



Основы разработки на C++: жёлтый пояс

Неделя 1

Целочисленные типы, кортежи, шаблонные функции



Оглавление

Целочисленные типы, кортежи, шаблонные функции	2
1.1 Целочисленные типы	2
1.1.1 Введение в целочисленные типы	2
1.1.2 Преобразования целочисленных типов	5
1.1.3 Безопасное использование целочисленных типов	7
1.2 Кортежи и пары	10
1.2.1 Упрощаем оператор сравнения	10
1.2.2 Кортежи и пары	12
1.2.3 Возврат нескольких значений из функций	15
1.3 Шаблоны функций	17
1.3.1 Введение в шаблоны	17
1.3.2 Универсальные функции вывода контейнеров в поток	19
1.3.3 Рефакторим код и улучшаем читаемость вывода	22
1.3.4 Указание шаблонного параметра-типа	24

Целочисленные типы, кортежи, шаблонные функции

1.1. Целочисленные типы

1.1.1. Введение в целочисленные типы

Вы уже знаете один целочисленный тип – это тип `int`. Начнём с проблемы, которая может возникнуть при работе с ним. Для этого вспомним задачу «Средняя температура» из первого курса. Нам был дан набор наблюдений за температурой, в виде вектора `t` (значения 8, 7 и 3). Нужно было найти среднее арифметическое значение температуры за все дни и затем вывести номера дней, в которые значение температуры было больше, чем среднее арифметическое. Должно получиться $(8 + 7 + 3)/3 = 6$.

```
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> t = {8, 7, 3}; // вектор с наблюдениями
    int sum = 0; // переменная с суммой
    for (int x : t) { // проитерировались по вектору и нашли суммарную температуру
        sum += x;
    }
    int avg = sum / t.size(); // получили среднюю температуру
    cout << avg << endl;
    return 0;
} // вывод программы будем писать последним комментарием листинга
// 6
```

Но в той задаче было ограничение: вам гарантировалось, что все значения температуры не

отрицательные. Если в таком решении у вас в исходном векторе будут отрицательные значения температуры, например, -8 , -7 и 3 (ответ $(-8 - 7 + 3)/3 = -4$), код работать не будет:

```
int main () {
    vector<int> t = {-8, -7, 3};    // сумму -4
    ...
    int avg = sum / t.size();    // t.size() не умеет хранить отрицательные числа
    cout << avg << endl;
    return 0;
}
// 1431655761
```

Это не -4 . На самом деле мы от незнания неаккуратно использовали другой целочисленный тип языка C++. Он возникает в `t.size()` – это специальный тип, который не умеет хранить отрицательные числа. Размер контейнера отрицательным быть не может, и это беззнаковый тип. Какая еще бывает проблема с целочисленными типами? Очень простой пример:

```
int main () {
    int x = 2'000'000'000;    // для читаемости разбиваем на разряды кавычками
    cout << x << " ";    // выводим само число
    x = x * 2;
    cout << x << " ";    // выводим число, умноженное на 2
    return 0;
}
// 2000000000 -294967296
```

Итак, запускаем код и видим, что 4 миллиарда в переменную типа `int` не поместилось.

Особенности целочисленных типов языка C++:

1. В языке C++ память для целочисленных типов ограничена. Если вам не нужны целые числа размером больше 2 миллиардов, язык C++ для вас выделит вот ровно столько памяти, сколько достаточно для хранения числа размером 2 миллиарда. Соответственно, у целочисленных типов языка C++ ограниченный диапазон значений.
2. Некоторые целочисленные типы языка C++ беззнаковые. Используя их, вы сможете хранить больше положительных значений, но не сможете хранить отрицательные.

Виды целочисленных типов:

- `int` – стандартный целочисленный тип.
 1. `auto x = 1;` – как и любая комбинация цифр имеет тип `int`;
 2. Эффективен: операции с ним напрямую транслировались в инструкции процессора;
 3. В зависимости от архитектуры имеет размер 64 или 32 бита, и диапазон его значений от -2^{31} до $(2^{31} - 1)$.
- `unsigned int (unsigned)` – беззнаковый аналог `int`.
 1. Диапазон его значений от 0 до $(2^{32} - 1)$. Занимает 8 или 4 байта.
- `size_t` – тип для представления размеров.
 1. Результат вызова `size()` для контейнера;
 2. 4 байта (до $(2^{32} - 1)$) или 8 байт (до $(2^{64} - 1)$). Зависит от разрядности системы.
- Типы с известным размером из модуля `cstdint`.
 1. `int32_t` – знаковый, всегда 32 бита (от -2^{31} до $(2^{31} - 1)$);
 2. `uint32_t` – беззнаковый, всегда 32 бита (от 0 до $(2^{32} - 1)$);
 3. `int8_t` и `uint8_t` всегда 8 бит; `int16_t` и `uint16_t` всегда 16 бит; `int64_t` и `uint64_t` всегда 64 бита.

Тип	Размер	Минимум	Максимум	Стоит ли выбрать его?
<code>int</code>	4 (обычно)	-2^{31}	$2^{31} - 1$	по умолчанию
<code>unsigned int</code>	4 (обычно)	0	$2^{32} - 1$	только положительные
<code>size_t</code>	4 или 8	0	$2^{32} - 1$ или $2^{64} - 1$	размер контейнеров
<code>int8_t</code>	1	-2^7	$2^7 - 1$	сильно экономить память
<code>int16_t</code>	2	-2^{15}	$2^{15} - 1$	экономить память
<code>int32_t</code>	4	-2^{31}	$2^{31} - 1$	нужно ровно 32 бита
<code>int64_t</code>	8	-2^{63}	$2^{63} - 1$	недостаточно <code>int</code>

Узнаём размеры и ограничения типов:

Как узнать размеры типа? Очень просто:

```
cout << sizeof(int16_t) << " "; // размер типа в байтах. Вызывается от переменной
cout << sizeof(int) << endl;
// 2 4
```

Узнаём ограничения типов:

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main() {
    cout << sizeof(int16_t) << " ";
    cout << numeric_limits<int>::min() << " " << numeric_limits<int>::max() << endl;
    return 0;
}
// 4 -2147483648 2147483647
```

1.1.2. Преобразования целочисленных типов

Начнём с эксперимента. Прибавим 1 к максимальному значению типа `int`.

```
#include <iostream>
#include <vector>
#include <limits> // подключаем для получения информации о типе
using namespace std;
int main() {
    cout << numeric_limits<int>::max() + 1 << " ";
    cout << numeric_limits<int>::min() - 1 << endl;
    return 0;
}
// -2147483648 2147483647
```

Получилось, что $\text{max} + 1 = \text{min}$, а $\text{min} - 1 = \text{max}$. Теперь попробуем вычислить среднее арифметическое $1000000000 + 2000000000$.

```
int x = 2'000'000000;
int y = 1'000'000000;
cout << (x + y) / 2 << endl;
// -647483648
```

Хоть их среднее вмещается в `int`, но программа сначала сложила `x` и `y` и получила число, не уместившееся в тип `int`, и только после этого поделила его на 2. Если в процессе случается

переполнение, то и с результатом будет не то, что мы ожидаем. Теперь попробуем поработать с беззнаковыми типами:

```
int x = 2'000'000000;
unsigned int y = x; // сохраняем в переменную беззнакового типа 2000000000
unsigned int z = -z;
cout << x << " " << y << " " << -x << " " << z << endl;
// 2000000000 2000000000 -2000000000 2294967296
```

Если значение уместится даже в `int`, то проблем не будет. Но если записать отрицательное число в `unsigned`, мы получим не то, что ожидали. Возвращаясь к задаче о средней температуре, посмотрим, в чём была проблема:

```
vector<int> t = {-8, -7, 3};
int sum = 0; // знаковое
for (int x : t){
    sum += x;
}
int avg = sum / t.size(); // sum / t.size(); уже беззнаковое, т.к. t.size() беззнаковое
cout << avg << endl;
```

Правила вывода общего типа:

1. Перед сравнениями и арифметическими операциями числа приводятся к общему типу;
2. Все типы размера меньше `int` приводятся к `int`;
3. Из двух типов выбирается больший по размеру;
4. Если размер одинаковый, выбирается беззнаковый.

Примеры:

Слева	Операция	Справа	Общий тип	Комментарий
<code>int</code>	<code>/</code>	<code>size_t</code>	<code>size_t</code>	большой размер
<code>int32_t</code>	<code>+</code>	<code>int8_t</code>	<code>int32_t (int)</code>	тоже больший размер
<code>int8_t</code>	<code>*</code>	<code>uint8_t</code>	<code>int</code>	все меньшие приводятся к <code>int</code>
<code>int32_t</code>	<code><</code>	<code>uint32_t</code>	<code>uint32_t</code>	знаковый к беззнаковому

Для определения типа в самой программе можно просто вызвать ошибку компиляции и посмотреть лог ошибки. Изменим одну строчку:

```
int avg = (sum / t.size()) + vector<int>{}; // прибавили пустой вектор
```

И получим ошибку, в логе которой указано, что наша переменная `avg` имеет тип `int`. Теперь попробуем сравнить знаковое и беззнаковое число:

```
int main () {
    int x = -1;
    unsigned y = 1;
    cout << (x < y) << " ";
    cout << (-1 < 1u) << endl; // суффикс u делает 1 типом unsigned по умолчанию
    return 0;
}
// 0 0
```

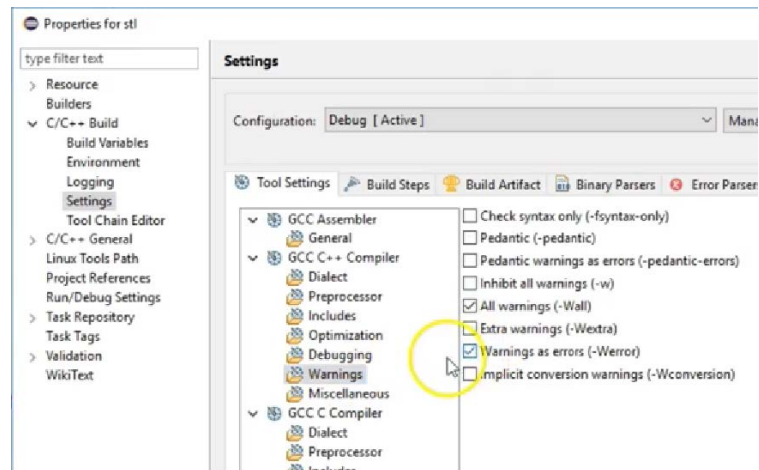
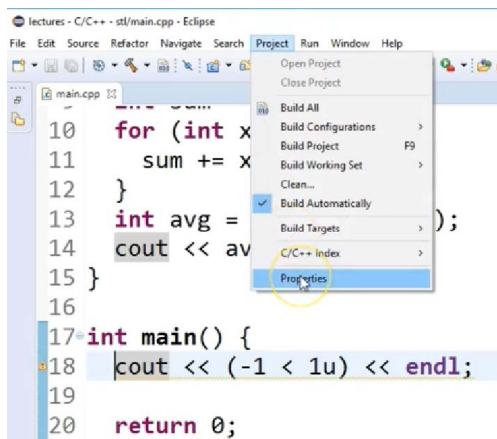
Как было сказано ранее, при операции между знаковым и беззнаковым типом обе переменные приводятся к беззнаковому. `-1`, приведённая к беззнаковому, становится очень большим числом, большим `1`. Суффикс `u` также приводит `1` к `unsigned`, а операция `<` теперь сравнивает `unsigned` `-1` и `1`. Причём в данном случае компилятор предупреждает нас о, возможно, неправильном сравнении.

1.1.3. Безопасное использование целочисленных типов

Настроим компилятор:

Попросим компилятор считать каждый warning (предупреждение) ошибкой. Project → Properties → C/C++ → Build → Settings → GCC C++ Compiler → Warnings и отмечаем Warnings as errors.

После этого ещё раз компилируем код. И теперь каждое предупреждение считается ошибкой, которую надо исправить. Это одно из правил хорошего кода.



```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < x.size(); ++i) {
        cout << i << " " << x[i] << endl;
    }
    return 0;
}
// error: "i < x.size()"... comparison between signed and unsigned
```

Есть два способа это исправить: объявить `i` типом `size_t` или явно привести `x.size()` к типу `int` с помощью `static_cast<int>(x.size())`.

```
int main () {
    vector<int> x = {4, 5};
    for (int i = 0; i < static_cast<int>(x.size()); ++i) {
        cout << i << " " << x[i] << " ";
    }
    return 0;
}
// 0 4 1 5
```

Исправляем задачу о температуре:

В задаче о температуре тоже приводим `t.size()` к знаковому с помощью оператора `static_cast`:

```
int main () {
    vector<int> t = {-8, -7, 3};    // сумма -4
```

```
...
int avg = sum / static_cast<int>(t.size()); // явно привели типы
cout << avg << endl;
return 0;
}
// -4
```

Предупреждений и ошибок не было. Всё, задача средней температуры для положительных и отрицательных значений решена! Таким образом если у нас где-то могут быть проблемы с беззнаковыми типами, мы либо следуем семантике и помним про опасности, либо приводим всё к знаковым с помощью `static_cast`.

Ещё примеры опасностей с беззнаковыми типами:

Переберём в векторе все элементы кроме последнего:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i < v.size() - 1; ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

После запуска код падает. Вычтя из `v.size()` 1 мы получили максимальное значение типа `size_t` и вышли из своей памяти. Чтобы такого не произошло, мы перенесём единицу в другую часть сложения:

```
int main() {
    vector<int> v; // пустой вектор
    for (size_t i = 0; i + 1 < v.size(); ++i) { // v.size() - беззнаковый 0
        cout << v[i] << endl;
    }
    return 0;
}
```

Теперь на пустом векторе у нас все компилируется и вывод пустой. А на непустом выводит все элементы, кроме последнего. Напишем программу вывода элементов вектора в обратном порядке:

```
int main() {
```

```
vector<int> v = {1, 4, 5};
for (size_t i = v.size() - 1; i >= 0; --i) {
    cout << v[i] << endl;
}
return 0;
}
```

На пустом векторе, очевидно, будет ошибка. Но даже на не пустом он сначала выводит 5, 4, 1, а затем очень много чисел и программа падает. Это произошло из-за того, что `i >= 0` выполняется всегда и мы входим в бесконечный цикл. От этой проблемы мы избавимся «заменой переменной» для итерации:

```
for (size_t k = v.size(); k > 0 ; --k) {
    size_t i = k - 1; // теперь
    cout << v[i] << endl;
}
```

Теперь всё работает нормально. В итоге, проблем с беззнаковыми типами помогают избежать:

- Предупреждения компилятора;
- Внимательность при вычитании из беззнаковых;
- Приведение к знаковым типам с помощью `static_cast`.

1.2. Кортежи и пары

1.2.1. Упрощаем оператор сравнения

Поговорим про новые типы данных – пары и кортежи. Начнём с проблемы, которая возникла в курсовом проекте курса «Белый пояс по C++»: мы должны были хранить даты в виде структур из полей «год», «месяц», «день» в ключах словарей. А поскольку словарь хранит ключи отсортированными, нам надо определить для этого типа данных оператор «меньше». Можно сделать это так:

```
#include <iostream>
#include <vector>
```

```
using namespace std;

struct Date {
    int year;
    int month;
    int day;
};

bool operator < (const Date& lhs, const Date& rhs) {
    if (lhs.year != rhs.year) {
        return lhs.year < rhs.year;
    }
    if (lhs.month != rhs.month) {
        return lhs.month < rhs.month;
    }
    return lhs.day < rhs.day;
}

...
```

Сначала сравниваем года, потом месяцы и затем дни. Но здесь много мест для ошибок и код довольно длинный. В эталонном решении курсового проекта эта проблема решена так:

```
bool operator < (const Date& lhs, const Date& rhs) {
    return vector<int>{lhs.year, lhs.month, lhs.day} <
        vector<int>{rhs.year, rhs.month, rhs.day};
}

...
```

Выигрыш в том, что для векторов лексикографический оператор сравнения уже определён и этот код работает для каких-нибудь двух дат:

```
int main() {
    cout << (Date{2017, 6, 8} < Date {2017, 1, 26}) << endl;
    return 0;
}

// 0
```

Действительно, первая дата больше второй и выводит 0. Но тип `vector` слишком мощный: он позволяет делать `push_back` в себя, удалять из середины, что-то ещё. Нам этот тип не нужен в данном случае. Нам нужно всего лишь объединить для левой даты и для правой даты три

значения в одно и сравнить. Кроме того, вектор работает только при однотипных элементах. Если бы у нас месяц был строкой, вектор бы не подходил.

Как же объединить разные типы в один массив? Для этого нужно подключить библиотеку `tuple`. И вместо вектора вызывать функцию `tie` от тех значений, которые надо связать. В нашем случае год, месяц и день левой даты и год, месяц и день правой даты надо связать и сравнить.

```
#include <tuple>
...
bool operator <(const Date& lhs, const Date&rhs) {
    auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // сохраним левую дату
    auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // и правую
    return lhs_key < rhs_key
}
int main() {
    cout << (Date{2017, "June", 8} < Date{2017, "January", 26}) << endl;
    return 0;
}
// 0
```

И действительно, строка «June» лексикографически больше строки «January» и наша программа делает то, что нужно. Теперь узнаем, какого типа у нас `lhs_key` и `rhs_key`, породив ошибку компиляции.

```
...
auto lhs_key = tie(lhs.year, lhs.month, lhs.day); // левая
auto rhs_key = tie(rhs.year, rhs.month, rhs.day); // правая
lhs_key + rhs_key; // тут нарочно порождаем ошибку
return lhs_key < rhs_key
...
// operator+ не определен для std::tuple<const int&, const std::string&, const int&>...
```

То есть они имеют тип `tuple<const int&, const string&, const int&>`. Tuple – это *кортеж*, т. е. структура из ссылок на нужные нам данные (возможно, разнотипные).

1.2.2. Кортежи и пары

Создадим кортеж из трёх элементов:

```
#include <iostream>
#include <vector>
#include <tuple>
using namespace std;
int main() {
    tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
    auto t = tie(7, "C++", true); // пытаемся связать константные значения в tie
    return 0;
}
```

В первой строке всё хорошо – мы создали структуру из трёх полей. А во второй – ошибка компиляции, потому что `tie` создает кортеж из ссылок на объекты (которые хранятся в каких-то переменных). Используем функцию `make_tuple`, создающую кортеж из самих значений. И будем обращаться к полям кортежа:

```
...
// tuple<int, string, bool> t(7, "C++", true); // просто создаем кортеж
auto t = make_tuple(7, "C++", true)
cout << get<1>(t) << endl;
return 0;
}
// C++
```

С помощью функции `get<1>(t)` мы получили 1-ый (нумерация с 0) элемент кортежа `t`. Использование кортежа `tuple` целесообразно, только если нам необходимо, чтобы в кортеже были разные типы данных.

Замечание: в C++ 17 разрешается не указывать шаблонные параметры `tuple` в `< ... >`, т. е. кортеж можно создавать так:

```
tuple t(7, "C++", true);
```

Но при компиляции компилятор попросит параметры. Оказывается надо явно сказать ему, чтобы он использовал стандарт C++ 17. Для этого снова идём в Project → Properties → C/C++ Build → Dialect → Other dialect flags и пишем `std = c++17`, т. е. версии C++ 17 (для этого компилятор должен быть обновлен до версии GCC7 или больше).

Если же мы хотим связать только два элемента, то используем структуру «пара» – `pair`. Пара – это частный случай кортежа, но отличие пары в том, что её поля называются `first` и `second` и к ним можно обращаться напрямую:

```
int main() {
    pair p(7, "C++"); // в новом стандарте можно и без <int, string>
    // auto p = make_pair(7, "C++"); // второй вариант
    cout << p.first << " " << p.second << endl;
    return 0;
}
// 7 C++
```

В результате нам вывело нашу пару. Эту структуру мы уже видели при итерировании по словарию. Так что нам удобно её использовать, не объявляя структуру явно.

Для примера создадим словарь (map):

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}};
    for (const auto& item : digits) {
        cout << item.first << " " << item.second << endl;
    }
    return 0;
}
// 1 one
```

Поскольку словарь – это пара «ключ-значение», то можно (как было объяснено в предыдущем курсе) распаковать значения item:

```
...
#include <map>
int main() {
    map<int, string> digits = {{1, "one"}}
    for (const auto& [key, value] : digits) {
        cout << key << " " << value << endl;
    }
    return 0;
}
// 1 one
```

Всё скомпилировалось, значит, мы можем итерироваться по словарию, не создавая пар.

1.2.3. Возврат нескольких значений из функций

Возврат нескольких значений из функций – ещё одна область применения кортежей и пар. Будем хранить класс с информацией о городах и странах:

```
#include <iostream>
#include <vector>
#include <utility>
#include <map>
#include <set>
using namespace std;

class Cities { // класс городов и стран
public:
    tuple<bool, string> FindCountry(const string& city) const {
        if (city_to_country.count(city) == 1) {
            return {true, city_to_country.at(city)};
            // city_to_country[city] выдало бы ошибку, потому что могло нарушить const словаря
        } else if (ambiguous_cities.count(city) == 1) {
            return {false, "Ambigious"};
        } else {
            return {false, "Not exists"}; // если нет
        }
        // выводит значение, нашлась ли единственная страна для города
    } // и сообщение - либо страна не нашлась, либо их несколько
private:
    // по названию города храним название страны:
    map<string, string> city_to_country;
    // множество городов, принадлежащих нескольким странам:
    set<string> ambiguous_cities;
}

int main() {
    Cities cities;
    bool success;
    string message; // свяжем кортежем ссылок
    tie(success, message) = cities.FindCountry("Volgograd");
    cout << " " << message << endl; // вывели результат
    return 0;
}
//0 Not exists
```


Всё работает. Таким образом, мы научились возвращать из метода несколько значений с помощью кортежа. А по новому стандарту можно получить кортеж и распаковать его в пару переменных:

```
int main() {
    Cities cities;
    auto [success, message] = cities.FindCountry("Volgograd"); // сразу распаковали
    cout << success << " " << message << endl;
    return 0;
}
//0 Not exists
```

Итак, если вы хотите вернуть несколько значений из функции или из метода, используйте кортеж. А если вы хотите сохранить этот кортеж в какой-то набор переменных, используйте `structured bindings` или функцию `tie`.

Кортежи и пары нужно использовать аккуратно. Они часто мешают читаемости кода, если вы начинаете обращаться к их полям. Например, представьте себе, что вы хотите сохранить словарь из названия города в его географические координаты. У вас будет словарь, у которого в значениях будут пары двух вещественных чисел. Назовем его `cities`. Как же потом вы будете, например, итерироваться по этому словарю?

```
int main() {
    map<string, pair<double, double>> cities;
    for (const auto& item : cities) {
        cout << item.second.first << endl; // абсолютно нечитаемый код
    }
    return 0;
}
```

Заключение: кортежи позволяют упростить написание оператора `<` или вернуть несколько значений из функции. Пары – это частный случай кортежей, у которых понятные названия полей, к которым удобно обращаться.

1.3. Шаблоны функций

1.3.1. Введение в шаблоны

Рассмотрим шаблоны функций на примере функции возведения числа в квадрат.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
using namespace std;

int Sqr(int x) { // функция возведения в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2) << endl; // результат выведем в поток
    return 0;
}
// 4
```

Функция работает. Теперь мы хотим возводить в квадрат дробные числа, например, 2.5. Получается снова 4. Это неправильно.

```
... cout << Sqr(2.5) << endl; // результат функции выведем в поток
// 4
```

Это происходит потому, что у нас нет аналогичной функции для работы с дробными числами. Заведём её:

```
int Sqr(int x) { // функция возведения целого числа в квадрат
    return x * x;
}

double Sqr(double x) { // функция возведения дробного числа в квадрат
    return x * x;
}

int main() {
    cout << Sqr(2.5) << endl;
    return 0;
}
```

```
}
// 6.25
```

Всё работает, но у нас появилось две функции, которые делают одно и то же, но с разными типами. Гораздо удобнее написать функцию, работающую с каким-то типом T:

```
using namespace std;
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    cout << Sqr(2.5) << " " << Sqr(3) << endl;
    return 0;
}
// 6.25 9
```

Собираем наш код и видим, что всё работает. Тип T компилятор выведет сам, чтобы типы поддерживали умножение. Нужно было только его объявить ключевым словом `template <typename T>`. Теперь попробуем возвести в квадрат пару:

```
#include <utility> // добавляем нужную библиотеку
... // код функции оставляем таким же
int main() {
    auto p = make_pair(2, 3); // создаем пару
    cout << Sqr(p) << endl; // пытаемся возвести ее в квадрат
    return 0;
}
// no match for 'operator*' (operand types are std::pair<int,int> and std::pair<int,int>)
```

Видим ошибку, т. к. для оператора умножения не определены аргументы «пара и пара». Тогда напишем шаблонный оператор умножения для пар:

```
using namespace std;
template <typename First, typename Second>
// т.к. умножение для пар не определено, вручную определим оператор умножения для пар:
pair<First, Second> operator * (const pair<First, Second>& p1,
                                const pair<First, Second>& p2) {
    // мы можем создавать переменные шаблонного типа
    First f = p1.first * p2.first;
```

```
Second s = p1.second * p2.second;
return {f, s};
}
template <typename T> // ключевое слово для объявления типа T
T Sqr(T x) {
    return x * x; // нам нужно, чтобы элемент x поддерживал операцию умножения
}

int main() {
    auto p = make_pair(2.5, 3); // создаем пару
    auto res = Sqr(p); // возводим пару в квадрат
    cout << res.first << " " << res.second << endl; // выводим получившееся
    return 0;
}
// 6.25 9
```

Код работает – пара возвелась в квадрат (и её дробная часть, и целая). Одним из важных плюсов языка C++ является возможность подобным образом избавляться от дублирований и сильно сокращать код.

1.3.2. Универсальные функции вывода контейнеров в поток

В курсах ранее мы часто печатали содержимое наших контейнеров, будь то `vector` или `map`, на экран. Для этого мы определяли специальную функцию либо перегружали оператор вывода в поток. Давайте это сделаем и сейчас:

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
// напишем свой оператор вывода в поток вектора целых типов
ostream& operator<< (ostream& out, const vector<int>& vi) {
    for (const auto& i : vi) { // проитерируем по вектору
        out << i << ' '; // выведем все элементы в поток
    }
    return out;
}
```

```
}

int main() {
    vector<int> vi = {1, 2, 3};
    cout << vi << endl;
}
// 1 2 3
```

Всё выводится. Но если поменять тип вектора на `double`, то будет ошибка:

```
...
vector<double> vi = {1, 2, 3}; // теперь дробный вектор
...
// no match for 'operator <<' (operand types are std::ostream and std::vector<double>)
```

Для решения этой проблемы можно было бы каждый раз заново дублировать код. Но с помощью шаблонов функций мы меняем тип вектора с `int` на шаблонный `T`:

```
using namespace std;
template <typename T> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const vector<T>& vi) { // вектор на шаблон
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}

int main() {
    vector<double> vi = {1.4, 2, 3}; // дробные числа
    cout << vi << endl;
}
// 1.4 2 3
```

Ошибки пропали, вывелся наш вектор из дробных чисел. Мы научились универсальным способом решать задачу для вектора. Таким же универсальным способом научимся решать задачу для других контейнеров.

```
int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <' (operand types are std::ostream and std::map<int, int>)
```

Видим, что оператор вывода для `map` не определён. Определим его так же, как и для вектора:

```
...
template <typename Key, typename Value> // объявили шаблонный тип T
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
...
int main() {
    map<int, int> m = {{1, 2}, {3, 4}};
    cout << m << endl;
}
// no match for 'operator <<' (operand types are std::ostream and std::pair<const int, int>)
```

Заметим, что ошибка для `map` имеет интересный вид: `pair<const int, int>`. Действительно, ведь `map` — это `pair`, в которой `key` нельзя модифицировать, а `value` можно. Получается, нам достаточно определить оператор вывода в поток для пары. Тогда мы сможем вывести и `map`.

```
// весь рабочий код
...
template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    out << p.first << ", " << p.second;
    return out;
}
template <typename T> // для vector
ostream& operator<< (ostream& out, const vector<T>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
template <typename Key, typename Value> // для пар
ostream& operator<< (ostream& out, const map<Key, Value>& vi) {
    for (const auto& i : vi) {
        out << i << ' ';
    }
    return out;
}
```

```
int main() {
    // vector<double> vi = 1.4, 2,3;
    // cout << vi << endl;
    map<int, int> m = {{1, 2}, {3, 4}}; // целые числа
    map<int, int> m2 = {{1.4, 2.1}, {3.4, 4}}; // дробные числа
    cout << m << ' ' << m2 << endl;
}
//1, 2 3, 4; 1.4, 2.1 3.4, 4
```

Код отработал корректно. Оба `map`'а вывелись, как нам и было нужно. Заметим, что код с вектором, если его добавить, до сих пор будет работать. Но этот код тоже можно доработать. Шаблоны для вектора и для `map`'а выглядят почти одинаково. Кроме того, для читаемости можно сделать улучшение: когда мы выводим `map`, обрамлять его в фигурные скобки, когда вектор – в квадратные, а когда мы выводим пару, обрамлять её круглыми скобками.

1.3.3. Рефакторим код и улучшаем читаемость вывода

Исправим предыдущую программу и сделаем её вывод более читаемым. Нужно создать шаблонную функцию, которая на вход будет принимать коллекцию. На вход этой функции будем передавать разделитель, через который надо вывести элементы нашей коллекции. Единственное, что мы требуем от входной коллекции: по ней можно итерироваться с помощью цикла `range-based for` и её элементы можно выводить в поток.

```
#include <iostream>
#include <vector>
#include <map>
#include <sstream>
#include <utility>
using namespace std;
template <typename Collection> // тип коллекции
string Join(const Collection& c, char d) { // передаем коллекцию и разделитель
    stringstream ss; // завели строковый поток
    bool first = true; // первый ли это элемент?
    for (const auto& i : c) {
        if (!first) {
            ss << d; // если вывели не первый элемент - кладем поток в разделитель
        }
    }
}
```

```

    first = false; // т.к. следующий элемент точно не будет первым
    ss << i; // кладем следующий элемент в поток
}
return ss.str();
}
template <typename First, typename Second> // для pair
ostream& operator<< (ostream& out, const pair<First, Second>& p) {
    return out << '(' <<p.first << ',' << p.second << ')'; // тоже изменили
}
template <typename T> // для vector изменили код и добавили скобочки
ostream& operator<< (ostream& out, const vector<T>& vi) {
    return out << '[' << Join(vi, ',') << ']';
} // оператор вывода возвращает ссылку на поток
template <typename Key, typename Value> // для map убрали аналогично vector
ostream& operator<< (ostream& out, const map<Key, Value>& m) {
    return out << '{' << Join(m, ',') << '}'; // и добавили фигурные скобочки
}

int main() {
    vector<double> vi = {1.4, 2, 3};
    pair<int, int> m1 = {1, 2};
    map<double, double> m2 = {{1.4, 2.1} , {3.4, 4}};
    cout << vi << ' ' << m1 << ' ' << m2 << endl;
}
// [1.4, 2,3] (1, 2) {(1.4, 2.1), (3.4, 4)}

```

Всё работает. Наша программа вывела сначала вектор, потом пару и затем `map`. Для более сложных конструкций она тоже будет работать. Например, вектор векторов:

```

int main() {
    vector<vector<int>> vi = {{1, 2}, {3, 4}};
    cout << vi << endl;
}
// [[1, 2], [3, 4]]

```

В итоге всё работает и на сложных контейнерах. Таким образом, мы сильно упростили наш код и избежали ненужного дублирования с помощью шаблонов и универсальных функций.

1.3.4. Указание шаблонного параметра-типа

Рассмотрим случай, когда компилятор не знает, как на основе вызова шаблонной функции вывести тип `T` на примере задачи о выводе максимального из двух чисел.

```
#include <iostream>
using namespace std;
template <typename T>
T max(T a, T b) {
    if (b < a) {
        return a;
    }
    return b;
}

int main() {
    cout << Max(2, 3) << endl;
    return 0;
}
// 3
```

Если оба числа целые, то всё работает. Но если поменять одно число на вещественное, мы увидим ошибки:

```
...
cout << Max(2, 3.5) << endl;
// deduce conflicting types for parameter 'T' (int and double)
```

Т. е. вывод шаблонного параметра типа `T` не может состояться, потому что компилятор не знает, что поставить: `int` или `double`. В таких ситуациях мы либо приводим переменные к одному типу, либо подсказываем компилятору таким образом:

```
cout << Max<double>(2, 3.5) << endl; // явно показываем компилятору тип T
// 3.5
```

А если же мы попросим `int`, получим следующее:

```
cout << Max<int>(2, 3.5) << endl; // 3.5 приведётся к int и мы сравнили
// 3
```

Писать уже существующие функции плохо, поэтому вызовем стандартную функцию `max` из библиотеки `algorithms`:

```
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    cout << max<int>(2, 3.5) << ' ' << max<double>(2, 3.5) << endl;
    return 0;
}
// 3 3.5
```

Функция `max` тоже шаблонная. Если явно указать тип, к которому приводить результат, у нас будет то же, что мы уже видели. Если же тип не указывать, произойдёт знакомая ошибка компиляции.

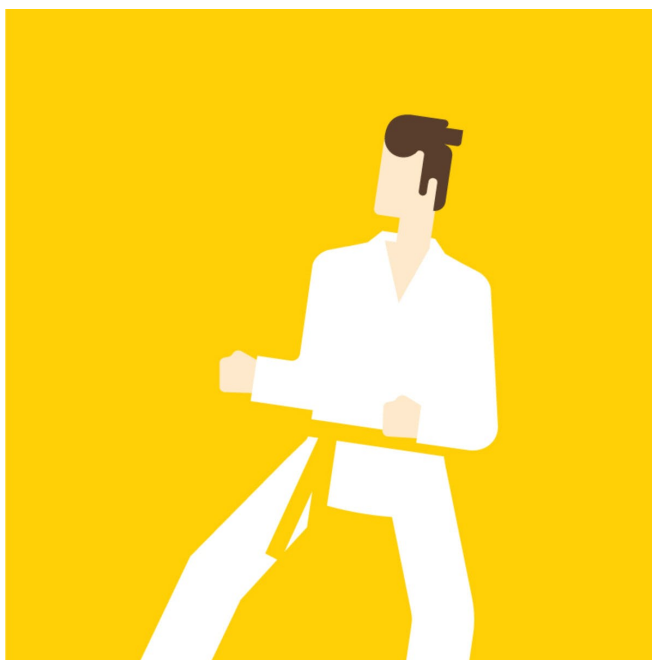
Подведём итоги:

1. Шаблонные функции объявляются так: `template <typename T> T Foo(T var) { ... };`
2. Вместо слова `typename` можно использовать слово `class`, т. к. в данном контексте они эквивалентны;
3. Шаблонный тип может автоматом выводиться из контекста вызова функции;
4. После объявления используется, как и любой другой тип;
5. Выведение шаблонного типа может происходить либо автоматически, на основе аргументов, либо с помощью явного указания в угловых скобках (`std::max<double>(2, 3.5)`);
6. Цель шаблонных функций: сделать код короче (избавившись от дублирования) и универсальнее.

Основы разработки на C++: жёлтый пояс

Неделя 2

Тестирование



Оглавление

Тестирование	2
2.1 Тестирование и отладка	2
2.1.1 Введение в юнит-тестирование	2
2.1.2 Декомпозиция решения в задаче «Синонимы»	3
2.1.3 Простейший способ создания юнит-тестов на C++	5
2.1.4 Отладка решения задачи «Синонимы» с помощью юнит-тестов	7
2.1.5 Анализ недостатков фреймворка юнит-тестов	10
2.1.6 Улучшаем <code>assert</code>	11
2.1.7 Внедряем шаблон <code>AssertEqual</code> во все юнит-тесты	13
2.1.8 Изолируем запуск отдельных тестов	15
2.1.9 Избавляемся от смешения вывода тестов и основной программы	17
2.1.10 Обеспечиваем регулярный запуск юнит-тестов	18
2.1.11 Собственный фреймворк юнит-тестов. Итоги	19
2.1.12 Общие рекомендации по декомпозиции программы и написанию юнит-тестов	20

Тестирование

2.1. Тестирование и отладка

2.1.1. Введение в юнит-тестирование

Вспомним, что говорилось в курсе «C++: белый пояс». Если решение не принимается тестирующей системой, то нужно:

1. Внимательно перечитать условия задачи;
2. Убедиться, что программа корректно работает на примерах;
3. Составить план тестирования (проанализировать классы входных данных);
4. Тестировать программу, пока она не пройдет все тесты;
5. Если идеи тестов кончились, но программа не принимается, то выполнить декомпозицию программы на отдельные блоки и покрыть каждый из них юнит-тестами.

Как юнит-тесты помогают в отладке:

1. Позволяют протестировать каждый компонент изолированно;
2. Их проще придумывать;
3. Упавшие юнит-тесты указывают, в каком блоке программы ошибка.

2.1.2. Декомпозиция решения в задаче «Синонимы»

На примере задачи «Синонимы» из курса «C++: белый пояс» покажем применение юнит-тестов.

Условие задачи:

Два слова называются синонимами, если они имеют похожие значения. Надо реализовать словарь синонимов, обрабатывающий три вида запросов:

- `ADD word1 word2` – добавить в словарь пару синонимов (`word1`, `word2`);
- `COUNT word` – выводит текущее количество синонимов для слова `word`;
- `CHECK word1 word2` – проверяет, являются ли слова синонимами.

Ввод:	Вывод:
<code>ADD program code</code>	
<code>ADD code cipher</code>	
<code>COUNT cipher</code>	<code>1</code>
<code>CHECK code program</code>	<code>YES</code>
<code>CHECK program cipher</code>	<code>NO</code>

Посмотрим на решение, которое у нас уже есть и которое надо протестировать:

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

int main() {
    int q; // считываем количество запросов
    cin >> q;
    // храним словарь синонимов (для строки хранит множество всех её синонимов)
    map<string, set<string>> synonyms;
    for (int i = 0; i < q; ++i) { // обрабатываем запросы
        string operation_code;
        cin >> operation_code;

        if (operation_code == "ADD") { // считываем две строки и добавляем в словарь
```

```
    string first_word, second_word;
    cin >> first_word >> second_word;
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
} else if (operation_code == "COUNT") { // считываем одну строку и выводим
// размер
    string word;
    cin >> word;
    cout << synonyms[word].size() << endl;
} else if (operation_code == "CHECK") { // считываем два слова и проверяем
    string first_word, second_word;
    cin >> first_word >> second_word;
    if (synonyms[first_word].count(second_word) == 1) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }
}
}
return 0;
}
```

Отправляем в тестирующую систему и видим, что решение не принялось. Он говорит нам, что неправильный ответ на третьем тесте, и больше информации нет. Решение надо тестировать на различных входных данных. Но вроде всё работает и стоит переходить к юнит-тестированию. А для него надо сначала выполнить декомпозицию задачи «Синонимы». Давайте ввод и вывод оставим в `main`, а обработку запроса вынесем в отдельные функции:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
using namespace std;
void AddSynonyms(map<string, set<string>>& synonyms, // функция добавления
                 const string& first_word, const string& second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word);
}
size_t GetSynonymCount(map<string, set<string>>& synonyms, // количество синонимов
                      const string& first_word) {
    return synonyms[first_word].size();
}
```

```

}
bool AreSynonyms(map<string, set<string>>& synonyms, // проверка
                 const string& first_word, const string& second_word) {
    return synonyms[first_word].count(second_word) == 1;
}
int main() {
    int q;
    cin >> q;
    map<string, set<string>> 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;
}

```

2.1.3. Простейший способ создания юнит-тестов на C++

Посмотрим, как писать юнит-тесты и как они должны себя вести на примере функции Sum, которая находит сумму двух чисел.

```
#include <iostream>
```



```
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y;
}
void TestSum() { // собираем набор тестов для функции Sum
    assert(Sum(2, 3) == 5); // мы ожидаем, что 2+3=5
    assert(Sum(-2, -3) == -5); // проверка отрицательных чисел
    assert(Sum(-2, 0) == -2); // проверка прибавления 0
    assert(Sum(-2, 2) == 0); // проверка, когда сумма = 0
    cout << "TestSum OK" << endl;
}
int main() {
    TestSum();
    return 0;
}
// TestSum OK
```

Тесты отработали и не нашли ошибок. Теперь посмотрим, что должно быть при наличии ошибок:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    return x + y - 1; // сделали заведомо неправильно
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(2, 3) == 5' failed ...
```

Нам написали, в каком файле, в какой строке какой **Assert** не сработал. Это облегчает поиск ошибок. Мы можем добиться другой ошибки, например:

```
#include <iostream>
#include <cassert> // подключаем assert'ы
using namespace std;
int sum(int x, int y) {
    if (x < 0) {
        x -= 1
    }
    return x + y;
}
// Assertion fail. main.cpp:7: void TestSum(): Assertion 'Sum(-2, -3) == -5' failed ...
```

Видим, что первый тест прошёл, а на втором уже ошибка. Таким образом, мы можем проверять каждую функцию по отдельности на ожидаемых значениях.

2.1.4. Отладка решения задачи «Синонимы» с помощью юнит-тестов

Для задачи «Синонимы» покроем каждую функцию юнит-тестами и сократим код заменой `map<string, set<string>>` на что-то более короткое:

```
#include <iostream>
#include <string>
#include <map>
#include <set>
#include <cassert>
using namespace std;
using Synonyms = map<string, set<string>>;
// сократили запись типа и везде изменили на Synonyms
void AddSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    synonyms[second_word].insert(first_word);
    synonyms[first_word].insert(second_word); // тут должен не сработать AddSynonyms
}
size_t GetSynonymCount(Synonyms& synonyms, const string& word) {
    return synonyms[word].size();
}
bool AreSynonyms(Synonyms& synonyms, const string& first_word, const string&
    second_word) {
    return synonyms[first_word].count(second_word) == 1;
}

void TestAddSynonyms() { // тестируем AddSynonyms
{
    Synonyms empty; // тест 1
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}}, // ожидаем, что при добавлении синонимов появятся две записи в
        // словаре
        {"b", {"a"}}
    };
    assert(empty == expected);
}
```

```

}
{ // заметим, что мы формируем корректный словарь и ожидаем, что он останется корректным

    Synonyms synonyms = { // если вдруг корректность нарушится, то assert скажет, где
        {"a", {"b"}}, // тест 2
        {"b", {"a", "c"}},
        {"c", {"b"}}
    };
    AddSynonyms(synonyms, "a", "c");
    const Synonyms expected = {
        {"a", {"b", "c"}},
        {"b", {"a", "c"}},
        {"c", {"a", "b"}}
    };
    assert(synonyms == expected);
}
cout << "TestAddSynonyms OK" << endl;
}

void TestCount() { // тестируем Count
{
    Synonyms empty;
    assert(GetSynonymCount(empty, "a") == 0);
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    assert(GetSynonymCount(synonyms, "a") == 2);
    assert(GetSynonymCount(synonyms, "b") == 1);
    assert(GetSynonymCount(synonyms, "z") == 0);
}
cout << "TestCount OK" << endl;
}

void TestAreSynonyms() { // тестируем AreSynonyms
{
    Synonyms empty; // пустой словарь для любых двух слов вернёт false
    assert(!AreSynonyms(empty, "a", "b"));
}
}

```

```

    assert(!AreSynonyms(empty, "b", "a"));
}
{
    Synonyms synonyms = {
        {"a", {"b", "c"}},
        {"b", {"a"}},
        {"c", {"a"}}
    };
    assert(AreSynonyms(synonyms, "a", "b"));
    assert(AreSynonyms(synonyms, "b", "a"));
    assert(AreSynonyms(synonyms, "a", "c"));
    assert(AreSynonyms(synonyms, "c", "a"));
    assert(!AreSynonyms(synonyms, "b", "c")); // false
    assert(!AreSynonyms(synonyms, "c", "b")); // false
}
cout << "TestAreSynonyms OK" << endl;
}
void TestAll() { // функция, вызывающая все тесты
    TestCount();
    TestAreSynonyms();
    TestAddSynonyms();
}
int main() {
    TestAll();
    return 0;
}
// TestCount OK
// TestAreSynonyms OK
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.

```

Видим, что Count и AreSynonyms работают нормально, а вот в AddSynonyms у нас ошибка. Смотрим, что не так. Идём в AddSynonyms и видим, что:

```

synonyms[first_word].insert(first_word); // мы должны к 1 слову добавить 2, а не 1

```

Теперь после исправления все наши тесты отработали успешно. Снова пробуем отправить в тестирующую систему наше решение, закомментировав вызов TestAll(); в main. Тестирование завершилось и решение принято тестирующей системой. Таким образом мы на простом примере продемонстрировали эффективность декомпозиции программы и юнит-тестов.

2.1.5. Анализ недостатков фреймворка юнит-тестов

По ходу разработки юнит-тестов во время решения задачи «Синонимы» мы смогли написать небольшой юнит-тест фреймворк. Посмотрим, что за фреймворк получился. Во-первых, он основан на функции `assert`. Его главный плюс – мы узнаём, какая именно проверка сработала неправильно. На предыдущей задаче мы видели:

```
// main.cpp:26: void TestAddSynonyms(): Assertion 'empty == expected' failed.
```

Основные недостатки:

1. При проверке равенства в консоль не выводятся значения сравниваемых переменных. И мы не знаем, чем была переменная `empty`;
2. После невыполненного `assert` код падает. Если в `TestAll` поставить `TestAddSynonyms` на первое место, то остальные два теста даже не начнутся;
3. Кроме того, у нас пока что результаты тестов выводят ОК в стандартный вывод и смешиваются с тем, что должен вывести код.

В C++ уже существует много фреймворков для работы с тестами, в которых этих недостатков нет.

C++ Unit Testing Frameworks:

1. Google Test
2. CxxTest
3. Boost Test Library

Далее мы свой Unit Testing Framework улучшим для того, чтобы показать, что текущих знаний C++ хватает для таких вещей. И вы будете понимать как он работает, и сможете его менять под свои нужды.

2.1.6. Улучшаем assert

Избавимся от первого недостатка `assert`: когда он срабатывает, мы не видим, чему равен каждый из операндов. Т. е. мы хотим видеть для кода такой вывод:

```
int x = Add(2, 3);
assert(x == 4);
// Assertion failed: 5 != 4
```

И работало оно для любых типов данных:

```
vector<int> sorted = Sort({1, 4, 3});
assert(sorted == vector<int> {1, 3, 4}));
// Assertion failed: [1, 4, 3] != [1, 3, 4]
```

Такие универсальные выводы помогут догадаться о возможной ошибке. Т. е. нам нужна функция сравнения двух переменных какого-то произвольного типа. Напишем шаблон `AssertEqual` перед `TestAddSynonyms()`.

```
#include <exception> // подключим исключения
#include <sstream>    // подключили строковые потоки
...
template <class T, class U>
void AssertEqual (class T& t, const U& u) {
    // значения двух разных типов для удобства
    if (t != u) { // если значения не равны, то мы даём знать, что этот assert не сработал
        ostringstream os;
        os << "Assertion failed: " << t << "!=" << u;
        throw runtime_error(os); // бросим исключение с сообщением со значениями t и u
    }
}
```

Встроим это в `TestCount()`. Заменим `assert` на наш `AssertEqual` внутри `TestCount()`.

```
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"), 0);
        AssertEqual(GetSynonymCount(empty, "b"), 0);
    }
    {
        Synonyms synonyms = {
```

```

    {"a", {"b", "c"}},
    {"b", {"a"}},
    {"c", {"a"}}
};
AssertEqual(GetSynonymCount(synonyms, "a"), 2);
AssertEqual(GetSynonymCount(synonyms, "b"), 1);
AssertEqual(GetSynonymCount(synonyms, "z"), 0);
}
}
// Warning ... comprison between signed and unsigned ...

```

Код скомпилировался, но мы получили Warning из-за сравнения между знаковым и беззнаковым типами в `AssertEqual`. Это происходит потому, что все константы (2, 1 и 0 в нашем случае) имеют тип `int` (как уже было сказано в неделе 1), который мы сравниваем с типом `size_t`. Исправляем это, дописав к ним `u` справа: `1 → 1u` и т. д.

Теперь, сделав нарочную ошибку где-нибудь в `GetSynonymCount`, мы получим предупреждение: `Assertion failed: 1 != 0`. Но пока мы не видим, какой именно `Assert` сработал. Исправим это, передавая в `Assert` строчку `hint` и также добавим каждому `Assert`'у строку идентификации, по которой мы сможем однозначно понять, какой именно `Assert` выдал ошибку:

```

void AssertEqual (class T& t, const U& u, const string& hint) {
    if (t != u) {
        ostreamstream os;
        os << "Assertion failed: " << t << "!=" << u << "Hint: " << hint;
        throw runtime_error(os);
    }
}
...
void TestCount() {
    {
        Synonyms empty;
        AssertEqual(GetSynonymCount(empty, "a"),
            0u, "Synonym count for empty dict a");
        AssertEqual(GetSynonymCount(empty, "b"),
            0u, "Synonym 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");
}
}
// Asserting failed: 1!= 0 Hint: Synonym count for empty dict

```

2.1.7. Внедряем шаблон AssertEqual во все юнит-тесты

Добавим `Assert` в `AreSynonyms`. Только `AssertEqual` нам не подходит, потому что в данной функции у нас только два константных значения: `true` и `false`. Вместо этого напомним аналог классического `assert`, который назовём `Assert` (C++ чувствителен к регистру). И если мы испортим функцию `AreSynonyms`, то получим соответствующую ошибку с подсказкой.

```

void Assert(bool b, const string& hint) {
    AssertEqual(b, true, hint);
}
... // модернизируем наш TestAreSynonyms
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 nonempty c b");
    }
}

```



```

    }
}
// Asserting failed: 0!= 1 Hint: AreSynonyms empty a b

```

Получили нужную ошибку, и по ней мы можем увидеть, где что-то не так. Осталась только функция `AddSynonyms`, и ловим ошибку:

```

void TestAddSynonyms() {
{
    Synonyms empty;
    AddSynonyms(empty, "a", "b");
    const Synonyms expected = {
        {"a", {"b"}},
        {"b", {"a"}}},
    };
    AssertEqual(empty, expected, "Add to 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");
}
}
// no match for 'operator <<' (operand types are std::ostream<char> and std::map ...)

```

Ошибка произошла с `empty` и `synonyms`, которые являются `map<string, set<string>`. Мы пытаемся их вывести в стандартный поток вывода (см. неделя 1). Пишем перегрузку оператора вывода для `map` и для `set`, предварительно исправив ошибку в `AreSynonyms`, допустим ошибку в `AddSynonyms` и поймаем её:

```

template <class T> // учимся выводить в поток set
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> // учимся выводить в поток map
ostream& operator << (ostream& os, const map<K, V>& m) {
    os << "{";
    bool first = true; // грамотная расстановка запятых
    for (const auto& kv : m) {
        if (!first) {
            os << ", ";
        }
        first = false;
        os << kv.first << ": " << kv.second;
    }
    return os << "}";
}

// Asserting failed: {a: {a}, b: {a} } != {a: {b}, b: {a}. Hint: Add to empty

```

Таким образом, мы внедрили шаблон `AssertEqual`, который позволяет найти ошибку, узнать, с чем она возникла и найти конкретное место в коде благодаря подсказке.

2.1.8. Изолируем запуск отдельных тестов

Теперь исправим следующий недостаток `Assert`: если он срабатывает, то код падает и другие тесты не выполняются. Аварийное завершение программы у нас возникало из-за вылета исключения в `AssertEqual`. Теперь будем ловить эти исключения в `main`:

```

int main() {
    try {
        TestAreSynonyms(); // ловим исключение
    }
}

```

```
} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAreSynonyms" << " fail: " << e.what() << endl;
} // если мы словили исключение, то работа всё равно продолжится
try {
    TestCount();
} catch (runtime_error& e) {
    ++fail_count;
    cout << " TestCount" << " fail: " << e.what() << endl;
}
try {
    TestAddSynonyms();
} catch (runtime_error& e) {
    ++fail_count;
    cout << "TestAddSynonyms" << " fail: " << e.what() << endl;
}
}
```

Но это неудобно, код дублируется. Нам бы хотелось, чтобы этот `try/catch` и вывод были написаны в одном месте, а мы туда могли бы передавать различные тестовые функции, и они бы там выполнялись, и исключения бы от них ловились, всё бы работало. В C++ это можно сделать, ведь функции имеют тип и их можно передавать в другие функции как аргумент. Создадим шаблон функции `RunTest`, который будет запускать тесты и ловить исключения.

```
...
template <class TestFunc>

void RunTest(TestFunc func, const string& test_name) { // передаём тест и его имя
    try {
        func();
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // ловим исключение
    }
}

int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
}
```

Заметим, что все тесты работают в любом порядке. Таким образом, мы с вами применили шаблон функций, для того чтобы передавать в качестве параметров функции другие функции, и смогли за счет этого написать универсальный такой шаблон, который для любого юнит-теста ловит исключения и позволяет нам все юнит-тесты, которые мы написали, выполнять при каждом запуске нашей программы.

2.1.9. Избавляемся от смешения вывода тестов и основной программы

Добавим в `RunTest` ещё одну удобную вещь: будем выводить OK снаружи каждого юнит-теста.

```
void RunTest(TestFunc func, const string& test_name) {
    try {
        func(); // заменим cout на cerr - стандартный поток ошибок
        cerr << test_name << " OK" << endl; // выводит OK, если всё работает
    } catch (runtime_error& e) {
        cerr << test_name << " fail: " << e.what() << endl; // fail, если ошибка
    }
}
```

Всё это время сам алгоритм решения задачи «Синонимы» (который всё это время был закомментирован), всё ещё хорошо работает. Вот он сам:

```
int main() { // когда передаём функцию как параметр, передаём только её имя без скобочек
    RunTest(TestAreSynonyms, "TestAreSynonyms");
    RunTest(TestCount, "TestCount");
    RunTest(TestAddSynonyms, "TestAddSynonyms");
    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;
    }
}
}
}
}

```

Вся проблема в том, что и юнит-тесты, и сама программа выводят в стандартный вывод. Пусть юнит-тесты выводят в `stderr` (стандартный поток ошибок). По окраске вывода в Eclipse можно отличать **стандартный вывод**, **стандартный ввод** и **стандартный поток ошибок**.

Вывод: **TestAreSynonyms OK, 5 TestAddSynonyms OK, TestCount OK, 1, COUNT a, 0**. Теперь не нужно окружать комментарием эти части кода перед отправкой, ведь они всё равно не повлияют на работу самой программы.

2.1.10. Обеспечиваем регулярный запуск юнит-тестов

Мы хотим, чтобы юнит-тесты были автоматическими, и их код находился где-то отдельно. Кроме того, стоит считать ошибки, чтобы если существует хоть одна, то программа не ждала получения данных от пользователя. Таким образом, мы хотим:

1. Запускаем тесты при старте программы. Если хоть один тест упал, программа завершается;
2. Если все тесты прошли, то должно работать решение самой задачи.

Для этого обернём наш шаблон `RunTest` в класс `TestRunner`:

```

class TestRunner { // класс тестирования
public:
    template <class TestFunc>
    void RunTest(TestFunc func, const string& test_name) {

```

```

    try { // RunTest стал шаблонным методом класса
        func();
        cerr << test_name << " OK" << endl;
    } catch (runtime_error& e) {
        ++fail_count; // увеличиваем счётчик упавших тестов
        cerr << test_name << " fail: " << e.what() << endl;
    }
}

~TestRunner() { // деструктор класса TestRunner, в котором анализируем fail_count
    if (fail_count > 0) { //это как раз тот момент, когда
        cerr << fail_count << " unit tests failed. Terminate" << endl;
        exit(1); // завершение программы с кодом возврата 1
    }
}

private:
    int fail_count = 0; // счётчик числа упавших тестов
};

void TestAll() { // переместили все тесты в одну функцию
    TestRunner tr;
    tr.RunTest(TestAddSynonyms, "TestAddSynonyms");
    tr.RunTest(TestCount, "TestCount");
    tr.RunTest(TestAreSynonyms, "TestAreSynonyms");
}

int main() {
    TestAll(); // т.к. мы деструктор класса объявили в самом классе,
...} // выполняется он в конце TestAll

```

Теперь, если мы словили хоть одну ошибку, то программа перестанет выполняться с кодом возврата 1. Если же ничего плохого не произошло, то мы можем ввести число команд и сами команды.

2.1.11. Собственный фреймворк юнит-тестов. Итоги

Подведём итоги написания собственного фреймворка. Его основные свойства:

- Если срабатывает `assert`, в консоль выводятся его аргументы (работает для контейнеров);
- Вывод тестов не смешивается с выводом основной программы;

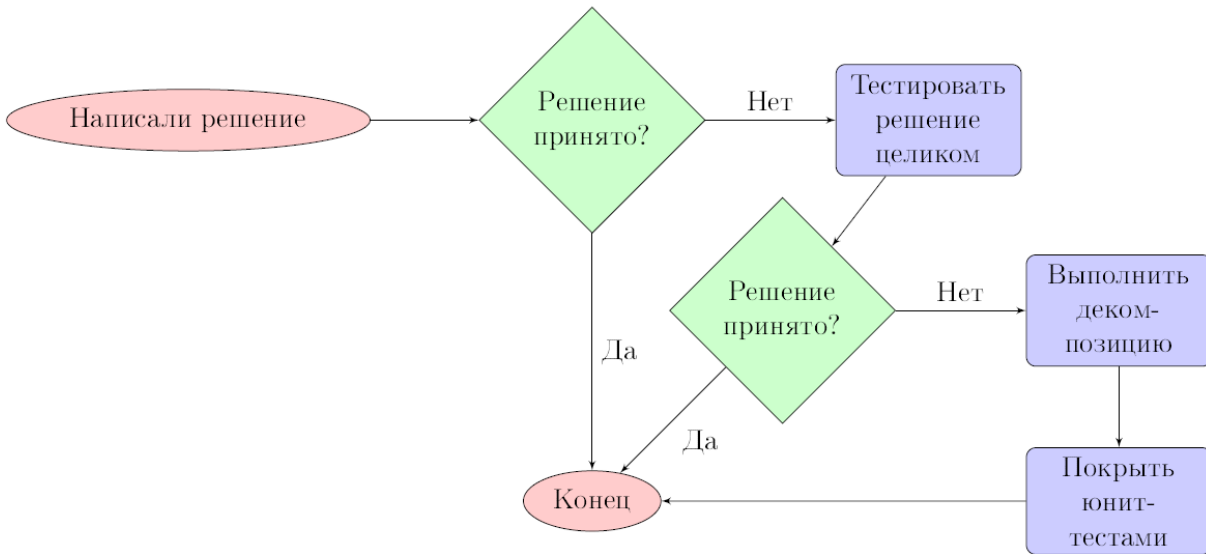
- При каждом запуске программы выполняются все юнит-тесты;
- Если хотя бы один тест упал, программа завершится с ненулевым кодом возврата.

Для того, чтобы пользоваться фреймворком, надо написать:

```
void TestSomething() { // функция, что-то тестирующая
    AssertEqual(..., ...);
    // выполняем какие-то проверки с помощью AssertEqual
}
void TestAll() {
    TestRunner tr;
    tr.RunTest(TestSomething, "TestSomething")
    // вызываем методом RunTest
}
int main() {
    TestAll(); // должна быть до самой программы
} // код фреймворка выложен рядом с видео
```

2.1.12. Общие рекомендации по декомпозиции программы и написанию юнит-тестов

Общий алгоритм решения задач с помощью декомпозиции и юнит-тестов:



Но кроме этой схемы, стоит выполнять декомпозицию задачи по ходу написания самого кода. Декомпозицию лучше делать сразу:

- Отдельные блоки проще реализовать и вероятность допустить ошибку ниже;
- Их проще тестировать, соответственно, выше вероятность найти ошибку или убедиться в её отсутствии;
- В больших проектах декомпозиция упрощает понимание и переиспользование кода;
- Уже реализованные функции можно брать и использовать в другом месте;
- Сама декомпозиция иногда защищает от ошибок.

Вспомним задачу «Уравнение» из курса «C++: белый пояс». Нужно было найти все различные действительные корни уравнения $Ax^2 + By^2 + C = 0$. Гарантируется, что $A^2 + B^2 + C^2 > 0$. Монолитное решение могло выглядеть так:

```

#include <cmath>
#include <iostream>
using namespace std;
int main() {

```



```
double a, b, c, D, x1, x2;
cin >> a >> b >> c;
D = b * b - 4 * a * c;
if ((a == 0 && c == 0) || (b == 0 && c == 0)) {
    cout << 0;
} else if (a == 0) {
    cout << -(c / b); // тут ошибка. Если b == 0, мы всё равно разделим на 0
} else if (b == 0) {
    cout << " ";
} else if (c == 0) {
    cout << 0 << " " << -(b / a);
} else if (D < 0) {
    cout << " ";
} else if (D == 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    cout << x1;
} else if (D > 0) {
    x1 = ((-1 * b) + sqrt(D)) / (2 * a);
    x2 = ((-1 * b) - sqrt(D)) / (2 * a);
    cout << x1 << " " << x2;
}
return 0;
}
```

Быстро просмотрев этот код, сложно понять, работает он или нет. А в нём есть ошибка, которую из-за монолитности кода сложно заметить сразу. Теперь рассмотрим декомпозированное решение той же задачи:

```
#include <cmath>
#include <iostream>
using namespace std;

void SolveQuadraticEquation(double a, double b, double c) {
    // ... тут решение гарантированного квадратного уравнения
}

void SolveLinearEquation(double b, double c) {
    // b * x + c = 0
    if (b != 0) { // тут не забыли проверить деление на 0
        cout << -c / b;
    }
}
```

```
}

int main() {
    double a, b, c;
    cin >> a >> b >> c;
    if (a != 0) { // точно знаем, что уравнение квадратное
        SolveQuadraticEquation(a, b, c);
    } else { // просто решаем линейное
        SolveLinearEquation(b, c);
    }
    return 0;
}
```

Юнит-тесты тоже лучше делать сразу. Причины:

- Разрабатывая тесты, вы сразу продумываете все варианты использования вашего кода и все крайние случаи входных данных;
- Тесты позволяют вам сразу проконтролировать корректность вашей реализации (особенно актуально для больших проектов);
- В больших проектах обширный набор тестов позволяет убедиться, что вы ничего не сломали во время дополнения кода.



Основы разработки на C++: жёлтый пояс

Неделя 3

Разделение кода по файлам



Оглавление

Разделение кода по файлам	2
2.1 Распределение кода по файлам	2
2.1.1 Введение в разработку в нескольких файлах на примере задачи «Синонимы»	2
2.1.2 Механизм работы директивы <code>#include</code>	9
2.1.3 Обеспечение независимости заголовочных файлов	11
2.1.4 Проблема двойного включения	12
2.1.5 Понятия объявления и определения	14
2.1.6 Механизм сборки проектов, состоящих из нескольких файлов	17
2.1.7 Правило одного определения	27
2.1.8 Итоги	29

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 создается пустой файл test_runner.h.

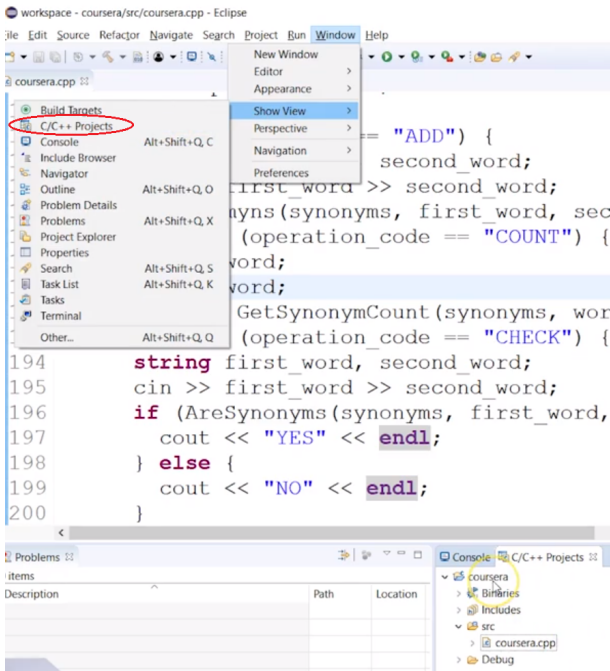


Рис. 2.1: Открытие C/C++ projects

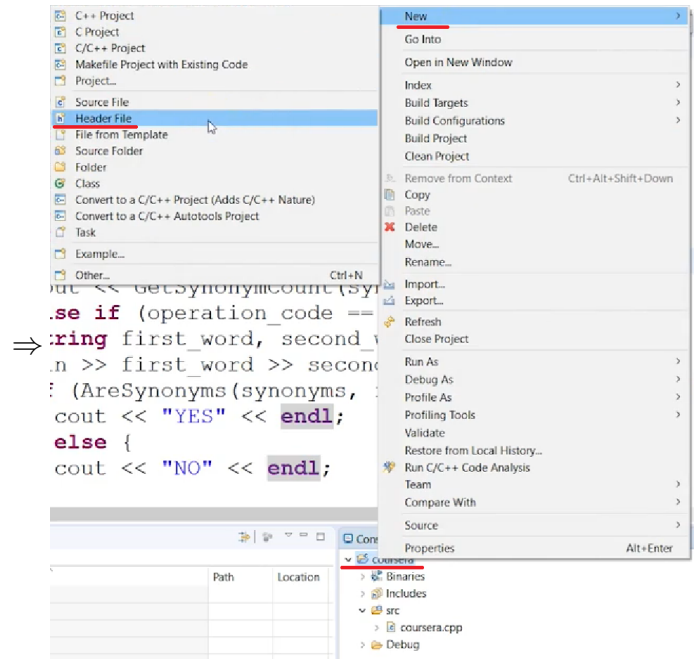


Рис. 2.2: New → Header File

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 функцию `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 проекта 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)

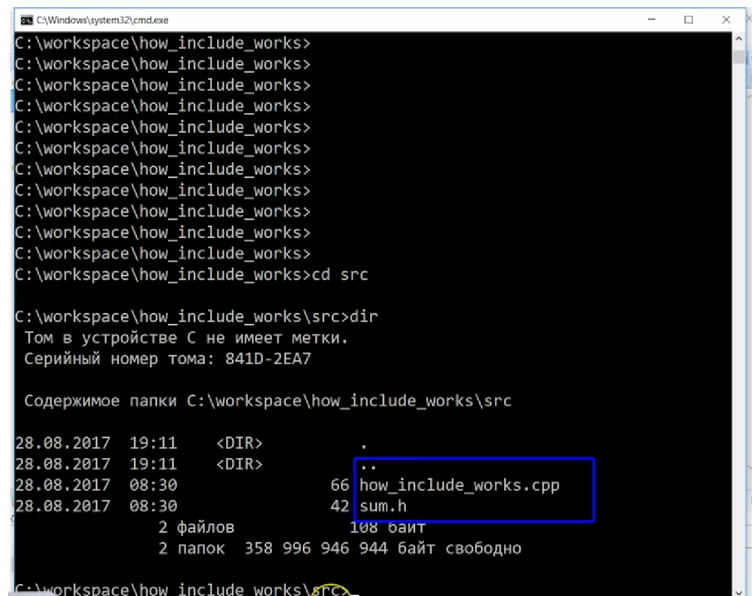


Рис. 2.3: Консоль cmd

Вызовем команду компилятора:

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

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

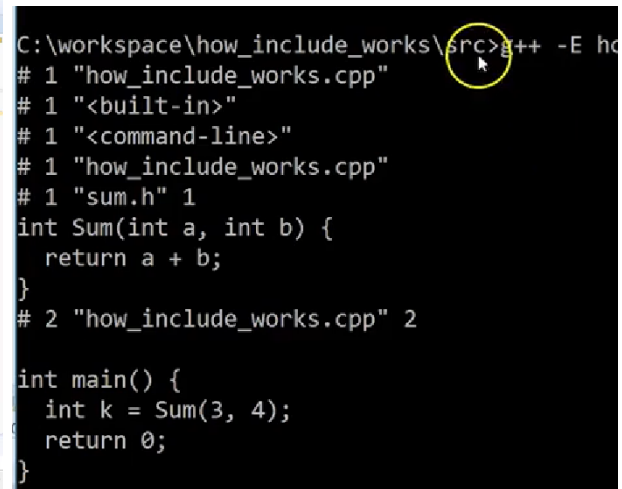


Рис. 2.4: Преппроессинг проекта

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:

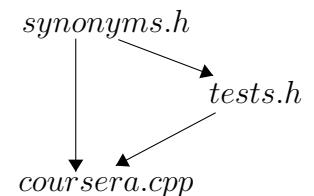
```
#include "test_runner.h"
```

```
#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. ОК, 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 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);
```

```
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

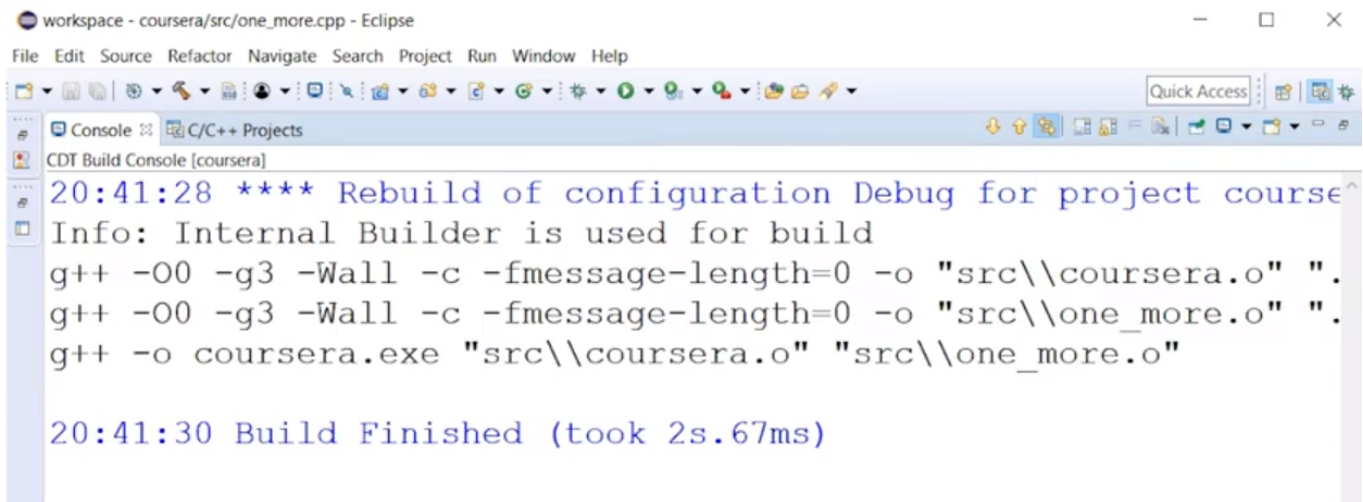
Когда мы начинали разговор о разделении кода на несколько файлов, то в качестве одного из недостатков хранения всего кода в одном файле мы называли то, что при минимальном изменении программы у нас она пересобирается вся, в случае, если весь код лежит в одном файле. Сейчас мы разделили код нашего проекта 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 `coursea.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 `coursea.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:



```

workspace - coursera/src/one_more.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
CDT Build Console [coursera]
20:41:28 **** Rebuild of configuration Debug for project course
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\one_more.o" ".
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"

20:41:30 Build Finished (took 2s.67ms)
  
```

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.

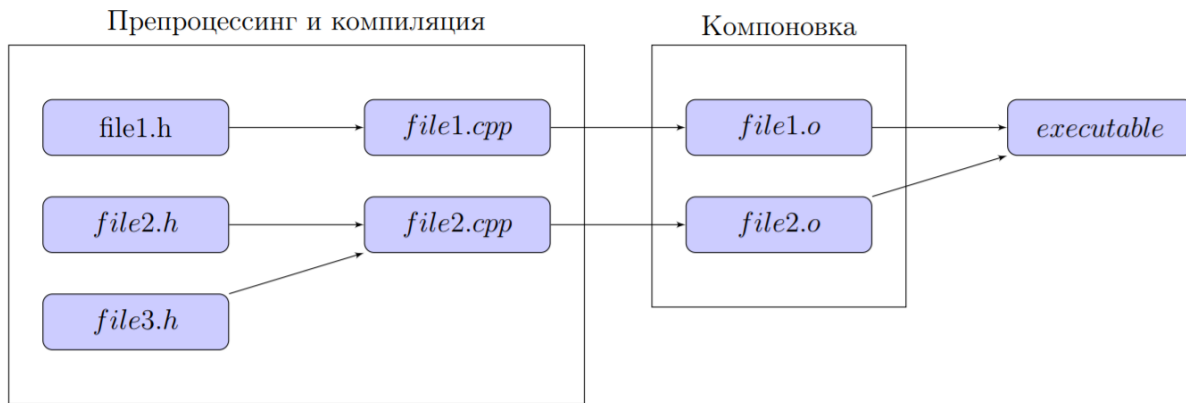


Рис. 2.6: Компиляция нескольких файлов

Как мы уже видели, первая стадия – это препроцессинг, когда выполняются все директивы `include`. Далее, 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.

```

20:44:29 **** Incremental Build of configuration Debug for proj
Info: Internal Builder is used for build
g++ -O0 -g3 -Wall -c -fmessage-length=0 -o "src\one_more.o" "src\one_more.cpp"
g++ -o coursera.exe "src\coursera.o" "src\one_more.o"
20:44:29 Build Finished (took 590ms)

```

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"), 0,

```



```
    "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.

Multiple definition of GetSynonymCount

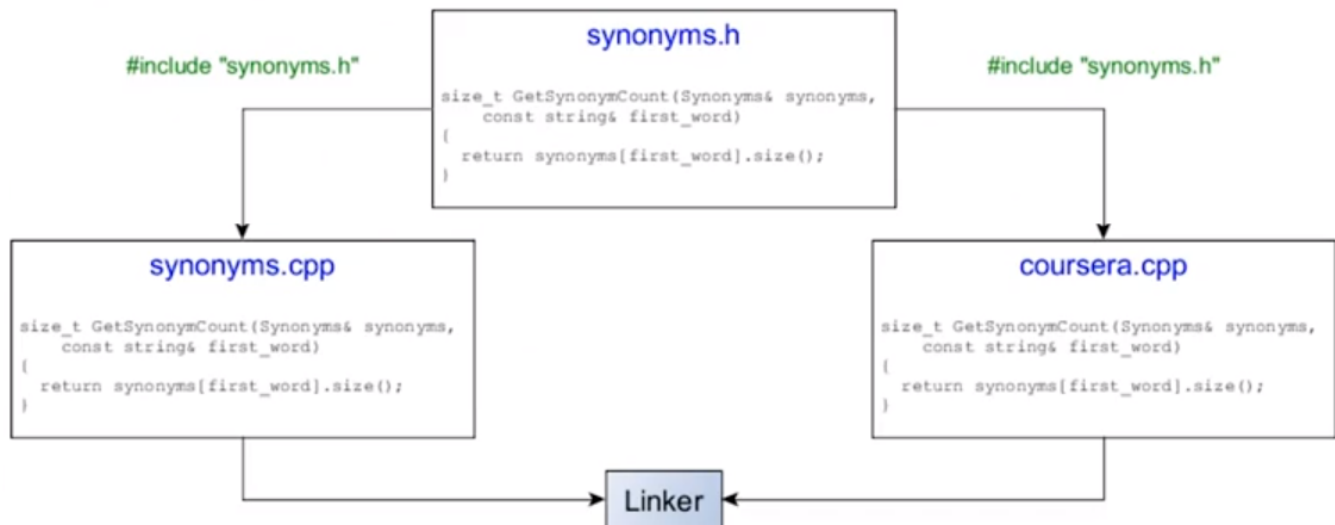


Рис. 2.9: Схема компиляции и сборки проекта

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.



Основы разработки на C++: жёлтый пояс

Неделя 4

Итераторы, алгоритмы, контейнеры



Оглавление

Итераторы, алгоритмы, контейнеры	3
4.1 Введение в итераторы	3
4.1.1 Введение в итераторы	3
4.1.2 Концепция полуинтервалов итераторов	5
4.1.3 Итераторы множеств и словарей	8
4.1.4 Продвинутое итерирование по контейнерам	9
4.2 Использование итераторов в алгоритмах и контейнерах	11
4.2.1 Использование итераторов в методах контейнеров	11
4.2.2 Использование итераторов в алгоритмах	12
4.2.3 Обратные итераторы	14
4.2.4 Алгоритмы, возвращающие набор элементов	15
4.2.5 Итераторы <code>inserter</code> и <code>back_inserter</code>	17
4.2.6 Отличия итераторов векторов и множеств	18
4.2.7 Категории итераторов, документация	19
4.3 Очередь, дек и алгоритмы поиска	20

4.3.1	Стек, очередь и дек	20
4.3.2	Алгоритмы поиска	22
4.3.3	Анализ распространённых ошибок	25

Итераторы, алгоритмы, контейнеры

4.1. Введение в итераторы

4.1.1. Введение в итераторы

Рассмотрим задачу. Пусть у нас есть вектор строк с языками программирования:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
    return 0;
}
```

Зададимся целью найти в нашем векторе язык, начинающийся с буквы 'C' или сказать, что такого языка нет. Можно написать цикл `for` и проверять первую букву каждого языка, но чтобы знать, нашёлся язык или нет, нам нужно хранить флажок типа `bool` и с каждым шагом всё больше шанс ошибиться. Но существует стандартный алгоритм `find_if()`, принимающий начало диапазона, конец диапазона и лямбда-функцию, которая ищет нужный язык. Дописываем строчку:

```
#include <algorithm> // подключаем модуль с алгоритмами
...
auto result = find_if(
    begin(langs), end(langs), [](const string& lang) { // лямбда-функция по языкам
        return lang[0] == 'C'; // возвращает true, если нулевая буква = 'C'
    });
cout << *result; // вызываем * от find_if, чтобы обратиться к найденному элементу
// C++
```

Видим, что нашли первый подходящий язык. Этот результат можно сохранить в какую-нибудь строку:

```
string& ref = *result; // сохраняем в строку
cout << ref << endl; // выводим
```

Причём, поскольку `*result` – неконстантная ссылка, мы по ней можем поменять элемент. Например:

```
string& ref = *result; // сохраняем в строку C++
ref = "D++";
cout << *result << endl;
// D++
```

А если у нас в векторе лежат не строки, а более сложные структуры:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang { // у каждого языка программирования есть название и возраст
    string name;
    int age;
};
int main() {
    vector<Lang> langs = // тоже поменяли на Lang
        {"Python", 26},
        {"C++", 34},
        {"C", 45},
        {"Java", 22},
        {"C#", 17}};

    auto result = find_if(
        begin(langs), end(langs),
        [](const Lang& lang) { // лямбда-функция немного меняется
            return lang.name[0] == 'C';
        });
    if (result == end(langs)) { // если результат выводит ссылку на конец диапазона
        cout << "Not found"; // значит, элемент не найден
    } else { // иначе мы нашли элемент
        cout << (*result).age << endl; // выводим возраст первого попавшегося языка на C
    }
}
```

```
    return 0;
}
// 34
```

Если результат не нашёлся, то `result` указывает на конец диапазона. Это значит, что мы можем спокойно проверить на наличие. Кроме того, обращаться можно короче:

```
cout << result->age << endl; // более удобная форма
```

А для языков на букву 'D' получим: `Not Found`. И `begin()`, `end()` и `result` указывают на какую-то позицию в контейнере. Все типы, указывающие на какую-то позицию контейнера, называются **итераторами**. А операция `*` над итератором называется **разыменованием итератора**. Выведем начало контейнера:

```
cout << begin(langs)->name << " " << langs.begin()->age << endl;
// Python 26
```

А вот `end(langs)` уже указывает не на последний элемент контейнера, а на конец. Заметим, что к началу можно обращаться методом `langs.begin()`.

4.1.2. Концепция полуинтервалов итераторов

Итераторы есть не только у вектора, но и у любого контейнера. Например, у строки:

```
#include <string>
...
string lang = langs[1].name;
auto it = begin(lang);
cout << *it;
// C
```

`begin()` у строки указывает на её первый символ. Пока что мы с их помощью только получали элемент контейнера, но из названия следует, что с их помощью можно итерироваться по контейнеру. Допишем:

```
++it; // получим следующий символ
cout << *it
// C+
```

Теперь мы вывели сначала нулевой элемент, а затем первый. Давайте с помощью цикла `for` проитерируемся по вектору `langs`:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
struct Lang {
    string name;
    int age;
};
int main() {
    vector<Lang> langs = // тоже поменяли на Lang
        {"Python", 26},
        {"C++", 34},
        {"C", 45},
        {"Java", 22},
        {"C#", 17}};
    for (auto it = begin(langs); it != end(langs); ++it) {
        cout << it-> name << " ";
    }
    return 0;
}
// Python C++ C Java C#
```

Таким образом мы вывели все языки. Когда итератор показывает на `begin(langs)`, выводится первый элемент. Итератор двигается вправо, проверяет, что он не достиг конца и т. д. Выводим последний элемент и сдвигаем итератор на `end`. Т. е. `end(langs)` – это итератор сразу за последним элементом. Получается полуинтервал $[begin, end)$.

Для 5 элементов получаем 5 элементов и `end`. А вот пустой диапазон представляется из равных итераторов `begin` и `end`. Попробуем обратиться к полю `end(langs)`. Код компилируется, но падает, потому что там либо чужая память, либо вообще ничего не лежит. Напишем универсальную функцию `PrintRange`, принимающую два итератора на `Lang`:

```
using LangIt = vector<string>::iterator; // укорачиваем запись
void PrintRange(LangIt range_begin, // начало
    LangIt range_end) { // конец
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " "; // мы не хотим переопределять вывод структур
    }
}
```

```

}
int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
    PrintRange(begin(langs), end(langs));
    return 0;
}
// Python C++ C Java C#

```

Всё снова работает. Вспомним, что итераторы есть для разных контейнеров, и хотелось бы написать универсальный итератор. Цикл `for` уже достаточно универсален. Напишем шаблонную функцию:

```

template <typename It> // написали шаблонную функцию
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора
                It range_end) { // и тут
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " ";
    }
}
...
PrintRange(begin(langs), end(langs)); // проходимся по контейнеру с языками
PrintRange(begin(langs[0]), end(langs[0])); // проходимся по контейнеру с python
// Python C++ C Java C# P y t h o n

```

Теперь, используя концепцию полуинтервалов, попробуем вывести половину векторов по отдельности, например, все языки строго до 'C' и отдельно все языки начиная с него:

```

auto border = find(begin(langs), end(langs), "C"); // раздел
PrintRange(begin(langs), border); // Python, C++
PrintRange(border, end(langs)); // C, Java, C#

```

Мы просто разбили по нужному языку наш полуинтервал на два диапазона.
Заключение:

- Итератор – способ задать позицию в контейнере;
- `begin(c)` и `end(c)` – границы полуинтервала [*begin*, *end*);
- Алгоритмы часто принимают пару итераторов, образующую полуинтервал;
- С помощью шаблонов легко написать свой универсальный алгоритм, например, `PrintRange`.

4.1.3. Итераторы множеств и словарей

Давайте посмотрим, как работает итератор от других контейнеров. Увидим, что по сути они работают точно так же. Например, множество. Давайте возьмем наш вектор языков и заменим его на множество языков.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>    // подключили множества
using namespace std;
template <typename It> // написали шаблонную функцию
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора
               It range_end) { // и тут
    for (auto it = range_begin; it != range_end; ++it) {
        cout << *it << " ";
    }
}

int main() {
    set<string> langs = {"Python", "C++", "C", "Java", "C#"};
    PrintRange(begin(langs), end(langs));
    // тоже выводит названия, но в алфавитном порядке
    auto it = langs.find("C++");
    PrintRange(begin(langs), it);
    // выведем все языки до C++, в отсортированном порядке
    return 0;
}
// C C# C++ Java Python C C#
```

Получим вывод сначала всех строк, а потом строк, которые лексикографически меньше C++. Теперь рассмотрим словарь. Но для вызова `PrintRange` должен быть определён оператор `<<`.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>    // подключили словари
using namespace std;
void PrintRange...
int main() {
    map<string, int> langs = // словарь Имя-возраст
```

```
    {"Python", 26},
    {"C++", 34},
    {"C", 45},
    {"Java", 22},
    {"C#", 17}];
return 0;
}
```

Какой тип у элементов словаря? Когда вы итерировались по нему `range based for`'ом, этот тип был парой. И здесь `*it` – это пара. Переопределять оператор вывода для словаря мы уже умеем. Создадим функцию `PrintMapRange` для вывода словаря:

```
void PrintMapRange(It range_begin, It range_end) {
    for (auto it = range_begin; it != range_end; ++it){
        cout << it->first << '/' << it->second << " ";
    }
}

...
auto it = langs.find("C++");
PrintMapRange(begin(langs), it); // вызовем наш
return 0;
}
// C/45 C#/17
```

Видим, что вывелись нужные нам языки и их возраст.

4.1.4. Продвинутое итерирование по контейнерам

Продemonстрируем, что итераторы универсальнее, чем `range-based for`, что их можно использовать в гораздо более широком числе случаев. Вернёмся к множеству языков. Сделаем множество строк с языками. И с помощью итераторов выведем множество в обратном порядке с помощью `--it`.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
```

```
auto it = end(langs);
while (it != begin(langs)) {
    --it;
    cout << *it << " ";
}
}
// C# Java C C++ Python
```

Всё работает. Итерирование в обратную сторону работает так: проверяем, что `it != begin`, двигаем его влево и уже можем вывести последний элемент. И т. д., пока не доходим до начала. Мы вывели вектор в обратном порядке. Кроме того, тот же код можно переписать в виде:

```
while (it != begin(langs)) { // заметим, что нельзя делать --it от it = begin
    --it;
    cout << *it << " ";
}
```

Опасные операции над итераторами:

- `*end(c)` – обращаться к элементу после последнего;
- `auto it = end(c); ++it;` – смотреть на элемент после `end`;
- `auto it = begin(c); --it;` – смотреть на элемент до `begin`.

Итераторы очень похожи на ссылки, но есть разница.

Отличия итераторов от ссылок:

- Итераторы могут указывать «в никуда» – на `end`. Ссылка всегда привязана к чему-то;
- Итераторы можно перемещать на другие элементы:

```
vector<int> numbers = {1, 3, 5};
auto it = numbers.begin();
++it; // it указывает на 3
int& ref = numbers.front();
++ref; // теперь numbers[0] == 2
```

В итоге мы узнали, что у всех контейнеров есть итераторы, и чем они отличаются от ссылок. И итераторы предоставляют универсальный способ обхода контейнеров.

4.2. Использование итераторов в алгоритмах и контейнерах

4.2.1. Использование итераторов в методах контейнеров

Где мы раньше встречали итераторы?

1. В алгоритмах, например `sort`, `count`, `count_if`, `reverse`, `find_if`;
2. Конструкторы вектора и множества;
3. Метод `find` у множества.

Рассмотрим методы контейнеров на примере вектора строк.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = find(begin(langs), end(langs), "C++"); // получаем позицию C++
langs.erase(it); // метод удаления элемента из контейнера по итератору it
langs.insert(begin(langs), "C++") // вставка перед итератором элемента C++
langs.erase(it, end(langs)); // удалили все с C++ и до конца
// C# Java C C++ Python
```

У вектора длины 5 есть 6 позиций для вставки от `v.insert(begin(), value)` до `v.insert(end(), value)`. Вставка в произвольное место вектора:

- Вставка диапазона

```
v.insert(it, range_begin, range_end);
// вставит диапазон [range_begin, range_end) в позицию it
```

- Вставка элемента несколько раз

```
v.insert(it, count, value);
// count раз вставляет элемент value в позицию it
```

- Вставка нескольких элементов

```
v.insert(it, {1, 2, 3});
// вставляет 1, 2, 3 в позицию it
```

4.2.2. Использование итераторов в алгоритмах

Рассмотрим алгоритмы, которые можно применять к векторам.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
remove_if(begin(langs), end(langs),
    [](const string& lang) { // лямбда-функция, по которой удаляем
        return lang[0] == 'C'; // хотим удалить все языки, начинающиеся на C
    });
PrintRange(begin(langs), end(langs));
// Java Python C C#
```

Но у нас не удалились C и C#. Оказывается даже у стандартных алгоритмов C++ итераторы – это настолько общая концепция, что они не позволяют обратиться к исходным контейнерам. Поэтому алгоритм `remove_if` может лишь подсказать вам, помочь вам удалить что-то из контейнера. Он помог. Как он помог? Всё, что должно в контейнере остаться, перебросил в начало этого контейнера. И он вернул новый конец вашего вектора, то, что должно стать его концом. И теперь ваша задача сделать то, что вернул `remove_if`, новым концом вектора. Как это можно сделать? Например, с помощью метода `erase`.

```
langs.erase(it, end(langs)); // удаляем начиная с итератора и до конца
// Java Python
```

Ещё раз: `remove_if` ничего не удаляет. Он лишь помогает нам удалить, потому что общие алгоритмы не могут влиять на контейнер (например, изменять его размер).

```
vector<string> langs = {"Python", "C++", "C++", "Java", "C++"};
// оставляем из подряд идущих повторов только один элемент
auto it = unique(begin(langs), end(langs));
langs.erase(it, end(langs)); // удаляем повторяющиеся, которые выкинуты в конец
PrintRange(begin(langs), end(langs));
// Python C++ Java C++
```

Таким образом мы удалили подряд идущие дубликаты элементов. Получается, что можно оставить в векторе только уникальные элементы, сначала их отсортировав, потом удалив повторы, то есть сначала `sort`, потом `unique`, и даже не нужно создавать их множество для этого, так получается экономнее.

Еще есть алгоритм нахождения минимума в массиве:

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
```

```
auto it = min_element(begin(langs), end(langs)); // кладем в итератор мин. элемент
cout << *it << endl; // min_element может вернуть end(langs), только если langs пуст
// C
```

```
auto it = max_element(begin(langs), end(langs)); // максимальный элемент
cout << *it << endl;
// Python
```

```
auto p =
    minmax_element(begin(langs), end(langs)); // пара - min и max в контейнере
cout << *p.first << ' ' << *p.second << endl;
// C Python
```

Таким образом, мы умеем получать минимальный и максимальный элементы в векторе. Но имеет ли смысл применять такие алгоритмы ко множеству? Ответ – нет. На множестве (`set`) нам достаточно взять `it = begin(langs)` для получения минимального элемента и `it = end(langs); --it;` для получения максимального.

Теперь попробуем вызвать `remove` для элементов множества:

```
set<string> langs = {"Python", "C++", "C", "Java", "C#"};
remove(begin(langs), end(langs), "C");
```

Но оно не скомпилировалось и выдало много ошибок. Вспомним, что делает `remove` – он переставляет элементы в конец для удаления. А множество не даёт переставлять элементы.

Но есть, например, алгоритм, проверяющий какое-то свойство для всего диапазона:

```
set<string> langs = {"Python", "C++", "C", "Java", "C#"};
cout << all_of(begin(langs), end(langs), [](const string& lang) {
    return lang[0] >= 'A' && lang[0] <= 'Z'; // все названия с большой буквы
}) << endl;
// 1
```

Все элементы начинаются с заглавной английской буквы и алгоритм выдал `true` (1). Если изменить первую букву какого-нибудь языка на малую, начнёт выводить 0. Что мы узнали?

1. Методы вектора, позволяющие вставить в конкретное место вектора или удалить: `insert`, `erase`;
2. Алгоритмы:

- `remove_if` – оставляет в начале вектора элементы, не удовлетворяющие условию;
- `unique` – оставляет в начале вектора по одному элементу из группы подряд идущих повторений;
- `min_element`, `max_element`, `minmax_element` – `min`, `max` элементы;
- `all_of` – проверка условия для всех элементов.

4.2.3. Обратные итераторы

Бывают не только обычные итераторы. Например, вывод в обратном порядке можно сделать так:

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
PrintRange(rbegin(langs), rend(langs)); // обратные итераторы
// C# Java C C++ Python
```

`rbegin()` и `rend()` от слова `reverse` – перевернутый. Это **обратные итераторы**.

Причём `*rbegin(langs) = "C#"`, а вот `*rend()` не скомпилируется.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = rbegin(langs); // итератор на последний элемент
cout << *it << " "; // выводим
++it;
cout << *it << " "; // сдвинули итератор и вывели предпоследний элемент
// C# Java
```

Если `begin` и `end` имеют тип `vector<string>::iterator`, то `rbegin` и `rend` в свою очередь имеют тип `vector<string>::reverse_iterator`.

Python	C++	C	Java	C#	
	Python	C++	C	Java	C#

`begin` указывает на Python, а `end` за C#;
а вот `rbegin` – уже на C# и `rend` перед Python.

Передадим обратные итераторы в наши алгоритмы:

```
auto result = find_if(
    rbegin(langs), rend(langs),
    [](const string& lang) { // лямбда-функция по языкам
        return lang[0] == 'C'; // возвращает true, если нулевая буква = 'C'
    })
```

```
});  
// C#
```

Раньше `find_if` возвращал первый подходящий элемент, а теперь последний (первый для диапазона обратных итераторов). Вызовем `sort` от обратных итераторов и посмотрим на результат:

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
#include <map> // подключили словари  
using namespace std;  
template <typename It> // написали шаблонную функцию  
void PrintRange(It range_begin, // поменяли здесь на шаблон итератора  
               It range_end) { // и тут  
    for (auto it = range_begin; it != range_end; ++it) {  
        cout << *it << " ";  
    }  
}  
  
int main() {  
    vector<string> langs = {"Python", "C++", "C", "Java", "C#"};  
    sort(rbegin(langs), rend(langs)); // сортируем по убыванию  
    PrintRange(begin(langs), end(langs));  
    return 0;  
}  
// Python Java C++ C# C
```

`reverse` для обратных операторов работает как и для прямых (переворачивает).

В итоге обратные итераторы упрощают итерирование по контейнеру в обратную сторону и могут быть переданы в алгоритмы. А `sort(rbegin(langs), rend(langs));` – простой способ сортировки вектора по убыванию.

4.2.4. Алгоритмы, возвращающие набор элементов

Мы рассматривали алгоритм `remove_if`, который позволяет удалить элемент по какому-то критерию в массиве, в диапазоне элементов в векторе. А что если мы хотим эти элементы не удалить, а аккуратно отложить, например, в конец вектора? Для этого есть алгоритм `partition`. Я вызвал алгоритм `partition` от диапазона, который я хочу разбить на две части, и передаю

критерий, по которому я хочу разбить.

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = partition(begin(langs), end(langs), [](const string& lang) {
    return lang[0] == 'C'; // делим по принципу "начинается или не начинается на C"
});
PrintRange(begin(langs), end(langs));
// C# C++ C Java Python
```

Сначала идут все языки, которые начинаются с буквы C в каком-то порядке, как получилось: C#, C++ и C, а потом все остальные языки. То есть все элементы, которые удовлетворяют критерию, для которых лямбда-функция возвращает `true`, перемещаются в начало диапазона. Причём мы можем сохранить результат `partition` (правую границу полуинтервала тех элементов, которые удовлетворяют условию) в итератор `it`.

Если мы хотим переложить какие-то элементы из одного вектора в другой, используем алгоритм `copy_if`:

```
// исправим PrintRange, чтобы выводил через запятую
...
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
vector<string> c_langs(langs.size()); // вектор, куда мы копируем, должен быть
                                     // объявлен и иметь подходящий размер
auto it = copy_if(begin(langs), end(langs), begin(c_langs),
    [](const string& lang) {
        return lang[0] == 'C';
    });
PrintRange(begin(c_langs), end(c_langs));
// C++, C, C#, , ,
```

Заметим, что размер остался равным 5. Итераторы не могут менять размер векторов. Но `copy_if` копирует подходящие под условие элементы и возвращает в `it` итератор на новый конец вектора, в который копировали.

Теперь поговорим о `set`'ах. Что можно делать с множествами в математике? Объединять, пересекать, вычитать. Рассмотрим алгоритмы C++, которые помогают делать это же с множествами здесь.

```
// исправим PrintRange, чтобы выводил через запятую
set<int> a = {1, 8 ,3};
set<int> b = {3, 6 ,8};
```

```
vector<int> v(a.size()); // вектор для хранения результата размера как set a
auto it = set_intersection(begin(a), end(a), begin(b), end(b), begin(v));
// intersection принимает два полуинтервала и итератор, куда сохранять результат
PrintRange(begin(v), end(v));
// 3, 8, 0,
```

В итоге мы имеем пересечение множеств и 0, который дополняет до размера `a`. Сам `set_intersection` возвращает итератор за концом итогового пересечения. Т. е.:

```
PrintRange(begin(v), it);
// 3, 8,
```

Замечание: если мы не укажем явно размер вектора, в который помещаем результат, то код упадёт.

4.2.5. Итераторы `inserter` и `back_inserter`

Рассмотрим более удобный способ.

```
#include <iterator>
...
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
vector<string> c_langs; // о размере уже не беспокоимся
copy_if(begin(langs), end(langs),
        back_inserter(c_langs), // специальный итератор, вставляющий в конец
        [](const string& lang) { // привычная лямбда-функция
            return lang[0] == 'C'; // снова выделяем только начинающиеся на C
        });
PrintRange(begin(c_langs), end(c_langs));
// C++, C, C#,
```

Вектор имеет нужный размер и не содержит лишних элементов. Итераторы умеют делать лишь ограниченный набор действий: `*` (ссылка на конкретный элемент), `++`, `--` и сравнение итераторов. А итератор `back_inserter` (если с ним происходят перечисленные операции) делает `push_back` в контейнер, к которому он относится.

Аналогично поступим с алгоритмом `set_intersection`. У множества есть просто `insert`:

```
set<int> a = {1, 8, 3}
```

```
set<int> b = {3, 6, 8}
set<int> res;
auto it = set_intersection(begin(a), end(a), begin(b), end(b),
    inserter(res, end(res))); // итератор для вставки в множество
PrintRange(begin(res), end(res));
```

`inserter` возвращает специальный итератор, который вставляет элемент в множество.

4.2.6. Отличия итераторов векторов и множеств

Рассмотрим задачу: найти в векторе строк язык, который начинается с буквы 'C', и найти позицию элемента в векторе (а не итератор).

```
vector<string> langs = {"Python", "C++", "C", "Java", "C#"};
auto it = find_if(begin(langs), end(langs),
    [](const string& lang) {
        return lang[0] == 'C';
    });
cout << it - begin(langs) << endl; // таким образом получаем номер элемента в векторе
// 1
```

Для получения номера элемента достаточно из полученного с помощью `find_if` вычесть итератор начала диапазона. Получим 1. Действительно, у 'C++' в векторе номер 1. Таким образом, **итераторы вектора можно вычитать друг из друга**. Но такое не работает для итераторов множества, т. к. для них не определена операция «минус». Аналогично итераторы вектора можно сравнивать не только с помощью `==` и `!=`, но и с помощью `<` и `>`. А для множеств применимо только равно и не равно.

Попробуем для множества чисел вывести все элементы, строго большие его самого.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
++it;
PrintRange(it, end(s));
// 8, 9,
```

Но часто бывает, что `it` нельзя изменить и поэтому нельзя писать `it + 1`. Но этого можно избежать операцией `next(it)`. По сути она прибавляет 1 с помощью `++`.

```
set<int> s = {1, 6, 8, 9};
auto it = s.find(6);
PrintRange(next(it), end(s));
// 8, 9,
```

Точно так же работает функция `prev`, только вычитает 1.

4.2.7. Категории итераторов, документация

Научимся читать документацию по языку C++ с помощью сайта ru.cppreference.com. Сейчас нас интересует документация по алгоритмам. Посмотрим на `unique_copy`.

```
template <class InputIt, class OutputIt>
ForwardIt unique_copy(InputIt first, InputIt last, OutputIt d_first);
```

Видим, что это шаблонная функция, которая принимает `InputIt first`, `InputIt last`, `OutputIt d_first`. Категории итераторов в документации:

- **Input:** итераторы, из которых можно читать (итераторы любых контейнеров). Но не подходят `inserter` и `back_inserter`;
- **Forward, Bidirectional:** обычные итераторы, из которых можно читать (кроме `set` и `map`, если по `forward`-итератору что-то меняется);
- **Random:** итераторы, к которым можно прибавлять число или вычитать друг из друга (итераторы векторов и строк);
- **Output:** итераторы, в которые можно писать (итераторы векторов и строк, `inserter` и `back_inserter`).

Например, `partial_sort` принимает `random`-итераторы, потому что он сортирует (переставляет элементы) и пользуется функцией.

4.3. Очередь, дек и алгоритмы поиска

4.3.1. Стек, очередь и дек

Рассмотрим новый контейнер: **очередь**. Очередь бывает из людей или запросов. Новые приходят в конец и удаляются из начала (или наоборот). Её можно реализовать с помощью вектора:

```
v.push_back(x);    // добавляем новый
v.erase(begin(v)); // удаляем из начала вектора, что очень долго
```

Элементы вектора хранятся подряд, и поэтому удаление из начала вектора будет работать за длину вектора, потому что он будет переставлять все элементы в начало, чтобы заполнить полученную пустоту: удалили нулевой, переместили первый на нулевое место, второй на первое и т.д.

Для работы с очередью есть специальный контейнер – **deque** (double-ended queue)

- Это двусторонняя очередь;
- `#include <deque>;`
- Быстрые операции:

```
d.push_back(x)    // добавление в конец
d.pop_back(x)     // удаление из конца
d.push_front(x)   // добавление в начало
d.pop_front(x)    // удаление из начала
d[i]              // обращение к элементу по индексу
```

На коде, представленном ниже, продемонстрируем скорость работы **deque**:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    int n = 80000;
    vector<int> v(n);
    while (!v.empty()) {
        v.erase(begin(v));
    }
}
```

```
}  
cout << "Empty!" << endl;  
return 0;  
}
```

Т.е. мы сделали примерно $80000^2/2$ операций (т. к. каждый раз удаляли и перемещали).

```
#include <iostream>  
#include <deque>  
#include <algorithm>  
using namespace std;  
int main() {  
    int n = 80000;  
    deque<int> v(n);  
    while (!v.empty()) {  
        v.erase(begin(v));  
    }  
    cout << "Empty!" << endl;  
    return 0;  
}
```

Сработало моментально. И даже `pop_front` тоже сработает сразу. Дек умеет больше, но, как следствие, он менее эффективен. Если нужно работать с двумя концами, используйте дек. Но если хватает вектора, используйте вектор.

Разберём ещё одну структуру: **очередь** (`queue`).

- Если нужна только очередь, используйте `queue`;
- Основана на деке, но работает немного быстрее;
- `#include <queue>;`
- Умеет совсем немного:

```
q.push(x), q.pop(x)    // вставляем в начало и удаляем из конца  
q.front(), q.back()   // ссылки на первый и последний элементы очереди  
q.size(), q.empty()   // размер и проверка на пустоту
```

Кроме того, существует **стек** (`stack`).

- Позволяет лишь добавлять в конец и удалять из конца;
- `#include <stack>;`
- Как вектор, но умеет меньше:

```
st.push(x), st.pop(x)  // вставляем в конец и удаляем из конца
st.top()              // ссылка на последний элемент
st.size(), st.empty() // размер и проверка на пустоту
```

4.3.2. Алгоритмы поиска

Рассмотрим специальный класс методов контейнеров и алгоритмов – алгоритмы поиска. Мы с ними уже сталкивались при поиске по вектору и множеству;

- Подсчёт количества:

```
count(begin(v), end(v), x);
s.count(x);
```

- Поиск:

```
find(begin(v), end(v), x);
s.find(x);
```

Рассмотрим задачу поиска элементов в контейнерах:

1. Где будем искать?

- Неотсортированный вектор (или строка);
- Отсортированный вектор;
- Множество (или словарь).

2. Что будем искать и проверять?

- Проверить существование;
- Проверить существование и найти первое вхождение;
- Найти первый элемент, больший или равный данному;

- Найти первый элемент, больший данного;
- Подсчитать количество;
- Перебрать все.

Как осуществляется поиск в неотсортированном векторе?

- Поиск конкретного элемента:

```
find(begin(v), end(v), x)
```

- Элемент по какому-то условию (больше, меньше, больше или равен):

```
find_if(begin(v), end(v), [](int y) {...})
```

- Посчитать количество:

```
count(begin(v), end(v), x)
```

- Перебрать все можно с помощью цикла и `find`.

Например, выведем позиции всех пробелов в строке:

```
for (auto it = find(begin(s), end(s), ' ');  
     it != end(s);  
     it = find(next(it), end(s), ' ')) { // переходим в цикле к следующему пробелу  
    cout << it - begin(s) << " "; // next(it) эквивалентен it + 1  
}
```

В отсортированном векторе поиск можно осуществить быстрее с помощью [бинарного поиска](#). Количество операций равно $\log_2(N)$ – двоичному логарифму числа элементов. Столько же работает поиск во множестве и словаре.

Отсюда следствие: если вы просто хотите быстро искать по набору элементов, но не хотите добавлять новые или удалять какие-то, вам достаточно отсортированного вектора. Это будет оптимальнее, чем если вы используете множество. В отсортированном векторе можно искать так:

- Проверка на существование:

```
binary_search(begin(v), end(v), x)
```

- Первый больший или равный данному:


```
lower_bound(begin(v), end(v), x)
```

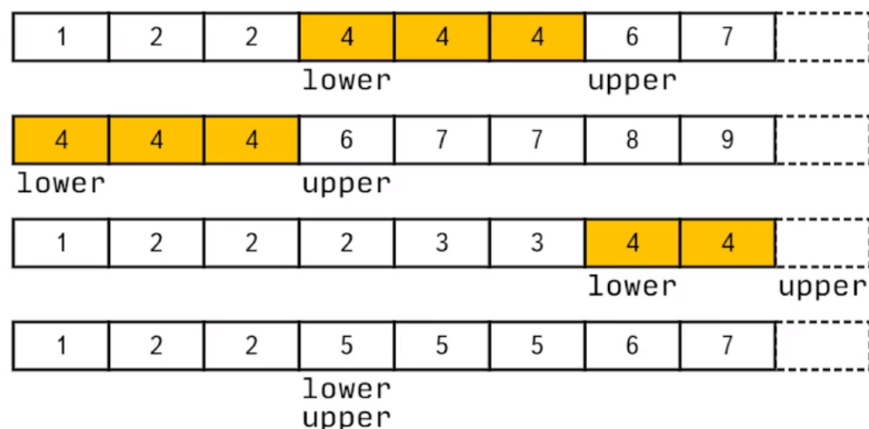
- Первый элемент, больший данного:

```
upper_bound(begin(v), end(v), x)
```

- Диапазон элементов, равных данному (аналог minmax):

```
equal_range(begin(v), end(v), x) ==  
make_pair(lower_bound(...), upper_bound(...))
```

lower_bound и upper_bound



Ещё про `equal_range`.

- Если элемент есть, то `equal_range = [lower_bound, upper_bound)` – диапазон всех вхождений;
- Если же элемента нет, то `lower_bound == upper_bound` – позиция, куда можно вставить элемент без нарушения порядка сортировки;
- Количество вхождений `== upper_bound - lower_bound`;
- А перебрать все элементы, равные данному, можно просто проитерировавшись от `lower_bound` до `upper_bound`.

Поиск во множестве мы уже знаем:

- `s.count(x);`
- `s.find(x);`
- `s.lower_bound(x);`
- `s.upper_bound(x);`
- `s.equal_range(x).`

4.3.3. Анализ распространённых ошибок

Первый пример распространённых ошибок – вычитание итераторов множества:

```
int main() {
    set<int> s = {1, 2, 7};
    end(s) - begin(s);
    return 0;
}
// no match for 'operator-'
```

Это ошибка простая, а вот если мы возьмём алгоритм, принимающий **Random** итераторы (например, `partial_sort`) и передадим ему итераторы множества:

```
int main() {
    set<int> s = {1, 2, 7};
    partial_sort(begin(s), end(s), end(s))
    return 0;
}
// no match for 'operator-', 'operator+', 'operator<'
```

Всё по той же причине. Итератор множества – не **Random** итератор, по нему нельзя сравнивать или перемещать элементы.

Теперь попробуем вызвать `remove`:

```
int main() {
    set<int> s = {1, 2, 7};
    remove(begin(s), end(s), 0);
    return 0;
}
// assignment of read-only location ...
```

Т. е. присваивание в итератор, содержимое которого нельзя менять. Ссылка под итераторами константная.

Теперь одна из самых страшных ошибок (не ловится на этапе компиляции) – передача диапазона, у которого итераторы от разных контейнеров:

```
int main() {
    vector<int> s1 = {1, 2, 7};
    vector<int> s2 = {2, 7};
    sort(begin(s1), end(s2)); // диапазон от начала одного вектора до конца другого
    return 0;
}
```

Запускаем код – и программа упала. В обратном порядке она, скорее всего, заиклится. Проверить это можно, закомментировав код, и если он заработает, проблема в нём.

Если же мы передадим итераторы разных типов, то:

```
int main() {
    vector<int> s1 = {1, 2, 7};
    vector<int> s2 = {2, 7};
    sort(begin(s1), rend(s2));
    return 0;
}
// deduced conflicting types for parameter random access iterator
```

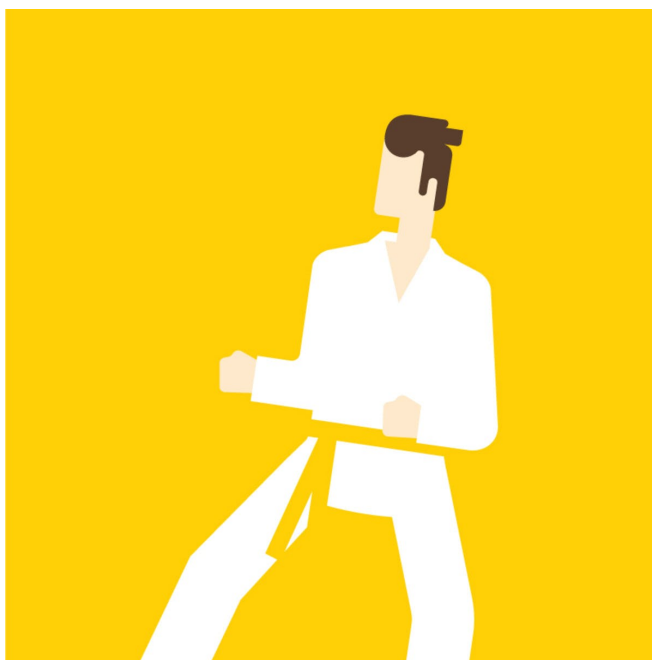
Итераторы должны иметь один тип. А у нас в данном случае тип разный и не получается вызвать функцию `sort`.



Основы разработки на C++: жёлтый пояс

Неделя 5

Наследование и полиморфизм



Оглавление

Наследование и полиморфизм	2
5.1 Наследование	2
5.1.1 Введение в наследование	2
5.1.2 Доступ к полям классов. Знакомство со списками инициализации	8
5.1.3 Порядок конструирования экземпляров классов	11
5.2 Полиморфизм	15
5.2.1 Унификация работы с классо-специфичным кодом. Постановка проблемы	15
5.2.2 Решение проблемы с помощью виртуальных методов	18
5.2.3 Свойства виртуальных методов. Абстрактные классы	21
5.2.4 Виртуальные методы и передача объектов по ссылке	23
5.2.5 Хранение объектов разных типов в контейнере с помощью <code>shared_ptr</code>	24
5.2.6 Задача о разборе арифметического выражения. Описание решения	25
5.2.7 Решение задачи частного примера	29
5.2.8 Описание и обзор общего решения задачи	30

Наследование и полиморфизм

5.1. Наследование

5.1.1. Введение в наследование

Наследование в языке C++ – это способ выразить связи между классами. Например, сказать, что один класс является подтипом другого класса, реализуя часть его функциональности, и может использоваться в связке вместе с ним. Посмотрим, как всё это работает на практическом примере. Мы хотим написать некий волшебный мир, в котором есть животные (кошки и собаки), которые могут есть фрукты – яблоки и апельсины, – а у фруктов есть некоторое количество здоровья. Значит, мы тогда реализуем все эти действия с помощью наших классов, пусть у нас кошки и собаки будут иметь метод «покушать», и когда они едят, будем выводить в консоль, что произошло.

```
#include <iostream>
using namespace std;
struct Apple { // структура Яблоко
    int health = 10;
};
struct Orange { // структура Апельсин
    int health = 5;
};
class Cat { // создаём класс Кошка
public:
    void Meow() const { // кошка может мяукать
        cout << "meow! ";
    }
    void Eat(const Apple& a) { // кошка может есть яблоко
        cout << "Cat eats apple. " << a.health << "hp." << endl;
    }
};
```

```
int main() {
    Cat c; // создали кошку
    c.Meow(); // помяукали
    Apple a; // создали яблоко
    c.Eat(a); // съели яблоко кошкой
    return 0;
}
// meow!
// Cat eats apple. 10hp.
```

Всё, кошка съела яблоко, в котором было 10 единиц здоровья.

А теперь попробуем скормить кошке апельсин. По идее она должна уметь есть фрукты, и апельсины тоже.

```
...
    Orange o; // создали апельсин
    c.Eat(o); // съели апельсин кошкой
    return 0;
}
// error: no matching function for call to 'Cat::Eat(Orange&)'
```

Появляется ошибка: нет метода «поесть» для апельсина. Определим его, продублировав метод яблока:

```
...// сразу под Meow() {}
void Eat(const Apple& a) { // кошка может есть яблоко
    cout << "Cat eats apple. " << a.health << "hp." << endl;
}
void Eat(const Orange& o) { // кошка может есть апельсин
    cout << "Cat eats orange. " << o.health << "hp." << endl;
} // продублировали всё, заменив apple на orange
// meow!
// Cat eats orange. 5hp.
```

Всё, апельсин тоже съели. Всё бы ничего, но теперь добавим собаку, скопировав класс Cat:

```
class Dog { // создаём класс Собака дублированием класса Cat с исправлениями
public: // удалили мяуканье
    void Eat(const Apple& a) {
        cout << "Dog eats apple. " << a.health << "hp." << endl;
```

```

}
void Eat(const Orange& o) {
    cout << "Dog eats orange. " << o.health << "hp." << endl;
} // продублировали все, заменив apple на orange
};
...
int main() {
    Cat c; // создали кошку
    c.Meow();
    Apple a;
    Orange o;
    c.Eat(o);
    Dog d;
    d.Eat(a); // съели яблоко кошкой
    return 0;
}
// meow!
// Cat eats orange. 5hp.
// Dog eats apple. 10hp.

```

Всё работает. Но мы 4 раза продублировали функцию `Eat()`. Заметим, что объекты `Apple` и `Orange` отличаются только числом `hp`, но с ними одинаково работает функция `Eat()`. Если же нам придётся добавить ещё фрукты, то мы будем вынуждены копировать код для этого фрукта и в кошке, и в собаке. Это ужасно.

Создадим **отношение наследования** для классов яблоко и апельсин. Ведь они являются наследниками **базового класса** фрукт (`Fruit`). Яблоко и апельсин – наследники фрукта, и мы явно зададим публичное наследование:

```

struct Fruit { // структура фрукты
    int health = 0; // поле здоровья, пусть по умолчанию будет 0 hp
    string type = "fruit"; // добавляем тип того, что мы едим
};

struct Apple : public Fruit { // указываем, что яблоко наследуется от фруктов
    Apple() { // нам достаточно написать конструктор для яблока
        health = 10; // яблоко видит поле health у фрукта, поэтому присваиваем 10
        type = "apple"; // явно указываем, что яблоко имеет тип яблока
    } // т. к. наследование публичное, мы видим все поля Fruit в Apple
};

```



```
struct Orange : public Fruit { // без отношения наследования : public Fruit
    Orange() { // наш код не заработает
        health = 5;
        type = "orange";
    }
};
//
```

Таким образом с помощью публичного наследования мы получили доступ к полям, объявленным в нашем базовом классе. Теперь мы можем исправить наш код следующим образом:

```
class Cat { // создаём класс Кошка
public:
    void Meow() const {
        cout << "meow! ";
    }
    void Eat(const Fruit& f) { // кошка ест фрукты
        cout << "Cat eats " << f.type << ". " << f.health << "hp.";
    }
};
...
int main() {
    Cat c;
    c.Meow();
    Orange o;
    c.Eat(o);
    return 0;
}
// meow! Cat eats orange. 5 hp.
```

Теперь для кошки есть всего один метод `Eat()` для всех фруктов. Аналогично делаем для собаки:

```
class Dog {
public:
    void Eat(const Fruit& f) { // собака ест фрукты
        cout << "Dog eats " << f.type << ". " << f.health << "hp. ";
    }
};
...
int main() {
    Dog d;
```

```

Orange o;
d.Eat(o);
return 0;
}
// Dog eats orange. 5hp.

```

Вот, мы уменьшили дублирование для фруктов. Добавим, например, ананас:

```

struct PineApple : public Fruit {
    PineApple() {
        health = 15;
        type = "papple";
    }
};
...
int main() {
    Dog d;
    PineApple p;
    d.Eat(p); // теперь собака съест ананас
    return 0;
}
//Dog eats papple. 15hp.

```

Аналогичным образом уберём дублирование из собаки и кошки. На самом деле они должны наследоваться от базового класса Животные. Каждое животное умеет есть фрукты, но не каждое животное может мяукать (это умеет только кошка).

```

class Animal {
public:
    void Eat(const Fruit& f) { // животное типа type ест фрукты
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    string type = "animal"; // уберём дублирование в методе Eat
};
class Cat : public Animal { // создаём отношение наследования
public: // т. к. наследование публичное, Cat видит переменную type от Animal
    Cat() {
        type = "cat";
    }
    void Meow() const {
        cout << "meow! ";
    }
};

```

```
    } // удалили метод Eat() у кошки. Она животное, а животные умеют есть
};
```

С помощью публичного наследования мы получили доступ к методам и полям класса «животное». И поэтому теперь кошка может вызывать метод `Eat`, хотя этот метод не определён внутри «кошки», а определён внутри класса, от которого отнаследован класс «кошка». И теперь точно так же мы уберем дублирование для класса «собака».

```
class Dog : public Animal {
public:
    Dog() {
        type = "dog";
    }
};
```

Всё продолжает работать. Но теперь и собака и кошка могут есть, хотя в их описании это явно не указано. Кроме того, кошка может мяукать, а собака – нет.

Чтобы показать всю силу наследования, давайте создадим функцию «покормить», `DoMeal`. В неё мы будем передавать некое животное, некоторый фрукт, при этом сама функция не будет знать, что это за конкретное животное и что это за конкретный фрукт. Она просто вызовет метод «поесть» для этого фрукта. И, по идее, вывод в консоль измениться не должен.

```
...
void DoMeal(Animal& a, Fruit& f) {
    a.Eat(f);
}
int main() {
    Dog d;
    Cat c;
    Orange o;
    Apple a;
    DoMeal(d, a); // эта функция ничего не знает ни о собаках, ни о кошках
    DoMeal(c, o); // она знает только о классах базового типа
    return 0;
}
// dog eats apple. 10hp.
// cat eats orange. 5hp.
```

Замечу, что сейчас в классе `Animal` у нас строчка типа `type` доступна всем. То есть снаружи эту строчку тоже можно менять, когда вы используете данный класс. И, на самом деле, в какой-

то момент по ошибке вы можете внести какие-то изменения, которые вносить не хотелось бы. Например, после того как мы покормим собаку, добавлять к ней звёздочку.

```
void DoMeal(Animal& a, Fruit& f) {
    a.Eat(f);
    a.type += "*";
}...
DoMeal(d, a);
DoMeal(d, o);
DoMeal(d, o);
// dog eats apple. 10hp.
// dog* eats apple. 10hp.
// dog** eats apple. 10hp.
```

О том, как этого избежать, в следующем видео.

5.1.2. Доступ к полям классов. Знакомство со списками инициализации

Зачем нужно наследование?

- Введение логической иерархии между классами ($Cat, Dog \in Animal$);
- Решение проблемы дублирования между однотипными структурами;
- Универсальные функции, работающие с базовыми классами-родителями (`DoMeal`).

Отношение публичного наследования объявляется так:

```
class TDerived : public TBase {...}
```

При таком публичном наследовании у класса-наследника появится в том числе доступ к полям и методам базового класса, от которого он унаследован.

Перейдём к проблеме непреднамеренной модификации полей классов в наших программах. Напомню проблему: у нас была функция «покормить», в неё мы добавляли звёздочку к типу, в результате двух кормёжек переменная `type` внутри объекта класса «собаки» менялась. Рассмотрим, как от этого можно защититься.

```
void DoMeal(const Animal& a, Fruit& f) {  
    a.Eat(f); // передаём константный тип, но тогда мы не сможем его изменять  
    a.type += "*";  
} // этот способ не используем из-за его локальности
```

Способ 1: можно сделать переменную в базовом классе защищённой (написав ключевое слово `protected`). В данном случае, когда мы пишем ключевое слово `protected`, никто извне не может обратиться к переменной типа `type`, в данном случае, у класса `animal`, у объектов класса `animal` в рантайме. Но при этом к объекту типа `type` имеют доступ наследники класса `animal`, то есть, в данном случае, когда мы будем создавать экземпляр класса «кошки», когда мы будем создавать этот объект в его конструкторе, будет доступ в переменную `type`.

Скомпилируем:

```
class Animal {  
public:  
    void Eat(const Fruit& f) { // животное типа type ест фрукты  
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";  
    }  
protected: // все переменные и методы ниже будут защищены (не видны снаружи)  
    string type = "animal"; // уберём дублирование в методе Eat  
};
```

И теперь компилятор будет жаловаться, пока мы не уберём изменение поля `type` в `DoMeal`. Заметим, что наследники могут обращаться к полям и менять их, но уже в себе.

Способ 2: объявление поля константным внутри базового класса. Это тоже хорошее, рабочее решение. Но далее мы видим, что где-то почему-то убирается константность. То есть читаем ошибку, видим, что в данном случае мы пытаемся записать новое значение в константную переменную. Но мы ведь сейчас находимся в конструкторе. Почему так происходит? Здесь надо сказать одну очень важную вещь: в языке C++ все объекты всегда имеют две стадии: либо объект сейчас создается, вот в данный текущий момент происходит его создание, он только-только появляется в памяти, его поля инициализируются, но он ещё до конца не создан. А вторая стадия – это момент, когда уже объект создан, то есть под него выделена память, там всё проинициализировано, как надо, и мы с ним работаем.

```
class Animal {  
public:  
    void Eat(const Fruit& f) {  
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";  
    }  
}
```

```
    const string type = "animal";
};
class Cat : public Animal {
public:
    Cat() { // на этом этапе переменные внутри класса уже проинициализированы
        type = "cat"; // тут ошибка из-за константности поля
    }
    void Meow() const {
        cout << "meow! ";
    }
};
... // прокомментируем все последующие обращения к type и увидим:
// animal eats apple. 10hp.
```

Видим, что изначально `type` в `Cat` инициализируется значением `animal`, а мы потом пытаемся его переопределить на `Cat`. Создадим конструктор класса животное:

```
class Animal {
public:
    Animal(const string& t = "animal") {
        type = t; // опять пытаемся в константную строку что-то записывать
    }
    void Eat(const Fruit& f) {
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    const string type;
};
// passing 'const string'....
```

Видим, что `type` проинициализирована ещё до фигурных скобок. Воспользуемся синтаксисом списков инициализации:

```
class Animal {
public:
    Animal(const string& t = "animal")
    : type(t) { // хотим проинициализировать type значением t
    }
    void Eat(const Fruit& f) {
        cout << type << " eats " << f.type << ". " << f.health << "hp. ";
    }
    const string type;
};
```

```
};  
// animal eats apple. 10hp.
```

Мы смогли проинициализировать переменную `type` нужным нам значением `t` с помощью синтаксиса списков инициализации. Теперь нам надо из класса-наследника как-то передать свой собственный тип. Для этого нам надо вызвать конструктор базового класса, в данном случае `animal`, с другим аргументом.

```
class Cat : public Animal {  
public:  
    Cat() : Animal("cat") { // передаём в Animal нужное нам значение  
    } // которое запишется в константный type  
    void Meow() const {  
        cout << "meow! ";  
    }  
}; // далее при любой попытке модификации type, компилятор нам не позволит это сделать  
// cat eats apple. 10hp.
```

Таким образом, мы можем защитить поля классов двумя способами:

1. Объявить поле в секции `protected`. Тогда к нему доступ будут иметь только наследники;
2. Объявить поле константным. Его нужно будет проинициализировать при создании объекта и его нельзя будет менять на протяжении жизни.

Синтаксис списков инициализации выглядит так:

```
MyClass(int v1, int v2) // в конструкторе класса после объявления аргументов  
: Var1(v1) // пишем инициализируемые поля классов и значения, которые запишем  
  , Var2(v2)  
{  
    ...  
}
```

5.1.3. Порядок конструирования экземпляров классов

Будем работать с упрощённой версией кода:

```
#include <iostream>

using namespace std;

struct Log { // структура данных логер
    Log(string name) : n(name) {
        cout << "+ " << n << endl; // пишет + и свой id, когда создается
    };
    ~Log() { // деструктор
        cout << "- " << n << endl; // пишет - и свой id, когда удаляется
    }
    string n;
};

struct Fruit {
    string type = "fruit";
    Log l = Log("Fruit"); // заведём логер для фруктов
};

struct Apple : public Fruit {
    Apple() {
        type = "apple";
    }
    Log l = Log("Apple"); // теперь логер для яблока
};

int main() {
    Apple a;
    return 0;
}

// + Fruit
// + Apple
// - Apple
// - Fruit
```

Видим, что логер был создан сначала внутри `Fruit`, затем внутри яблока, затем удалён внутри яблока и только потом внутри `Fruit`. Это может говорить о том, что если у нас есть некоторая иерархия классов, то при создании объекта, принадлежащего данной иерархии, всегда будет сначала вызываться конструктор базового класса, затем класса-наследника и так далее по це-

почке. При этом, когда объекты данных классов будут удаляться, сначала всегда вызывается объект самого последнего класса наследника и так далее по цепочке вверх, пока мы не дойдем до деструктора самого базового класса.

Теперь зарефакторим наш код:

```
struct Fruit {
    Fruit(const string& t)
        : l(t + " (Fruit)") {} // инициализируем логер строкой + именем базового класса
    const Log l; // сделали логер константным
};
// т. к. яблоко унаследовано от Fruit, то перед Apple вызывается конструктор Fruit
struct Apple : public Fruit {
    Apple() : Fruit("apple") {
    }
    Log l = Log("Apple"); // теперь логер для яблока
};
// + apple (Fruit)
// + Apple
// - Apple
// - apple (Fruit)
```

Теперь сделаем яблокам уникальные идентификаторы:

```
struct Apple : public Fruit {
    Apple(const string& t) : Fruit(t), l(t) { // добавили идентификатор и логер
    }
    const Log l;
};

int main() {
    Apple a1 = Apple("a1");
    Apple a2 = Apple("a2");
    return 0;
}
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
```

Значит, сначала создали объект **a1**, потом объект **a2**. Когда для них начали вызываться деструкторы, произошло все ровно в обратном порядке. Значит, когда мы создаём переменные обычные, конструирование и удаление объектов работает точно так же. **Кто создаётся рань-**

ше, тот удаляется позже. Создадим более сложный класс Яблочного дерева, который будет содержать в себе много яблок:

```
class AppleTree {
public:
    AppleTree()
        : a1("a1")
        , a2("a2") {
    }
    Log l = Log("AppleTree");
    Apple a1;
    Apple a2;
}

int main() {
    AppleTree at;
    return 0;
}

// + AppleTree
// + a1 (Fruit) // + a1 // + a2 (Fruit) // + a2
// - a2 // - a2 (Fruit) // - a1 // - a1 (Fruit)
// - AppleTrees
```

Вопрос: от чего зависит порядок инициализации? Даже поменяв строчки местами:

```
: a2("a2")
, a1("a1") { // выскочит пара warning-ов
```

Всё равно порядок создания не изменится. Список инициализации не меняет порядок создания. Он просто указывает значения, которыми мы их инициализируем. Реально же порядок зависит только от того, как они перечислены внутри класса.

Хотя это всего лишь предупреждение, инициализация не в том порядке может нам помешать:

```
class AppleTree {
public:
    AppleTree(const string& t)
        : type(t) // инициализируем переменную type первой
        , a1(type + "a1") // и тут её активно используем
        , a2(type + "a2") {
    }
}
```

```
Apple a1;
Apple a2;
string type; // а здесь эта переменная объявлена последней
};

int main() {
    AppleTree at("AppleTree");
    return 0;
}
// terminate called after throwing an instance of std::bad_alloc
```

Видим ошибки или какой-то бред. Поскольку мы инициализировали `type` последним и использовали его перед этим, мы обратились к неизвестной памяти. Если изменить порядок, то всё будет работать.

5.2. Полиморфизм

5.2.1. Унификация работы с классо-специфичным кодом. Постановка проблемы

Наследование нам помогало с дедупликацией повторяющегося кода в различных классах. Соответственно, дедупликацию мы исключали за счёт создания базового класса и вынесения общего кода в методы этого класса. Восстановим код из лекции:

```
#include <iostream>
using namespace std;

class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
private:
    const string type_;
};
```

```
class Cat : public Animal {
public:
    Cat() : Animal("cat") {}
    void Meow() const {
        cout << "Meow!" << endl;
    }
};

class Dog : public Animal {
public:
    Dog() : Animal("dog") {}
    void Bark() const {
        cout << "Whaf!" << endl;
    }
};

int main() {
    Cat c;
    Dog d;
    c.Eat("apple");
    d.Eat("orange");
    return 0;
}
// cat eats apple
// dog eats orange
```

Мы научились делать общий код для разных классов. Давайте теперь рассмотрим не общий код, а специфичный для каждого класса. Давайте напомним функции, которые позволяют нам единообразно просить животных издавать разные звуки. Соответственно, самый простой вариант – это написать отдельную функцию для кошки и для собаки:

```
void MakeSound(const Cat& c) {
    c.Meow();
}

void MakeSound(const Dog& d) {
    d.Bark();
}

int main() {
    ...
    MakeSound(c);
    MakeSound(d);
}
```

```
}  
// Meow!  
// Whaf!
```

Снова возникает проблема:

- Если мы имеем X различных классов: Cat, Dog, Horse, ...
- И у нас Y методов для каждого класса: голос, действие, ...
- Получим в итоге $X * Y$ функций!

Что же можно сделать? Первая же идея, которая приходит нам в голову – это воспользоваться методикой из предыдущей лекции. Давайте мы объединим функцию `MakeSound` и унесём её в базовый класс.

```
class Animal {  
public:  
    Animal(const string& type) : type_(type) {}  
    void Eat(const string& fruit) {  
        cout << type_ << " eats " << fruit << endl;  
    }  
    void Voice() const { // приходится делать громоздкую конструкцию,  
        if (type_ == "cat") { // поскольку animal не знает про котов  
            cout << "Meow!" << endl;  
        } else if (type_ == "dog") {  
            cout << "Whaf!" << endl;  
        }  
    }  
private:  
    const string type_;  
}; // удаляем функции Bark и Meow  
...  
void MakeSound(const Animal& a) {  
    a.Voice();  
}  
  
int main() { ...  
    MakeSound(c);  
    MakeSound(d);  
}
```

Вроде исправили, но остались минусы, и основной заключается в том, что различное поведение классов-потомков необходимо переносить в базовый класс. Дополним класс животных, добавив попугая:

```
...
} else if (type_ == "dog") {
    cout << "Whaf!" << endl;
} else if (type == "parrot") { // попугай называет себя (по имени) хорошим
    cout << name_ << " is good!" << endl; // попытаемся сделать это тут
}
...
class Parrot : public Animal {
public:
    Parrot(const string& name) : Animal("parrot"), name_(name) {}
private:
    const string& name_; // имя попугая делаем приватным
};
```

Ловим ошибку: в базовом классе неизвестно приватное поле `name`. Таким образом мы не можем перенести функцию `Voice` из класса `Parrot` в родительский класс `Animal`.

5.2.2. Решение проблемы с помощью виртуальных методов

У нас не получилось объединить различные специфичные методы разных классов в рамках одного метода в базовом классе (мы не смогли использовать приватное поле одного из классов).

Вынесем логику определённых классов обратно в эти классы:

```
#include <iostream>
using namespace std;

class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
private:
    const string type_;
```

```
};

class Cat : public Animal {
public:
    Cat() : Animal("cat") {}
    void Voice() const {
        cout << "Meow!" << endl;
    }
};

class Dog : public Animal {
public:
    Dog() : Animal("dog") {}
    void Voice() const {
        cout << "Whaf!" << endl;
    }
};

class Parrot : public Animal {
public:
    Parrot(const string& name) : Animal("parrot"), name_(name) {}
    void Voice() const {
        cout << name_ << " is good!" << endl;
    }
private:
    const string name_;
};

void MakeSound(const Animal& a) {
    a.Voice();
}

int main() {
    Cat c;
    Dog d;
    MakeSound(c);
    MakeSound(d);
    return 0;
}
// error: 'const class Animal' has no member named 'Voice'
```

Для избавления от ошибки просто объявим этот метод в `animal`, но там он ничего не будет делать:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    void Voice() const {}; // вот тут
private:
    const string type_;
};
//
```

Всё корректно скомпилировалось, но никто не издаёт звуков. Нам нужно, чтобы базовый класс узнал о методах в классах-потомках. Это делается добавлением ключевого слова **virtual** к методу в базовом классе.

```
...
virtual void Voice() const {}; // в Animal
// Meow!
// Whaf!
```

То есть мы сказали базовому классу, что в классах-потомках может быть своя реализация метода `Voice` с той же сигнатурой (возвращаемый тип и константность должны совпадать). И теперь, когда мы вызываем метод `Voice()` у базового класса, то будет вызываться этот метод у нужного класса-потомка.

Теперь добавим попугая:

```
int main() {
    Parrot p("Kesha");
    MakeSound(p);
    return 0;
}
// Kesha is good
```

Всё работает и для попугая. Теперь допустим, что мы захотели изменить название `Voice()` в `Animal` на `Sound()`.

```
virtual void Sound() const {}; // в Animal
```



```
// error: 'const class Animal' has no member named 'Voice'
```

Ошибка возникает в функции `MakeSound()`. Поменяем название и здесь:

```
void MakeSound(const Animal& a) {  
    a.Sound();  
}  
//
```

Программа собирается, но ничего не выводит. Это происходит, потому что в потомках остался метод `Voice()`, но базовый класс ничего про него не знает.

Вернём обратно название `Voice()` везде. И в каждом потомке напишем:

```
void Voice() const override {  
    ...  
}  
...
```

Всё снова работает, как мы ожидаем. Если же мы сейчас повторим те же действия по переименованию в `Sound()` в базовом классе и в `MakeSound()`. Теперь нам выдаётся ошибка:

```
// ... marked 'override', but does not override
```

Компилятор подсказывает, что в базовом классе метод `Voice` неизвестен, и в этом случае мы не совершим ошибки по случайному переименованию функций.

Таким образом, мы узнали про два новых ключевых слова:

- **virtual** добавляется к методам в базовом классе. Позволяет вызывать методы производных классов через ссылку на базовый класс;
- **override** добавляется к методам в производных классах (классах-потомках). Требует объявления метода в базовом классе с такой же сигнатурой.

5.2.3. Свойства виртуальных методов. Абстрактные классы

Методы, помеченные словом **virtual**, называются **виртуальными**. Рассмотрим их свойства. Добавим к получившемуся после наших изменений коду класс Лошадь, в котором не будем

объявлять метод `Sound()`:

```
class Horse : public Animal {
public:
    Horse() : Animal("horse") {}
};
...
int main() { ...
    Horse h;
    MakeSound(h);
    return 0;
}
//
```

Для лошади ничего не вывелось. Посмотрим, какой метод `Sound()` вызывается для лошади. Изменим в `Animal` метод `Sound`:

```
virtual void Sound() const {
    cout << type_ << " is silent " << endl; // по умолчанию будет говорить так
}
// horse is silent
```

Таким образом, в данном отношении виртуальные методы не отличаются от обычных: если метод не объявлен в классе-потомке, то он будет вызван из базового класса.

Рассмотрим другое полезное свойство виртуальных методов: допустим, мы хотим, чтобы у каждого потомка был по-своему реализован метод `Sound()`. Пока под это условие не подходит `Horse`. Будем считать, что мы просто забыли его реализовать и хотим, чтобы компилятор подсказывал нам, что нам надо реализовать. Потребуем обязательной реализации в классах-потомках:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    virtual void Sound() const = 0; // это не присваивание, а формальный синтаксис
private:
    const string type_;
};
// cannot declare variable 'h' to be of abstract type 'Horse'
```

Мы сказали виртуальному методу `Sound()`, что он является абстрактным (чисто виртуальным), т.е. требует обязательной реализации в каждом классе-потомке. Класс `Horse` считается абстрактным, потому что в нем нет реализации абстрактного метода `Sound()`, и мы не можем создавать объекты абстрактных классов.

5.2.4. Виртуальные методы и передача объектов по ссылке

Заметим, что мы передавали объекты классов-потомков по ссылке на базовый класс.

Попробуем передать по значению:

```
void MakeSound(const Animal a) { // теперь без &
    a.Voice();
}
```

Программа не собралась, потому что компилятор считает, что мы хотим преобразовать этот объект в тип базового класса. Но мы не можем создавать объекты базового класса, потому что он является абстрактным (из-за виртуального метода `Voice()` в базовом классе). Уберём абстрактность метода `Voice()`:

```
class Animal {
public:
    Animal(const string& type) : type_(type) {}
    void Eat(const string& fruit) {
        cout << type_ << " eats " << fruit << endl;
    }
    virtual void Voice() const {
        cout << type_ << " is silent" << endl;
    }
    ...
// cat is silent
```

Видим, что при передаче по значению вызываются методы базового класса, а не класса-потомка. При передаче по значению мы теряем всю информацию о классе-потомке, и у нас сохраняется информация только о базовом классе, в котором `Voice()` печатает `is silent`. То есть при передаче по значению мы теряем преимущества виртуальных методов.

5.2.5. Хранение объектов разных типов в контейнере с помощью `shared_ptr`

Вернёмся к функции `main()`:

```
int main() {
    Cat c;
    Dog d;
    Parrot p("Kesha");
    Horse h;

    MakeSound(c);
    MakeSound(d);
    MakeSound(p); // 4 очень похожих вызова
    MakeSound(h);
    return 0;
}
```

Хочется сделать вызовы универсальными с помощью контейнеров. Мы могли бы сложить всех животных в контейнер и, например, в цикле `for` пробегаться по ним. Все объекты мы можем объединить ссылкой на базовый класс:

```
#include <memory> // заголовок с shared_ptr
...
int main() {
    // vector<Animals> animals; вектор из ссылок создать не получится
    // но есть другой тип из базовых библиотек, с помощью которого можно это сделать:
    shared_ptr<Animal> a;

    a = make_shared<Cat>(); // в угловых скобках указываем реальный тип
    a->Sound(); // похожим на итераторы образом вызываем метод Sound()
    return 0;
}
// Meow!
```

Видим, что вызвался метод `Sound` для `Cat`.

Самое полезное свойство `shared_ptr`, которое мы использовали сейчас: в качестве типа объекта, в который оборачивается `shared_ptr` можем указать просто `Animal`, и при этом `shared_ptr` будет вести себя как ссылка (мы сложим в него объекты производных классов и сможем обращаться к интерфейсу базового класса). Второе свойство: у `shared_ptr` интерфейс, похожий на итераторы (с ними просто работать).

```
int main() {
    shared_ptr<Animal> a;
    a = make_shared<Cat>();
    vector<shared_ptr<Animal>> animals; // векторы животных из shared_ptr
    for (auto a : animals) { // передаем по значению, а можно и const auto& a
        MakeSound(*a); // получаем объект, который обернут в shared_ptr
    }
    return 0;
}
// Meow!
```

Всё работает, теперь сделаем то же для вектора животных.

```
int main() {
    shared_ptr<Animal> a;
    a = make_shared<Cat>();
    vector<shared_ptr<Animal>> animals = {
        make_shared<Cat>(),
        make_shared<Dog>(),
        make_shared<Parrot>("Keshha"), // имя передаётся в конструктор попугая
    }
    for (auto a : animals) {
        MakeSound(*a);
    }
    return 0;
}
// Meow! Whaf! Keshha is good!
```

С помощью `shared_ptr` мы можем помещать различные типы объектов производных классов в контейнеры и обходить их с помощью циклов, как и обычные типы.

5.2.6. Задача о разборе арифметического выражения. Описание решения

Поставим задачу написать парсер арифметических выражений:

- Входные данные: Выражение, состоящее из операций `+`, `-`, `*`, цифр и переменной `x`. Её значение нам дано;

- Выходные данные: значение выражения при введённом значении переменной.

Пример работы:

```
// Enter expression:
// 1+2-3*x+x
// Enter x:
// 5
// Expression value -7. Enter x:
// 1
// Expression value 1...
```

Теперь попробуем написать наш разбиратель:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>

using namespace std;

int main() {
    string tokens;
    cout << "Enter expression: ";
    cin >> tokens; // запрашиваем у пользователя выражение
    int x = 0;
    auto expr = Parse(tokens, x); // функция разбора выражения для всех x
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << "Expression value: " << expr->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}
```

Теперь реализуем функции Parse и Evaluate. Мы пока не знаем ничего о типе переменной expr.

```
struct Node { // тип выводимых значений
public:
    int Evaluate() {
        return 0;
    }
};
```

```

    }
};
Node Parse(const string& tokens, int& x) // должна возвращать объект типа Node
    // у которого есть метод Evaluate, вычисляющий выражение

```

Чтобы понять, как разбирать арифметическое выражение, сделаем это вручную на примере:

```

int main() {
    string tokens;
    string tokens = "5+7-x*x+x"; //  $-x^2 + x + 12$ 
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << "Expression value: " << expr->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}

```

Для простоты разбора будем использовать только числа из одной цифры, а также не будет скобок, пробелов и ненужных символов. И будем считать, что выражение всегда корректно. Разделим выражение на типы ввода:

```

struct Node { // тип выводимых значений -- это базовый класс для переменных и цифр
public:
    virtual int Evaluate() {
        return 0;
    } // сделали метод виртуальным
}; // т. к. для каждого класса-потомка надо описывать, что делать в Evaluate

class Digit : public Node {
public: // класс цифра, который будет всегда возвращать значение самой цифры
    Digit(int d) : d_(d) {
    }
    int Evaluate() override {
        return d_;
    }
private:
    const int d_;
}

class Variable : public Node { // переменная x

```

```

public:
    Variable(const int& x) : x_(x) {
    } // сохраняем константную ссылку
    int Evaluate() override { // при вычислении x возвращаем его значение
        return x_;
    }
private:
    const int& x_;
} // остались операции. Заметим, что у них разный приоритет. Умножение раньше всего

class Operation { // для сохранения операции надо сохранять левый и правый операнды
public: // передаём символ операции, левое и правое слагаемое
    Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
        op_(op), left_(left), right_(right) {
    }
    int Evaluate() {
        // вычисление операций оставим на потом
    }
private:
    const char op_;
    shared_ptr<Node> left_, right_;
};

```

Заметим, что в выражении $5 + 7 * 3$ для операции $+$ левое слагаемое – это число 5, а правое – это результат умножения 7 и 3. При вычислении значения операции нужно сначала вычислить оба слагаемых, а потом произвести саму операцию и вернуть значение.

```

class Operation : public Node { // унаследуем от Node
public:
    Operation(char op, shared_ptr<Node> left, shared_ptr<Node> right) :
        op_(op), left_(left), right_(right) {
    }
    int Evaluate() override { // операция унаследована от Node
        if (_op == '*') {
            return _left->Evaluate() * _right->Evaluate();
        } else if (_op == '+') {
            return _left->Evaluate() + _right->Evaluate();
        } else if (_op == '-') {
            return _left->Evaluate() - _right->Evaluate();
        }
        return 0; // если операция не распознана, но для корректной строки не дойдём
    }
};

```



```
}
...
```

5.2.7. Решение задачи частного примера

Вспомним, как у нас вообще вычисляются арифметические выражения. Во-первых, они вычисляются слева направо для операций с равным приоритетом. Операции с большим приоритетом вычисляются в первую очередь. То есть для нашего выражения сначала мы должны вычислить операцию $x * x$. Затем у нас остаются три равноправные операции: это сложение, вычитание и сложение. (в выражении $5 + 7 - x * x + x$). И мы должны их вычислить слева направо. Опишем множители:

```
int main() {
    string tokens = "5+7-x*x+x";
    shared_ptr<Node> var1 = make_shared<Variable>(x); // делаем операцию умножения
    shared_ptr<Node> var2 = make_shared<Variable>(x);
    shared_ptr<Node> mul1 = make_shared<Operation>('*', var1, var2);
    shared_ptr<Node> dig1 = make_shared<Digit>(5); // делаем операцию сложения
    shared_ptr<Node> plus1 = make_shared<Operation>('+', dig1, dig2);
    shared_ptr<Node> dig2 = make_shared<Digit>(7);
    shared_ptr<Node> var3 = make_shared<Variable>(x);
    shared_ptr<Node> minus1 = make_shared<Operation>('-', plus1, mul1);
    shared_ptr<Node> plus2 = make_shared<Operation>('+', minus1, var3);
    // итоговый результат будет в самом правом и самом неприоритетном операторе, plus2
    cout << "Enter x: ";
    while (cin >> x) { // пока пользователь будет вводить значения, вычисляем
        cout << plus2->Evaluate() << endl;
        cout << "Enter x: ";
    }
    return 0;
}
```

При значении $x = 1$ выведет 12. Это правда. При $x = 5$ получим: -8 и т. д.

5.2.8. Описание и обзор общего решения задачи

Решим исходную задачу: выражение мы должны, на самом деле, вводить также с консоли — оно должно быть произвольное — с точностью до тех условий, которые мы указали ранее. И мы должны его уметь преобразовывать в нужный нам формат не вручную, а с помощью алгоритма, который бы эффективно по нему проходил и сам составлял необходимый набор объектов, переменных, операций.

Возьмем заранее заготовленный код (находится в материалах к видеолекции) и разберём его:

```
#include <iostream>
#include <memory>
#include <stack>
#include <vector>
using namespace std;
// структура классов у нас остаётся той же
struct Node { // базовый класс
    virtual int Evaluate() const = 0;
};

struct Value : public Node { // класс для значения. Ранее был Digit
    Value(char digit) : _value(digit - '0') {
    }
    int Evaluate() const override {
        return _value;
    }
private:
    const uint8_t _value;
};

struct Variable : public Node { // класс для переменной x
    Variable(const int &x) : _x(x) {
    }
    int Evaluate() const override {
        return _x;
    }
private:
    const int &_x;
};

// алгоритм называется shunting-yard, алгоритм сортировочных станций.
struct Op : public Node { // класс для операций
```

```

Op(char value)
: precedence([value] {
    if (value == '*') {
        return 2;
    } else {
        return 1;
    }
}()),
_op(value) {
}

const uint8_t precedence;
int Evaluate() const override {
    if (_op == '*') {
        return _left->Evaluate() * _right->Evaluate();
    } else if (_op == '+') {
        return _left->Evaluate() + _right->Evaluate();
    } else if (_op == '-') {
        return _left->Evaluate() - _right->Evaluate();
    }
    return 0;
}

// передаём левый и правый операнды отдельными методами. удобнее для произвольного
void SetLeft(shared_ptr<Node> node) {
    _left = node;
}

void SetRight(shared_ptr<Node> node) {
    _right = node;
}

private:
    const char _op;
    shared_ptr<const Node> _left, _right;
};

template <class Iterator>
shared_ptr<Node> Parse(Iterator token, Iterator end, const int &x) {
    // функцию Parse сделали шаблонной, т. е. для любых контейнеров из символов
    if (token == end) {
        return make_shared<Value>('0');
    }
    stack<shared_ptr<Node>> values;
    stack<shared_ptr<Op>> ops;

```

```

auto PopOps = [&](int precedence) { // реализация алгоритма сортировочных станций
    while (!ops.empty() && ops.top()->precedence >= precedence) {
        auto value1 = values.top();
        values.pop();
        auto value2 = values.top();
        values.pop();
        auto op = ops.top();
        ops.pop();
        op->SetRight(value1);
        op->SetLeft(value2);
        values.push(op);
    }
};

while (token != end) {
    const auto &value = *token;
    if (value >= '0' && value <= '9') {
        values.push(make_shared<Value>(value));
    } else if (value == 'x') {
        values.push(make_shared<Variable>(x));
    } else if (value == '*') {
        PopOps(2);
        ops.push(make_shared<Op>(value));
    } else if (value == '+' || value == '-') {
        PopOps(1);
        ops.push(make_shared<Op>(value));
    }
    ++token;
}

while (!ops.empty()) {
    PopOps(0);
}

return values.top();
}

int main() {
    string tokens;
    cout << "Enter expression: ";
    getline(cin, tokens); // считываем выражение
    int x = 0;
    auto node = Parse(tokens.begin(), tokens.end(), x); // парсим выражение
    // по итераторам начала и конца строки
}

```

```
cout << "Enter x: ";
while (cin >> x) {
    cout << "Expression value: " << node->Evaluate() << endl;
    cout << "Enter x: ";
}
return 0;
}
```

Введём различные переменные x для выражения $5 + 7 - x * x + x$ и получим: для $x = 0$ ответ 12, для 1 – 12, для 2 – 10. Всё верно. Введём другое выражение и убедимся в корректности. Кроме того, x может быть отрицательным.

Таким образом, с помощью полученных нами знаний о типе `shared_ptr` и виртуальных методах мы смогли реализовать довольно непростой алгоритм и решить относительно сложную задачу по разбору выражений и вычислению их с использованием внешней переменной.