GeekBrains

Сложные моменты C++

© geekbrains.ru

Вспомогательные классы и лямбда-функции_

На этом уроке

1. Рассмотрим вспомогательные классы языка C++, которые будут полезны при разработке программ.
2. Познакомимся с анонимными функциями и функторами и научимся их использовать.

Оглавление

Введение

Мы приступаем к изучению нового курса: “Сложные моменты C++”. В этом курсе будут рассмотрены дополнительные возможности, которые появились в новых стандартах языка C++ и которые существенно облегчают написание программ. Помимо изучения полезных нововведений современного C++ в курсе подробно рассказывается о библиотеке STL и особенностях ее применения в коде. Также освещаются темы по установки сторонних библиотек с помощью менеджера пакетов Conan. Затрагиваются вопросы многопоточного программирования, сериализации данных, юнит-тестирования и многое др.

В этом уроке мы начинаем знакомиться с дополнительными возможностями современных стандартов языка C++ и познакомимся с кортежами, парами, опциональными типами, типом variant. Еще узнаем про лямбда-функции и функциональные объекты. Эти возможности актуальны для написания лаконичного и красивого кода на C++.

Типы данных

Начнем с рассмотрения следующего примера. Нам дан вектор чисел:

```
vector<int> numbers{ -5, -7, 3 };
```

Необходимо найти среднее арифметическое этих чисел. Напишем программу:

```
vector<int> numbers{ -5, -7, 3 };
int sum = 0;
      for (auto n : numbers) {
      sum += n;
      }
      int average = sum / numbers.size();
      cout << average << endl;
```

Вывод: 1431655762

Программа работает неверно. Дело в том, что метод size() вектора возвращает значение типа size_t. Это специальный беззнаковый целочисленный тип для представления размеров, ведь они не могут быть отрицательными. В нашей программе произошло неявное приведение типа переменной sum к типу std::size_t и, как следствие, ответ оказался неверным.

Как узнать размер типа данных в байтах, а также минимальные и максимальные значения, которые он может содержать. Рассмотрим программу:

```
#include <iostream>
#include <limits> // для функции numeric_limits
using namespace std;

int main()
{
```

```
        cout << sizeof(size_t) << " bytes" << endl;
        cout << "Min: " << numeric_limits<size_t>::min() << endl;
        cout << "Max: " << numeric_limits<size_t>::max() << endl;
}
```

Вывод:

4 bytes

Min: 0

Max: 4294967295

С помощью этой программы вы можете проверить и другие целочисленные типы данных в C++.

Тип	Размер в байтах	Минимум	Максимум
int	4 или 8	-2^{31}	$2^{31}-1$
size_t	4 или 8	0	$2^{31}-1$
int8_t	1	-2^{7}	$2^{7}-1$
int16_t	2	-2^{15}	$2^{15}-1$
int32_t	4	-2^{31}	$2^{31}-1$
int64_t	8	-2^{63}	$2^{63}-1$

Размер типа int зависит от архитектуры ОС и может составлять 4 или 8 байт. Чтобы точно быть уверенным, что тип данных занимает определенное количество байт, следует использовать 4 последних типа из модуля cstdint. В 11-м стандарте C++ были введены и другие целочисленные типы данных с фиксированным количеством байт. С полным списком можно ознакомиться на официальном сайте.

Важно! При вычислении арифметических действий все переменные приводятся к общему типу, который будет самым наибольшим в выражении, при этом приоритет всегда отдается беззнаковым типам.

Нашу программу можно исправить с помощью оператора static_cast, преобразовав тип size_t к типу int:

```
int average = sum / static_cast<int>(numbers.size());
```

Кортежи

Прежде чем мы начнем знакомиться с кортежами, давайте рассмотрим один пример. Создадим класс Time с перегруженным оператором меньше:

```
class Time {
private:
	int hour, minute, second;
public:
	Time() : hour(0), minute(0), second(0) {}
	Time(int h, int m, int s) : hour(h), minute(m), second(s) {}
	friend bool operator< (const Time& t1, const Time& t2);
};
bool operator<(const Time& t1, const Time& t2)
{
	if (t1.hour != t2.hour) {
	return t1.hour < t2.hour;
	}
	if (t1.minute != t2.minute) {
	return t1.minute < t2.minute;
	}
	return t1.second < t2.second;
}

int main()
{
	Time breakfast_time(9, 15, 0);
	Time lunch_time(13,0,0);
	cout << boolalpha << (breakfast_time < lunch_time) << endl;
}
```

Вывод: true

Проблема этого кода состоит в том, что в операторе меньше можно довольно легко запутаться и допустить ошибку. Рассмотрим вариант того, как можно эту реализацию упростить с помощью вектора:

```
bool operator<(const Time& t1, const Time& t2)
{
	return vector<int>{t1.hour, t1.minute, t1.second} <
		vector<int>{t2.hour, t2.minute, t2.second};
}
```

Код выглядит компактнее и понятнее, но использование вектора, достаточно мощного контейнера C++, здесь излишне. Тем более оно было бы невозможно, если мы бы сравнивали пары разных типов данных. Будем использовать вместо вектора функцию tie из заголовочного файла tuple:

```
bool operator<(const Time& t1, const Time& t2)
{
	return tie(t1.hour, t1.minute, t1.second) <
```

```
        tie(t2.hour, t2.minute, t2.second);
}
```

Функция tie() возвращает **кортеж** (tuple) – тип данных, представленный структурой из ссылок на другие типы данных. Для этого типа данных уже перегружен оператор < так, что происходит сравнение по каждому элементу этой структуры.

В C++ есть основные свойства, присущие кортежам - их можно создавать, сравнивать и распаковывать. Кроме того, значения кортежей можно менять.

```
        // создание кортежа
        tuple<int, string, double> tuple1(1, "Hello", 2.72);
        auto tuple2 = make_tuple(2, "world", 3.14);
        //auto tuple3 = tie(1, "Hello", 2.72); // ошибка

    // получение значений кортежа
    cout << get<1>(tuple1) << endl;
        cout << get<2>(tuple2) << endl;

        // изменение значений кортежа
    get<0>(tuple1) += get<0>(tuple2);
    cout << get<0>(tuple1) << endl;

    // распаковка значений кортежа
    int x;
        string s;
        double d;
        tie(x, s, d) = tuple1;
        cout << x << " " << s << " " << d << endl;
```

Вывод:

Hello

3.14

3

3 Hello 2.72

Создать кортеж можно через конструктор, передав параметры, либо через функцию make_tuple. Также можно создать кортеж, используя функцию tie, но она должна принимать ссылки на объекты, которые уже хранятся в каких-нибудь переменных.

Получить или изменить значение элемента кортежа можно, используя функцию get, которая в параметре шаблона (в <> скобках) принимает порядковый номер элемента, а в круглых скобках – кортеж.

Распаковка позволяет получать из кортежа сразу несколько значений в отдельные переменные, с которыми дальше проще работать.

Рассмотрим еще один пример с распаковкой элементов кортежа:

```
void print_student(size_t id, const string& name, double grade)
{
	cout << "Student " << quoted(name) << ", ID: " << id << ", Grade: " << grade
<< '\n';
}

int main() {
	auto students = {
		make_tuple(234, "John Doe"s, 3.7),
		make_tuple(345, "Billy Foo"s, 4.0),
		make_tuple(456, "Cathy Bar"s, 3.5),
	};
	for (const auto& student : students) {
		apply(print_student, student);
	}
}
```

Вывод:

Student "John Doe", ID: 234, Grade: 3.7

Student "Billy Foo", ID: 345, Grade: 4

Student "Cathy Bar", ID: 456, Grade: 3.5

Функция apply — это вспомогательная функция (из заголовочного файла tuple) для работы с кортежами, которая позволяет нам писать код, не имея сведений обо всех типах данных, которые появляются в нашем коде.

В данном примере у нас есть массив кортежей. Кортеж имеет тип tuple<int, string, double>. Также у нас есть функция print_student с сигнатурой void print_student(size_t, const string&, double). Вызов функции apply(print_student, student) приведет к вызову функции print_student с соответствующими параметрами, извлеченными из кортежа. Если фукнция print_student что-то возвращала бы, то apply вернула бы результат работы функции print_student.

Пары

Пара – это кортеж с двумя элементами. По сравнению с кортежем пара обладает более понятным синтаксисом, а именно доступ к элементам осуществляется при помощи полей first и second.

Основные операции, которые можно производить с парами такие же, как и у кортежей.

```
    // создание пары
```

```
    pair<int, string> pair1(1, "Hello");
    auto pair2 = make_pair(2, "world");

    // получение значений пары
    cout << pair1.first << " " << pair1.second << endl;
    cout << pair2.first << " " << pair2.second << endl;

    // изменение значений пары
  pair1.first += pair2.first;

  // распаковка значений пары
  int x;
  string s;
    tie(x, s) = pair1;
    cout << x << " " << s << endl;
```

Вывод:

1 Hello

2 world

3 Hello

Создание пары может осуществляться в конструкторе, либо при помощи метода make_pair.

В C++17 можно не указывать шаблонные параметры у кортежа и пары при объявлении при условии, что они будут проинициализированы.

```
tuple tuple1(1, "Hello", 2.72);
pair pair1(1, "Hello");
```

Распаковка пары может пригодиться, когда мы захотим пройтись по элементам вектора, состоящего из пар.

```
// Фамилия поэта и год рождения
  vector<pair<string, int>> poets{
    {"Pushkin", 1799},
    {"Lermontov", 1814},
    {"Zhukovsky", 1783}
    };
    for (const auto& [surname, birthday] : poets) {
    cout << surname << " " << birthday << endl;
    }
```

Код становится намного понятнее, когда мы вместо обращения first и second используем переменные с соответствующими именами.

Возврат нескольких значений из функции

Пары и кортежи позволяют возвращать из функции сразу несколько значений. Рассмотрим пример класса, который содержит список поэтов и функцию поиска года рождения поэта по его фамилии. Если фамилия поэта не будет найдена в векторе, то возвращается false и 0.

```
class Poets {
private:
	vector<pair<string, int>> poets;
public:
	Poets() {
		poets = {
			{"Pushkin", 1799},
			{"Lermontov", 1814},
			{"Zhukovsky", 1783}
		};
	}
	pair<bool, int> FindPoet(string find_surname) {
	for (const auto& [surname, birthday] : poets) {
		if (surname == find_surname) {
		return {true, birthday};
		}
	}
	return {false, 0 };
	}
};

int main()
{
	Poets poets_list;
	bool success;
	int year;
	tie(success, year) = poets_list.FindPoet("Lermontov");
	cout << success << " " << year << endl;
	tie(success, year) = poets_list.FindPoet("Tyutchev");
	cout << success << " " << year << endl;
}
```

Пару значений, возвращаемую методом FindPoet, удобно обрабатывать, т.к. по первому элементу пары понятно, найдено ли значение года рождения, а во втором элементе пары хранится сам года рождения.

Тип optional

Ситуация, рассмотренная в предыдущем примере, настолько часто встречается, что в C++17 был введен специальный тип данных - optional. Опциональный тип состоит из значения и флага, который используется для обозначения того, доступно значение или нет. Такая обёртка выразительно представляет объект, который может быть пустым. Для его использования необходимо подключить заголовочный файл **optional**.

Перепишем метод FindPoet с использованием типа optional:

```
optional<int> FindPoet(string find_surname) {
      for (const auto& [surname, birthday] : poets) {
            if (surname == find_surname) {
            return birthday;
            }
      }
      return nullopt;
}
```

Пара pair<bool, int> заменилась на тип optional<int>. Если поэт был найден в векторе, возвращается его год рождения. Если поэт не был найден возвращается nullopt. Это специальное слово означает отсутствие значения.

В функции main мы можем получить значение типа optional и проверить его, содержит ли оно реальное значение или нет.

```
Poets poets_list;

      optional<int> year = poets_list.FindPoet("Lermontov");
      if (year) cout << *year << endl;
      else cout << "not found" << endl;

      year = poets_list.FindPoet("Tyutchev");
      if (year.has_value()) cout << year.value() << endl;
      else cout << "not found" << endl;
```

Вывод:

1814

not found

Optional имеет оператор приведения к типу bool, поэтому мы можем проверить, содержит ли этот тип значение при помощи конструкции if. Тот же результат мы получим, если воспользуемся метод has_value, который возвращает true или false. Чтобы получить само значение, можно воспользоваться

оператором * или методом value. Если вы попытаетесь получить значение опционального типа, когда метод has_value возвращает false, то получите исключение.

У типа optional также есть метод value_or, который возвращает значение, если оно доступно, в противном случае будет возвращаться значение, указанное в круглых скобках:

```
optional<int> year = poets_list.FindPoet("Lermontov");
 cout << year.value_or(-1) << endl;

 year = poets_list.FindPoet("Tyutchev");
 cout << year.value_or(-1) << endl;
```

Вывод:

1814

-1

Если мы попробуем получить доступ к объекту типа optional, который не содержит значения, то будет сгенерировано исключение logic_error. Таким образом, нельзя работать с объектами этого типа, не проверяя их.

Замена void* на класс any

Может случиться так, что нам понадобится сохранить элементы любого типа в переменной. В языке C и C++ до принятия C++17 мы имели возможность хранить указатели на различные объекты в указателе типа void*. Такой указатель сам по себе не может сказать, на какой объект ссылается, поэтому нужно вручную создать некий дополнительный механизм, который сообщит, чего стоит ожидать. Данное решение приводит к созданию некрасивого и небезопасного кода.

В C++17 появляется тип std::any. Для использования этого типа необходимо подключить заголовочный файл any. Он разработан для того, чтобы хранить переменные любого вида, и предоставляет средства, которые позволяют выполнить проверку, безопасную для типов, и получить доступ к данным.

В следующем примере мы реализуем функцию, которая пробует вывести на экран какие-либо данные. В качестве ее типа аргумента служит тип any.

```
void print_anything(const any& a)
{
	if (!a.has_value()) {
		cout << "Nothing.\n";
	}
	else if (a.type() == typeid(string)) {
```

```
                cout << "It's a string: " << quoted(any_cast<const string&>(a)) <<
'\n';
        }
        else if (a.type() == typeid(int)) {
                cout << "It's an integer: " << any_cast<int>(a) << '\n';
        }
        else if (a.type() == typeid(vector<int>)) {
                cout << "It's a vector: ";
                for (auto n : any_cast<const vector<int>&>(a)) cout << n << " ";
                cout << '\n';
        }
        else {
                cout << "Can't handle this item.\n";
        }
}

int main() {
        print_anything({});
        print_anything("abc"s);
        print_anything(123);
        print_anything(vector<int>{1, 2, 3});
}
```

Вывод:

Nothing.

It's a string: "abc"

It's an integer: 123

It's a vector: 1 2 3

В функции print_anything вначале мы проверили, содержит ли аргумент какие-то данные, или же это пустой экземпляр типа any. Если он не пуст, то можно попробовать сравнивать его с разными типами до тех пор, пока не получим совпадение. Сравнение осуществляется с использованием функции typeid из заголовочного файла typeinfo. Преобразование к определенному типу данных можно выполнить с помощью any_cast. Обратите внимание: any_cast<T>(x) возвращает копию внутреннего значения. Если нужно получить ссылку, чтобы избежать копирования сложных объектов, то следует использовать конструкцию any_cast<T&>(x).

Заменяем union на std::variant

Тип **variant**, по сути, представляет собой обновленную версию типа **union**. Он не использует кучу, поэтому настолько же эффективно задействует память и время, как и решение, основанное на объединениях, так что нет нужды описывать решения через union самостоятельно. С появлением класса variant union можно считать устаревшим. Тип может хранить все что угодно, кроме ссылок

массивов или объектов типа void. Для того, чтобы использовать тип variant, необходимо подключить заголовочный файл variant.

Рассмотрим пример записи данных в переменную типа variant и извлечения данных.

```
variant<string, int, bool> var = 42;
int number = get<int>(var);
cout << number << '\n';
var = string("Hello!");;
string* str = get_if<string>(&var);
if (str) cout << *str << '\n';
var = false;
```

Вывод:

42

Hello!

Если мы точно знаем, какой тип содержится в переменной variant, то мы можем ее извлечь с помощью функции get. Если же произойдет ошибка при извлечении, то будет выброшено исключение. Функция get_if вернет нулевой указатель в случае ошибки, поэтому сначала рекомендуется проверить этот указатель.

Рассмотрим пример посложнее.

Создадим два класса cat и dog и создадим массив из объектов этих классов, не прибегая к полиморфизму.

```
class cat {
	string name;
public:
	cat(string n) : name{ n } {}
	void meow() const {
		cout << name << " says Meow!\n";
	}
};

class dog {
	string name;
public:
	dog(string n) : name{ n } {}
	void woof() const {
		cout << name << " says Woof!\n";
	}
};

using animal = variant<dog, cat>;

struct animal_voice
{
```

```
        void operator()(const dog& d) const { d.woof(); }
        void operator()(const cat& c) const { c.meow(); }
};


int main() {
        vector<animal> animals{ cat{"Tuba"}, dog{"Balou"}, cat{"Bobby"} };
        for (const animal& a : animals) {
                visit(animal_voice{}, a);
        }
}
```

Вывод:

Tuba says Meow!

Balou says Woof!

Bobby says Meow!

Пользовательский тип animal является псевдонимом типа variant<dog, cat>. Далее мы определили структуру animal_voice, в которой дважды реализуется оператор (). Одна из реализаций — перегруженная версия, принимающая экземпляры типа dog, вторая же принимает экземпляры типа cat. Для этих типов она просто вызывает функции woof или meow.

Функция visit принимает объект структуры и экземпляр типа variant. Объект структуры должен реализовывать перегруженные версии для всех вероятных типов, которые может хранить variant. В нашем примере в структуре animal_voice перегружены операторы () для двух типов: dog и cat.

Тип variant похож на тип any, поскольку они оба могут содержать объекты разных типов, и нужно определять во время работы программы, что именно в них хранится, прежде чем получить доступ к их содержимому.

С другой стороны, тип variant отличается от any тем, что мы должны объявлять, экземпляры каких типов он может хранить в виде списка шаблонных типов. Экземпляр типа variant<A, B, C> должен хранить один экземпляр типа A, B или C. Нельзя сделать так, чтобы в экземпляре типа variant не хранился ни один экземпляр. Это значит, что тип variant не поддерживает возможность опциональности.

Лямбда-функции

Вернемся к примеру с вектором, содержащим пары фамилий поэтов и их годов рождения. Давайте отсортируем этот вектор по второму элементу пары, то есть по годам рождения. Для этого воспользуемся стандартной алгоритмической функцией sort. Но нам нужно как-то указать этой функции, что сравнение должно производиться по второму элементу пары. Для этого в качестве 3

параметра функции sort передадим функцию, в которой задается правило сравнения элементов вектора. Такие функции часто называют «компараторы», от англ compare, сравнивать.

```
#include <iostream>
#include <vector>
#include <algorithm> // для sort

using namespace std;

bool Compare(pair<string, int> poet1, pair<string, int> poet2) {
	return poet1.second < poet2.second;
}

int main()
{
	vector<pair<string, int>> poets{
	{"Pushkin", 1799},
	{"Lermontov", 1814},
	{"Zhukovsky", 1783}
	};
	sort(poets.begin(), poets.end(), Compare);
	for (const auto& [surname, birthday] : poets) {
	cout << surname << " " << birthday << endl;
    }
}
```

Вывод:

Zhukovsky 1783

Pushkin 1799

Lermontov 1814

Функция Compare возвращает true, если первый аргумент меньше, чем второй. В противном случае возвращает false.

Этот пример рабочий, но его можно улучшить. Проблема состоит в том, что функция sort требует указатель на функцию в качестве аргумента. Из-за этого мы вынуждены определить новую функцию, которая будет использована только один раз, дать ей имя и поместить её в глобальную область видимости (т.к. функции не могут быть вложенными!). При этом она будет настолько короткой, что быстрее и проще понять её смысл, посмотрев лишь на одну строку кода, нежели изучать описание этой функции и её имя.

Лямбда-выражение (или просто «лямбда») в программировании позволяет определить анонимную функцию внутри другой функции.

Синтаксис лямбда-выражений является одним из самых странных в языке C++, и вам может потребоваться некоторое время, чтобы к нему привыкнуть.

Лямбда-выражения имеют следующий синтаксис:

```
[ захватываемыеВыражения ] ( параметры ) -> возвращаемыйТип
{
   Выражения;
}
```

Поля **захватываемыеВыражения** (о которых речь пойдет дальше) и **параметры** могут быть пустыми, если они не требуются программисту.

Также обратите внимание, что лямбда-выражения не имеют имен, поэтому нам и не нужно будет их предоставлять.

Давайте перепишем предыдущий пример, но уже с использованием лямбда-выражений:

```
int main()
{
       vector<pair<string, int>> poets{
       {"Pushkin", 1799},
       {"Lermontov", 1814},
       {"Zhukovsky", 1783}
       };
       sort(poets.begin(), poets.end(),
                            [](pair<string, int> poet1, pair<string, int> poet2)
                             {return poet1.second < poet2.second; } );

       for (const auto& [surname, birthday] : poets) {
       cout << surname << " " << birthday << endl;
    }
}
```

При этом всё работает точно так же, как и в случае с указателем на функцию. В данном примере мы не указывали явно тип возвращаемого значения, он был выведен компилятором автоматически. Можно было бы и явно указать тип возвращаемого значения:

```
[](pair<string, int> poet1, pair<string, int> poet2) -> bool
                              {return poet1.second < poet2.second; }
```

Тип лямбда-выражения

Написание лямбды в той же строке, где она используется, иногда может затруднить чтение кода. Поэтому мы можем создать переменную, которая будет хранить лямбда-выражение.

```
auto compare = [](pair<string, int> poet1, pair<string, int> poet2)
```

```
            {return poet1.second < poet2.second; };
```

На самом деле у лямбд нет типа, который мы могли бы явно использовать. Когда мы пишем лямбду, компилятор генерирует уникальный тип лямбды, который нам не виден. Мы можем использовать ключевое слово auto (начиная с C++11), чтобы получить переменную, являющуюся лямбда-функцией.

Хотя мы не знаем тип лямбды, есть несколько способов её хранения для использования после определения. Если лямбда ничего не захватывает (т.е. квадратные скобки пустые), то мы можем использовать обычный указатель на функцию. Как только лямбда что-либо захватывает (в квадратных скобках есть какие-то выражения), указатель на функцию больше не будет работать. Однако function (из заголовочного файла `functional`) может использоваться для лямбд, даже если они что-то захватывают:

```
// сохранение лямбды в указатель на функцию
bool (*compare2) (pair<string, int> poet1, pair<string, int> poet2) =
       [](pair<string, int> poet1, pair<string, int> poet2)
       {return poet1.second < poet2.second; };

// сохранение лямбды в тип function
function compare3 = [](pair<string, int> poet1, pair<string, int> poet2)
                     {return poet1.second < poet2.second; };
```

Лямбда-выражения можно передавать в другие функции, при этом в качестве параметра для лямбды будет использоваться тип functional:

```
void repeat(int repetitions, const function<void(int)>& fn)
{
       for (int i{ 0 }; i < repetitions; ++i)
       {
       fn(i);
       }
}

int main()
{
       repeat(3, [](int i) { cout << i << '\n'; } );
}
```

Вывод:

0

1

2

В function указан тип возвращаемого значения лямбды, а также типы параметров.

Лямбда-захват

Давайте напишем программу, которая будет выводить всех поэтов, родившихся в определенном веке. Реализуем через стандартную алгоритмическую функцию for_each. Эта функция первым параметром принимает итератор на начало контейнера, вторым параметром – итератор на конец контейнера, третьим – функцию, которая будет применяться к результату разыменовывания каждого итератора контейнера.

```
int main()
{
	vector<pair<string, int>> poets{
	{"Pushkin", 1799},
	{"Lermontov", 1814},
	{"Zhukovsky", 1783}
	};
	int century ;
	cin >> century;
	for_each(poets.begin(), poets.end(),
	[](const auto& poet)
	{
		if (poet.second < century * 100 && poet.second >= (century -1)*100)
		cout << poet.first << endl;
	});
}
```

В этой программе мы предоставили пользователю ввести век самостоятельно. Соответственно лямбда должна использовать значение century. Но данный код не скомпилируется, т.к. лямбдам доступны только определенные внешние идентификаторы: глобальные идентификаторы, объекты, известные во время компиляции и со статической продолжительностью жизни. Переменная century не соответствует ни одному из этих требований, поэтому лямбда не может её увидеть. Вот для этого и существует **лямбда-захват**.

Чтобы лямбда увидела переменную century, необходимо поместить ее в квадратных скобках (в области захвата).

```
for_each(poets.begin(), poets.end(),
	[century](const auto& poet)
	{
		if (poet.second < century * 100 && poet.second >= (century -1)*100)
		cout << poet.first << endl;
	});
```

В таком случае программа скомпилируется и будет работать. Хотя может показаться, будто в вышеприведенном примере наша лямбда напрямую обращается к значению переменной century, но это не так. Когда выполняется лямбда-определение, то для каждой захватываемой переменной

внутри лямбды создается клон этой переменной (с идентичным именем). Данные переменные-клоны инициализируются с помощью переменных из внешней области видимости с тем же именем. То есть имя захватываемой переменной должно совпадать с захватом в лямбде.

По умолчанию переменные захватываются как константные значения. Это означает, что при создании лямбды, она захватывает константную копию переменной из внешней области видимости, что означает, что значения этих переменных лямбда изменить не может.

Давайте теперь посчитаем, сколько поэтов удовлетворяет нашему условию:

```
int main()
{
       vector<pair<string, int>> poets{
       {"Pushkin", 1799},
       {"Lermontov", 1814},
       {"Zhukovsky", 1783}
    };
       int century, count=0;
       cin >> century;
       for_each(poets.begin(), poets.end(),
       [century, count](const auto& poet)
       {
              if (poet.second < century * 100 && poet.second >= (century - 1) * 100)
{
              cout << poet.first << endl;
              count++;
              }
       });
    cout << count << endl;
}
```

Создадим переменную count и захватим ее лямбдой. Если условие выполняется, то будем увеличивать ее значение. Данный код не скомпилируется, т.к. захват происходит путем создания константной копии переменной. Нам бы хотелось переменную count изменять, поэтому нужно лямбду пометить ключевым словом mutable:

```
    for_each(poets.begin(), poets.end(),
       [centure, count](const auto& poet) mutable
       {
            if (poet.second < centure * 100 && poet.second >= (centure - 1) * 100) {
              cout << poet.first << endl;
              count++;
              }
       });
```

В данном контексте, ключевое слово **mutable** удаляет спецификатор const со всех переменных, захваченных по значению. Теперь код компилируется, но выводит 0. Программа работает неверно. Обратите внимание, что переменная со столетием стала также доступна к изменению

При вызове лямбда захватила копию переменной count. Затем, когда лямбда увеличивает значение переменной count, то, на самом деле, она уменьшает значение копии, а не исходной переменной.

Подобно тому, как функции могут изменять значения аргументов, передаваемых им по ссылке, мы также можем захватывать переменные по ссылке, чтобы позволить нашей лямбде влиять на значения аргументов. Для нашего примера мы бы хотели изменять count, как если бы передали его по ссылке в обычную функцию.

Чтобы захватить переменную по ссылке, мы должны добавить знак амперсанда (&) к имени переменной, которую хотим захватить. В отличие от переменных, которые захватываются по значению, переменные, которые захватываются по ссылке, не являются константными (если только переменная, которую они захватывают, не является изначально const).

Изменим нашу лямбду:

```
	for_each(poets.begin(), poets.end(),
	[centure, &count](const auto& poet)
	{
		if (poet.second < centure * 100 && poet.second >= (centure - 1) * 100)
{
		cout << poet.first << endl;
		count++;
		}
	});
```

Теперь при вводе числа 18 будет выведено две фамилии и число 2. Все работает отлично.

В некоторых программах есть необходимость захватить много переменных и перечислять их все будет обременительно. Если вы изменяете свою лямбду, то вы можете забыть добавить или удалить захватываемые переменные. К счастью, есть возможность заручиться помощью компилятора для автоматической генерации списка переменных, которые нам нужно захватить.

Захват по умолчанию захватывает все переменные, упомянутые в лямбде. Если используется захват по умолчанию, то переменные, не упомянутые в лямбде, не будут захвачены.

Чтобы захватить все задействованные переменные по значению, используйте = в качестве значения для захвата. Чтобы захватить все задействованные переменные по ссылке, используйте & в качестве значения для захвата.

Давайте модифицируем нашу программу со списком поэтов. Теперь мы хотим выводить фамилии поэтов, рожденных в заданном пользователем диапазоне лет.

```
int main()
{
	vector<pair<string, int>> poets{
```

```
        {"Pushkin", 1799},
        {"Lermontov", 1814},
        {"Zhukovsky", 1783}
        };
        int count=0;
        int min_year, max_year;
        cin >> min_year >> max_year;
    for_each(poets.begin(), poets.end(),
        [=, &count](const auto& poet)
        {
                if (poet.second < max_year && poet.second >= min_year) {
                cout << poet.first << endl;
                count++;
                }
        });
    cout << count << endl;
}
```

Благодаря оператору = в области захвата лямбда функции мы захватываем все переменные (кроме count), которые будут использоваться в лямбде, по значению. В то же время, переменную count мы захватываем по ссылке. Вполне допускается захватить некоторые переменные по значению, а другие — по ссылке, но при этом каждая переменная может быть захвачена только один раз:

```
    int count = 0;
    int min_year, max_year;
        // захватываем переменные min_year и max_year по значению, а count - по
ссылке
    [min_year, max_year, &count]() {};
        // захватываем переменныую count по ссылке, а все остальные по значению
    [=, &count]() {};
    // захватываем переменную min_year по значению, а все остальные по ссылке
    [&, min_year]() {};
    // запрещено, так как мы уже определили захват по ссылке для всех переменных
    [&, &min_year]() {};
    // запрещено, так как мы уже определили захват по значению для всех переменных
    [=, min_year]() {};
    // Запрещено, так как переменная min_year используется дважды
    [min_year, &max_year, &min_year]() {};
        // Запрещено, так как захват по умолчанию должен быть первым элементом в
списке захвата
        [min_year, &]() {};
```

Переменные захватываются в точке определения лямбды. Если переменная, захваченная по ссылке, прекращает свое существование до прекращения существования лямбды, то лямбда остается с висячей ссылкой:

```
// Функция возвращает лямбду
auto makeWalrus(const string& name)
{
```

```
    // Захват переменной name по ссылке и возврат лямбды
    return [&]() {
       cout << "I am a walrus, my name is " << name << '\n'; // неопределенное
поведение
       };
}

int main()
{
    // Создаем новый объект с именем Roofus.
    // sayName является лямбдой, возвращаемой функцией makeWalrus()
    auto sayName{ makeWalrus("Roofus") };

    // Вызов лямбды, которую возвращает функция makeWalrus()
    sayName();
}
```

Вызов функции makeWalrus() создает временный объект string из строкового литерала "Roofus". Лямбда в функции makeWalrus() захватывает временную строку по ссылке. Данная строка уничтожается при выполнении возврата makeWalrus(), но при этом лямбда все еще ссылается на нее. Затем, когда мы вызываем sayName(), происходит попытка доступа к висячей ссылке, что чревато неопределенным поведением (unexpected behavior).

Если мы хотим, чтобы захваченная переменная name была валидна, когда используется лямбда, то нам нужно захватить данную переменную по значению (либо явно, либо с помощью захвата по умолчанию по значению).

Копии лямбд

Рассмотрим следующий код:

```
void invoke(const function<void(void)>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Выполняем инкремент и выводим на экран локальную копию переменной i
    auto count{ [i]() mutable {
       cout << ++i << '\n';
       } };

       invoke(count);
       invoke(count);
       invoke(count);
}
```

Вывод:

1

1

1

В данном примере мы передаем лямбду в функцию invoke и вызываем ее. Когда с помощью лямбды создается объект function, то он внутри себя создает копию лямбда-выражения. Таким образом, наш вызов fn() фактически выполняется при использовании копии лямбды, а не самой лямбды.

Для передачи лямбды в функцию по ссылке в C++ есть функция ref (из заголовочного файла functional). Ref позволяет нам передавать обычный тип, как если бы это была ссылка. Обёртывая нашу лямбду в ref всякий раз, когда кто-либо пытается сделать копию нашей лямбды, он будет делать копию ссылки, а не фактического объекта.

Вот наш обновленный код с использованием ref:

```
void invoke(const function<void(void)>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Выполняем инкремент и выводим на экран локальную копию переменной i
    auto count{ [i]() mutable {
       cout << ++i << '\n';
       } };

       invoke(ref(count));
       invoke(ref(count));
       invoke(ref(count));
}
```

Вывод:

1

2

3

Функциональные объекты

На самом деле, лямбды не являются функциями. Лямбды являются особым типом объектов, который называется **функциональным объектом** или **функтором**. Функторы — это объекты, содержащие перегруженный operator(), который и делает их вызываемыми подобно обычным функциям. Преимущество функтора над обычной функцией заключается в том, что функторы могут хранить данные в переменных-членах (поскольку они сами являются классами).

Рассмотрим простейший пример использования функтора, когда в классе перегружается операция (), а потом объект класса используется подобно функции, принимая аргументы.

```
class Square {
public:
       int operator() (const int value) const {
       return value * value;   //возводим в квадрат
    }
};

int main() {
       Square sq;
       cout << sq(10) << '\n';
       cout << sq(20) << '\n';
}
```

Вывод:

100

400

На вопрос "когда функциональные объекты полезны?" можно сказать, что они полезны тогда, когда функции должны вести себя как объекты. В C++ функторы занимают важное место в стандартной библиотеке шаблонов (в STL). Например, в алгоритмы сортировки можно передавать некоторый критерий для поиска, а роль этого критерия берёт на себя функтор:

```
class Less {
public:
       bool operator() (const int& left, const int& right) {
       return left < right;
       }
};

int main() {
       vector<int> numbers = { 3,4,1,5,2 };
       Less less;    //создаём функциональный объект, т. е. функтор
    sort(numbers.begin(), numbers.end(), less);   //передаём функтор в алгоритм
сортировки
    for (const int& n : numbers) {
       cout << n << " ";
```

```
        }
}
```

Вывод: 1 2 3 4 5

Во многих случаях функторы представляют собой предикаты:

- **Предикат** — это функция, которая возвращает булево значение
- **Предикатный класс** — это класс, перегруженная операция () которого возвращает булево значение

Домашнее задание

Имеется база сотрудников и номеров их телефонов. Требуется написать соответствующие структуры для хранения данных, и заполнить контейнер записями из базы. Затем необходимо реализовать методы обработки данных, а также вывести на экран всю необходимую информацию.

Важно! Имена переменным, классам и функциям давайте осознанные, состоящие из слов на английском языке.

1. Создайте структуру **Person** с 3 полями: фамилия, имя, отчество. Поле отчество должно быть опционального типа, т.к. не у всех людей есть отчество. Перегрузите оператор вывода данных для этой структуры. Также перегрузите операторы < и == (используйте tie).
2. Создайте структуру **PhoneNumber** с 4 полями:
   - код страны (целое число)
   - код города (целое число)
   - номер (строка)
   - добавочный номер (целое число, опциональный тип)

   Для этой структуры перегрузите оператор вывода. Необходимо, чтобы номер телефона выводился в формате: +7(911)1234567 12, где 7 – это номер страны, 911 – номер города, 1234567 – номер, 12 – добавочный номер. Если добавочного номера нет, то его выводить не надо.
3. Создайте класс **PhoneBook**, который будет в контейнере хранить пары: Человек – Номер телефона. Конструктор этого класса должен принимать параметр типа *ifstream* – поток данных, полученных из файла. В теле конструктора происходит считывание данных из файла и

заполнение контейнера. Класс PhoneBook должен содержать перегруженный оператор вывода, для вывода всех данных из контейнера в консоль.

В классе PhoneBook реализуйте метод **SortByName**, который должен сортировать элементы контейнера по фамилии людей в алфавитном порядке. Если фамилии будут одинаковыми, то сортировка должна выполняться по именам, если имена будут одинаковы, то сортировка производится по отчествам. Используйте алгоритмическую функцию *sort*.

Реализуйте метод **SortByPhone**, который должен сортировать элементы контейнера по номерам телефонов. Используйте алгоритмическую функцию *sort*.

Реализуйте метод **GetPhoneNumber**, который принимает фамилию человека, а возвращает кортеж из строки и PhoneNumber. Строка должна быть пустой, если найден ровно один человек с заданном фамилией в списке. Если не найден ни один человек с заданной фамилией, то в строке должна быть запись «not found», если было найдено больше одного человека, то в строке должно быть «found more than 1». Для прохода по элементам контейнера используйте алгоритмическую функцию for_each.

Реализуйте метод **ChangePhoneNumber,** который принимает человека и новый номер телефона и, если находит заданного человека в контейнере, то меняет его номер телефона на новый, иначе ничего не делает. Используйте алгоритмическую функцию *find_if*.

Функция main будет выглядеть так:

```
int main() {

	ifstream file("XXX"); // путь к файлу PhoneBook.txt
	PhoneBook book(file);
	cout << book;

	cout << "------SortByPhone-------" << endl;
	book.SortByPhone();
	cout << book;

	cout << "------SortByName--------" << endl;
	book.SortByName();
	cout << book;

	cout << "-----GetPhoneNumber-----" << endl;
	// лямбда функция, которая принимает фамилию и выводит номер телефона этого 		человека, либо строку с ошибкой
	auto print_phone_number = [&book](const string& surname) {
		cout << surname << "\t";
		auto answer = book.GetPhoneNumber(surname);
		if (get<0>(answer).empty())
			cout << get<1>(answer);
		else
			cout << get<0>(answer);
			cout << endl;
```

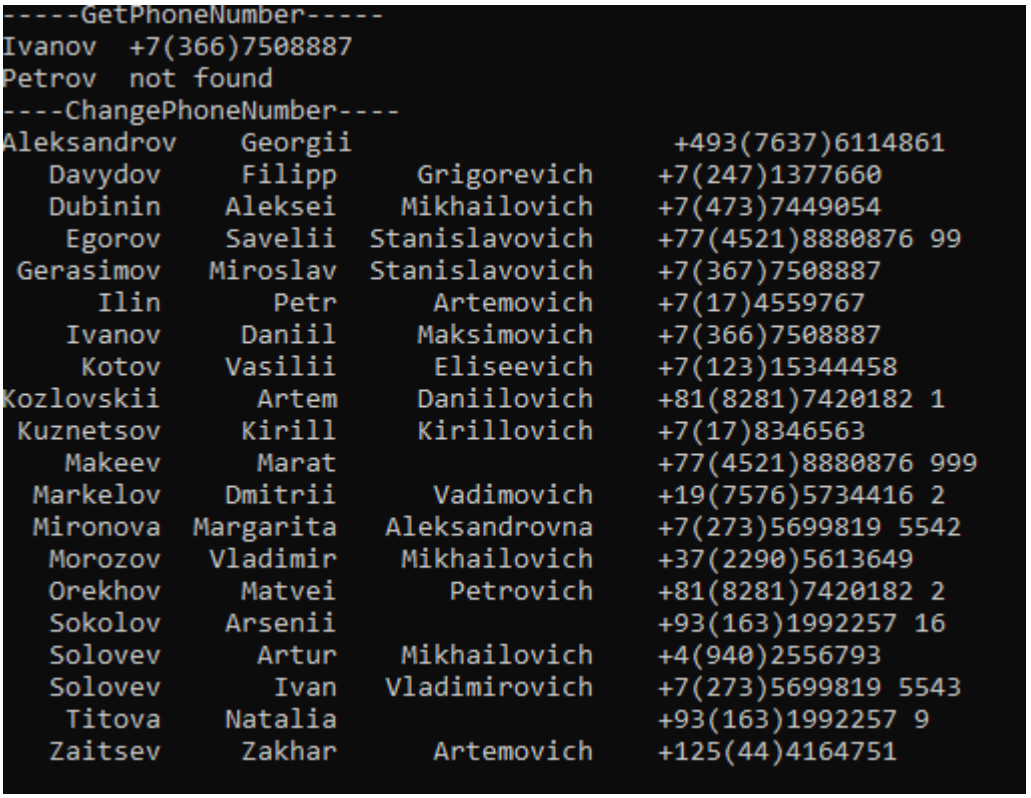
```
        };

        // вызовы лямбды
        print_phone_number("Ivanov");
        print_phone_number("Petrov");

        cout << "----ChangePhoneNumber----" << endl;
        book.ChangePhoneNumber(Person{ "Kotov", "Vasilii", "Eliseevich" },
        PhoneNumber{7, 123, "15344458", nullopt});
        book.ChangePhoneNumber(Person{ "Mironova", "Margarita", "Vladimirovna" },
PhoneNumber{ 16, 465, "9155448", 13 });
        cout << book;
 }
```

У вас должен получиться следующий вывод:

```
      Ilin        Petr       Artemovich    +7(17)4559767
   Zaitsev      Zakhar       Artemovich    +125(44)4164751
   Dubinin     Aleksei     Mikhailovich    +7(473)7449054
   Solovev       Artur     Mikhailovich    +4(940)2556793
 Gerasimov    Miroslav   Stanislavovich    +7(367)7508887
    Makeev       Marat                     +77(4521)8880876 999
   Solovev        Ivan    Vladimirovich    +7(273)5699819 5543
    Egorov     Savelii   Stanislavovich    +77(4521)8880876 99
   Sokolov     Arsenii                     +93(163)1992257 16
   Davydov      Filipp      Grigorevich    +7(247)1377660
   Morozov    Vladimir     Mikhailovich    +37(2290)5613649
   Orekhov      Matvei        Petrovich    +81(8281)7420182 2
    Titova     Natalia                     +93(163)1992257 9
  Markelov     Dmitrii       Vadimovich    +19(7576)5734416 2
Kozlovskii       Artem      Daniilovich    +81(8281)7420182 1
 Kuznetsov      Kirill      Kirillovich    +7(17)8346563
  Mironova   Margarita    Aleksandrovna    +7(273)5699819 5542
     Kotov     Vasilii       Eliseevich    +7(367)7508888
    Ivanov      Daniil      Maksimovich    +7(366)7508887
Aleksandrov     Georgii                      +493(7637)6114861
------SortByPhone-------
   Solovev       Artur     Mikhailovich    +4(940)2556793
      Ilin        Petr       Artemovich    +7(17)4559767
 Kuznetsov      Kirill      Kirillovich    +7(17)8346563
   Davydov      Filipp      Grigorevich    +7(247)1377660
  Mironova   Margarita    Aleksandrovna    +7(273)5699819 5542
   Solovev        Ivan    Vladimirovich    +7(273)5699819 5543
    Ivanov      Daniil      Maksimovich    +7(366)7508887
 Gerasimov    Miroslav   Stanislavovich    +7(367)7508887
     Kotov     Vasilii       Eliseevich    +7(367)7508888
   Dubinin     Aleksei     Mikhailovich    +7(473)7449054
  Markelov     Dmitrii       Vadimovich    +19(7576)5734416 2
   Morozov    Vladimir     Mikhailovich    +37(2290)5613649
    Egorov     Savelii   Stanislavovich    +77(4521)8880876 99
    Makeev       Marat                     +77(4521)8880876 999
Kozlovskii       Artem      Daniilovich    +81(8281)7420182 1
   Orekhov      Matvei        Petrovich    +81(8281)7420182 2
    Titova     Natalia                     +93(163)1992257 9
   Sokolov     Arsenii                     +93(163)1992257 16
   Zaitsev      Zakhar       Artemovich    +125(44)4164751
Aleksandrov     Georgii                      +493(7637)6114861
------SortByName--------
Aleksandrov     Georgii                      +493(7637)6114861
   Davydov      Filipp      Grigorevich    +7(247)1377660
   Dubinin     Aleksei     Mikhailovich    +7(473)7449054
    Egorov     Savelii   Stanislavovich    +77(4521)8880876 99
 Gerasimov    Miroslav   Stanislavovich    +7(367)7508887
      Ilin        Petr       Artemovich    +7(17)4559767
    Ivanov      Daniil      Maksimovich    +7(366)7508887
     Kotov     Vasilii       Eliseevich    +7(367)7508888
Kozlovskii       Artem      Daniilovich    +81(8281)7420182 1
 Kuznetsov      Kirill      Kirillovich    +7(17)8346563
    Makeev       Marat                     +77(4521)8880876 999
  Markelov     Dmitrii       Vadimovich    +19(7576)5734416 2
  Mironova   Margarita    Aleksandrovna    +7(273)5699819 5542
   Morozov    Vladimir     Mikhailovich    +37(2290)5613649
   Orekhov      Matvei        Petrovich    +81(8281)7420182 2
   Sokolov     Arsenii                     +93(163)1992257 16
   Solovev       Artur     Mikhailovich    +4(940)2556793
   Solovev        Ivan    Vladimirovich    +7(273)5699819 5543
    Titova     Natalia                     +93(163)1992257 9
   Zaitsev      Zakhar       Artemovich    +125(44)4164751
```



Дополнительные материалы

1. C++0x (C++11). Лямбда-выражения. https://habr.com/ru/post/66021/
2. Что не так с std::visit в современном C++. https://habr.com/ru/post/415737/
3. Использование std::optional в C++17. https://habr.com/ru/post/372103/
4. Работа с кортежами C++ (std::tuple). Функции foreach, map и call. https://habr.com/ru/post/318236/

Используемые источники

1. *Stroustrup, Bjarne* The C++ Programming Language (Fourth Edition)
2. Lambda expressions
3. Undefined behavior
4. Fixed width integer types