итерирования по словарю. Например, так бы выглядел код итерирования с использованием старого стандарта:

```
#include <map>

using namespace std;

int main() {
        map<string, int> m = {{"one", 1}, {"two", 2}};
        for (const auto& item : n) {
                item.first, item.second;
        }

        return 0;
}
```

В данном коде имеются следующие проблемы:

- Переменная item имеет «странный» тип с полями first и second.
- Нужно либо помнить, что first соответствует ключу, а second — значению текущего элемента, либо заводить временные переменные.

В новом стандарте появляется возможность писать такой код более понятно:

```
map<string, int> m = {{"one", 1}, {"two", 2}};
for (const auto& [key, value] : m) {
        key, value;
}
```

2.3.3. Контейнер set

Допустим, необходимо сохранить для каждого человека, является ли он известным. В этом случае можно было бы завести словарь, ключами в котором были бы строки, а значениями — логические значения:

```
map<string, bool> is_famous_person;
```

Теперь, чтобы указать, что какие-то люди являются известными, можно написать следующий код:

```
is_famous_person["Stroustrup"] = true;
is_famous_person["Ritchie"] = true;
```

Имеет ли смысл добавлять в этот словарь людей, которые являются неизвестными? Наверное, нет: таких людей слишком много и их нет нужды хранить, когда можно хранить только известных людей. А в этом случае значениями в таком словаре являются только true.

Создание множества. Добавление элементов.

Для решения такой задачи более естественно использовать другой контейнер — множество (set). Для работы с множествами необходимо подключить соответствующий заголовочный файл:

```
#include <set>
```

Теперь можно создать множество известных людей:

```
set<string> famous_persons;
```

Добавить в это множество элементы можно с помощью метода insert:

```
famous_persons.insert("Stroustrup");
famous_persons.insert("Ritchie");
```

Печать элементов множества

Функция PrintSet, позволяющая печатать на экране все элементы множества строк, реализуется следующим образом:

```
void PrintSet(const set<string>& s) {
  cout << "Size = " << s.size() << endl;
  for (auto x : s) {
    cout << x << endl;
  }
}
```

В эту функцию сразу добавлен вывод размера множества — он может быть получен с помощью метода size.

Теперь можно вывести на экран элементы множества известных людей:

```
PrintSet(famous_persons);
```

```
Size = 2
Ritchie
Stroustrup
```

Элементы множества выводятся в отсортированном порядке, а не в порядке добавления.

Также гарантируется уникальность элементов. То есть повторно никакой элемент не может быть добавлен в множество.

```
set<string> famous_persons;
famous_persons.insert("Stroustrup");
famous_persons.insert("Ritchie");
famous_persons.insert("Stroustrup");
PrintSet(famous_persons);
```

```
Size = 2
Ritchie
Stroustrup
```

Удаление элемента

Удаление из множества производится с помощью метода erase:

```
set<string> famous_persons;
famous_persons.insert("Stroustrup");
famous_persons.insert("Ritchie");
famous_persons.insert("Anton");
PrintSet(famous_persons);
```

```
Size = 3
Anton
Ritchie
Stroustrup
```

```
    famous_persons.erase("Anton");
    PrintSet(famous_persons);
```

```
Size = 2
Ritchie
Stroustrup
```

Создание множества с известными значениями

С помощью фигурных скобок можно создать множество, заранее указывая значения содержащихся в нем элементов. Например, множество названий месяцев может быть инициализировано как:

```
    set<string> month_names =
        {"January", "March", "February", "March"};
    PrintSet(month_names);
```

```
Size = 3
February
January
March
```

Сравнение множеств

Как и другие контейнеры, множества можно сравнивать:

```
    set<string> month_names =
        {"January", "March", "February", "March"};
    set<string> other_month_names =
        {"March", "January", "February"};

    cout << (month_names == other_month_names) << endl;
```

В результате будет выведено «1», то есть эти множества равны.

Проверка принадлежности элемента множеству

Для того, чтобы быстро проверить, принадлежит ли элемент множеству, можно использовать метод count:

```
    set<string> month_names =
        {"January", "March", "February", "March"};
    cout << month_names.count("January") << endl;
```

Создание множества по вектору

Чтобы создать множество по вектору, не обязательно писать цикл. Реализовать это можно следующим образом:

```
vector<string> v = {"a", "b", "a"};
set<string> s(begin(v), end(v));
PrintSet(s);
```

```
Size = 2
a
b
```

С помощью аналогичного синтаксиса можно создать и вектор по множеству.