

Лекция 1. Переход к современному C++11

ИУ8

October 11, 2017

Что нас ждёт

- Распределенный экзамен
- 8 лекций
- >8 семинаров
- >17 лабораторных работ
- Курсовая работа (подробности у семинаристов)

Программа курса

В ходе лекций будут затронуты следующие темы:

- Введение в C++11
- Семантика перемещения
- "Умные" указатели
- Многопоточность
- Сетевое взаимодействие
- и многое другое...

Дополнительные материалы для изучения

- 1 С. Мэйерс - Эффективное использование C++
- 2 С. Мэйерс - Эффективный и современный C++
- 3 Н. Джосаттис - C++ стандартная библиотека. Второе издание
- 4 Липпман - Язык программирования C++. Базовый курс
- 5 Бьерн Страуструп - Язык программирования C++
- 6 Б. Эккель, Ч. Эллисон - Философия C++
- 7 Уильямс - Параллельное программирование на C++ в действии
- 8 Александреску - Современное проектирование на C++
- 9 Саттер - Решение сложных задач на C++
- 10 Блог Алёны C++ - <http://alenacpp.blogspot.ru/>
- 11 ISO/IEC 14882:2011 C++11 или Working Draft, Standard for Programming Language C++
- 12 Standard C++ site - <https://isocpp.org/>

На текущей лекции рассмотрим

- Основные понятия: Ошибка времени компиляции и времени выполнения
- Основные понятия: Классы и объекты
- Основные понятия: Область видимости объектов
- Основные понятия: Деструкторы
- Основные понятия: Динамическая память и стек
- Основные понятия: Копирование объектов
- Основные понятия: Ссылки
- Современный C++: `auto`
- Современный C++: `using`
- Современный C++: `static_assert`
- Современный C++: `rangebase loop`
- STL: контейнеры
- STL: итераторы
- STL: алгоритмы
- STL: функторы и лямбда

Ошибка времени компиляции

Под **ошибкой времени компиляции** подразумевается ошибка, которая происходит **во время выполнения компиляции** исходного кода программы.

Пример

```
1 std::string s = 1234;
```

```
1 float foo() {  
2     flot number = 0  
3     return number;  
4 }
```

```
1 std::string s = "message";  
2 std::vector<s> vec();
```

Ошибка времени выполнения

Под **ошибкой времени выполнения** подразумевается ошибка, которая происходит **во время работы программы**. Такая ошибка приводит к аварийному завершению программы, либо к другим неопределенным последствиям.

Пример

```
1 int * ptr = nullptr;  
2 int value = *ptr;
```

Пример

```
1 int main() {  
2     throw std::logic_error();  
3 }
```

Основные понятия

Логические ошибки

Под **логической ошибкой** подразумевается допущенная разработчиком ошибка, которая приводит к непредвиденному поведению программы.

Пример

```
1 // pow2 returns x**2
2 float pow2(float x) {
3     auto result = x;
4     result *= result;
5     return x;
6 }
```

Пример

```
1 int i = 0;
2 std::cin >> i;
3 if ( i = 0 ) {
4     std::cout << "i eq 0" << std::endl;
5 } else {
6     std::cout << "i not eq 0" << std::endl;
7 }
```


Основные понятия

Классы и объекты

Класс – это тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д.

Описав класс, мы можем создать его экземпляр – объект. **Объект** – это уже конкретный представитель класса.

Пример

```
1 std::string; // std::string is class
2
3 struct Student { // Student is class
4     std::string name;
5     std::string last_name;
6     // ...
7 };
```

Пример

```
1 std::string str; // 'str' is object
2 Student Anna("Anna", "Pavlovna"); // 'Anna' is object
```

Объявление и определение

Объявление функции устанавливает ее имя, а также тип возвращаемого значения и список параметров. Функция должна быть объявлена перед вызовом.

В определение функции входит также понятие тела функции – набора инструкций, заключенных в фигурные скобки.

Пример

```
1 std::string get_name(Student const&);  
2  
3 int foo(){  
4     std::cout << get_name(Anna);  
5 }  
6  
7 std::string get_name(Student const& student) {  
8     return student.name;  
9 }
```

Область видимости объектов

Область видимости объекта - часть исходного кода программы, в которой к объекту можно непосредственно обращаться по его идентификатору.

Пример

```
1 int main() {  
2     int x = 0;  
3     int y = 0;  
4     std::cin >> x;  
5     std::cin >> y;  
6     if (x != y) {  
7         Point p = {x, y};  
8         std::cout << p.x, p.y;  
9     }  
10    // std::cout << p;  
11    std::cout << x << ", " << y;  
12 }
```

Конструкторы и деструкторы

Конструктор — это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.

Деструктор — специальный метод класса, который служит для уничтожения элементов класса.

Пример

```
1 struct Student {  
2     std::string name;  
3     Student(const std::string& student_name)  
4         : name(student_name)  
5     {}  
6  
7     ~Student() = default;  
8 };
```

Деструкторы вызываются в порядке, обратном порядку создания объектов.

Деструкторы

```
1 struct A {  
2     A() { std::cout << "A"; }  
3     ~A() { std::cout << "~A"; }  
4 };  
5  
6 struct B {  
7     A obj;  
8     B() { std::cout << "B"; }  
9     ~B() { std::cout << "~B"; }  
10 };  
11  
12 int main() {  
13     A a;  
14     std::cout << "---";  
15     { B b; }  
16     std::cout << "---";  
17 }
```

Вывод программы

```
1 A---AB~B~A---~A
```

Стек

Программы на C++ используют под локальные и временные переменные так называемую автоматическую память (automatic storage). Обычно автоматическая память реализована поверх стека программы, поэтому ее называют стековой. Ее большой плюс – выделение и освобождение памяти выполняется крайне быстро. Минус – относительно небольшой объем

Пример

```
1 struct point { float x, y; };  
2 int num = 0;  
3 point p{1.f, 2.f};  
4 char buff[1024] = {0};
```

Память для таких переменных выделяется в момент объявления переменной.

Память, выделенная для таких переменных, автоматически освобождается при выходе из области видимости переменной.

Динамическая память

Динамическая память выделяется и освобождается с помощью специальных инструкций. Это позволяет управлять временем жизни объектов.

Пример

```
1 struct point {  
2     float x, y;  
3 };  
4 int main() {  
5     point * p = new point{1.f, 2.f};  
6     delete p;  
7  
8     char * buff = new char[1024];  
9     delete[] buff;  
10 }
```

Ошибки работы с динамической памятью

Утечка памяти. Данная ошибка возникает в случае, когда разработчик не освобождает выделенную память.

```
1 int N = 1 << 32;  
2 while(N--) {  
3     int * array = new int[1024];  
4 }
```

Двойное освобождение памяти. Данная ошибка возникает, когда разработчик вызывает функцию освобождения памяти к участку памяти, который уже был освобожден.

```
1 int * array = new int[1024];  
2 int * ptr = array;  
3 delete[] ptr;  
4 delete[] array;
```


Ошибки работы с динамической памятью

Использование некорректной функции для освобождения памяти.

```
1 int * p = new int[100];  
2 delete p;  
3  
4 int * q = new int(100);  
5 delete[] q;  
6  
7 int * w = new int[100];  
8 free(w);
```

Присваивание объектов

Часто требуется **заменить** объект на копию другого объекта.
Реализуется такая операция с помощью оператора присваивания.

```
1 struct Student {
2     std::string name;
3
4     Student& operator=(const Student& rhs) {
5         if (&rhs == this) return *this;
6
7         DEBUG_LOG << "Assignment operator";
8         this->name = rhs.name;
9         return *this;
10    }
11 };
12
13 int main() {
14     Student Anna = {"Anna Pavlovna"};
15     Student Vasya = {"Prosto Vasya"};
16     Vasya = Anna; // Assignment operator
17 }
```

Копирование объектов

Часто требуется **создать** копию уже существующего объекта. Для этого используется операция копирования. Реализуется такая операция с помощью конструктора копирования.

```
1 struct Student {  
2     std::string name;  
3  
4     Student(const Student& rhs) {  
5         DEBUG_LOG << "copy ctor";  
6         this->name = rhs.name;  
7     }  
8 };  
9  
10 int main() {  
11     Student Anna = {"Anna Pavlovna"};  
12     Student copy(Anna); // copy ctor  
13 }
```

Нежелательное копирование объектов

Копирование объектов приводит к **созданию нового объекта**. Иногда создание нового объекта избыточно и достаточно использовать уже созданный объект.

Пример

```
1 void print_message(std::string message) {  
2     std::cout << message << std::endl;  
3 }  
4  
5 int main() {  
6     std::string message = "Some students are smart";  
7     print_message(message); // creating copy!  
8 }
```

Передача аргументов по ссылке

Ссылка - это псевдоним созданного объекта. Другими словами, ссылка - это новое имя для старого объекта.

Пример

```
1 void print_message(std::string& message) {  
2     std::cout << message << std::endl;  
3 }  
4  
5 int main() {  
6     std::string message = "Some students are smart";  
7     print_message(message); // using old object!  
8 }
```

Константная ссылка

Константная ссылка - это ссылка, которая не позволяет изменять объект. Таким образом, константная ссылка позволяет вызывать **только** константные методы классов. Если разработчик попытается изменить объект, используя константную ссылку, то произойдет **ошибка времени компиляции**.

Константные ссылки позволяют избежать логических ошибок разработчика.

Пример

```
1 void print_message(const std::string& message) {  
2     std::cout << message << std::endl;  
3     message = "All students are smart"; // compilation error  
4 }  
5  
6 int main() {  
7     std::string message = "Some students are smart";  
8     print_message(message); // using old object!  
9 }
```

Определение типа при объявлении

```
1 auto b = true; // b - bool
2 auto c = 'x'; // c - char
3 auto i = 123; // i - int
4 auto d = 1.2; // d - double
5 auto z = sqrt(d); // z - double
6 auto & ref = c; // ref - char&
7 auto * ptr = &d; // ptr - double*
8 const auto & cref = b; // cref - const bool&
9 const auto * cptr = &z; // cptr - const double*
```

Современный C++11

Без использования auto

```
1 // C++98
2 int x1 = 27;
3 int x2(27);
4 // C++11
5 int x3 = {27};
6 int x4{27};
```

С использованием auto

```
1 auto x1 = 27;
2 auto x2(27);
3 auto x3 = {27};
4 auto x4{27};
5 // auto x5 = {1, 2, 3.0};
6 // error: impossible to deduce T
```

Различие вывода template и auto

```
1 auto x = {11, 23, 9}; // x - std::initializer_list<int>
2
3 template<typename T>
4 void f(T param);
5 // f({11, 23, 19}); // error: impossible to deduce T
6
7 template<typename T>
8 void f(std::initializer_list<T> initList);
9
10 f({11, 23, 19});
11 // T is int.
12 // initList is std::initializer_list<int>
```


Новые спецификаторы

- `override`
- `final`
- `delete`
- `default`
- `noexcept`
- ссылочный квалификатор

Современный C++11

override

Компилятор, обнаружив **override**, проверяет существование **виртуального** метода с данной сигнатурой в базовом классе. Если же такого метода нет — возникает ошибка времени компиляции.

problem code

```
1 struct A {
2     virtual void
3     m1(const char * str) {
4         std::cout << "A::" << str;
5     }
6 };
7 struct B : public A {
8     virtual void
9     m1(char * str) {
10         std::cout << "B::" << str;
11     }
12 };
13 int main() {
14     A * ptr = new B();
15     ptr->m1("abc");
16 }
17 // std::cout << A::abc
```

with override

```
1 struct A {
2     virtual void
3     m1(const char * str) {
4         std::cout << "A::" << str;
5     }
6 };
7 struct B : public A {
8     virtual void
9     m1(char * str) override {
10         std::cout << "B::" << str;
11     }
12 };
13 int main() {
14     A * ptr = new B();
15     ptr->m1("abc");
16 }
17 // compilation error
```

Современный C++11

final

Спецификатор **final** позволяет запрещать в классах-наследниках переопределение определенных методов. Этот спецификатор также запрещает наследование от некоторого класса

Запрет наследования

```
1 struct A final {
2     virtual void m1() {
3         std::cout << "A::";
4     }
5 };
6
7 // a 'final' class type cannot be
8 // used as a base class
9 struct B : public A {
10     void m1() override {
11         std::cout << "B::";
12     }
13 };
14 int main() {
15     A * ptr = new B();
16     ptr->m1("abc");
```

Запрет переопределения

```
1 struct A {
2     virtual void m1() final {
3         std::cout << "A::";
4     }
5 };
6
7 struct B : public A {
8     // function declared as 'final'
9     // can't be overridden by B::m1
10     void m1() override {
11         std::cout << "B::";
12     }
13 };
14 int main() {
15     A * ptr = new B();
16     ptr->m1("abc");
```

delete

Спецификатор **delete** призван пометить те методы, работать с которыми нельзя. То есть, если программа ссылается явно или неявно на эту функцию, возникает ошибка на этапе компиляции. Запрещается даже создавать указатели на такие функции.

```
1 template <class charT, class traits = char_traits<charT>>
2 class basic_ios: public ios_base{
3     // ...
4 private:
5     basic_ios(const basic_ios&) = delete;
6     basic_ios& operator=(const basic_ios&) = delete;
7 };
```

default

Суть **default** заключается в том, что пользователь может указать компилятору реализовать тот или иной метод класса по умолчанию.

```
1 class Foo {  
2 public:  
3     Foo() = default;  
4     Foo(Foo &&) = default;  
5     Foo& operator=(Foo &&) = default;  
6     Foo(const Foo &) = default;  
7     Foo& operator=(const Foo &) = default;  
8     ~Foo() = default;  
9 };
```

- конструктор по-умолчанию
- конструктор копирования
- оператор присваивания
- деструктор
- конструктор перемещения (введен в C++11)
- оператор перемещения (введен в C++11)

using

using используется для определения псевдонима типа. Псевдоним типа является именем, ссылающимся на ранее определённый тип. Псевдоним шаблона является именем, ссылающимся на семейство типов

using vs typedef

```
1 // typedef <std::unordered_map<std::string, std::string> HashStrStr;  
2 using HashStrStr =  
3     std::unique_ptr<std::unordered_map<std::string, std::string>>;  
4  
5 //typedef void (*FPtr)(int, const HashStrStr&);  
6 using FPtr = void (*)(int, const HashStrStr&);
```

Преимущество using

Ключевое отличие: возможность использовать шаблоны

```
1 template<typename T>
2 struct MyAllocVec {
3     typedef std::vector<T, MyAlloc<T>> type;
4 };
5 // client code: MyAllocVec<T>::type myVec;
6 template<typename T>
7 class Widget {
8     typename MyAllocVec<T>::type vec;
9     // ...
10};
```

```
1 template<typename T>
2 using MyAllocVec = std::vector<T, MyAlloc<T>>;
3 // client code: MyAllocVec<T> myVec;
4 template<typename T>
5 class Widget {
6     MyAllocVec<T> vec;
7     // ...
8};
```

Современный C++11

Управление ошибками: static assertion

Если существует возможность обнаружить ошибку до выполнения программы, то такую ошибку предпочтительно выявлять на этапе компиляции. Для этого используется `static_assert`

`static_assert(C, M)` печатает сообщение *M* как ошибку компиляции, если условие *C* **НЕ** выполняется.

Пример

```
1 // Exp 1. check integer size
2 static_assert(4<=sizeof(int), "integers are too small");
3
4 // Exp 2. using static_assert
5 constexpr double C = 299792.458; // km/s
6 void f(double speed) {
7     // 160 km/h == 160.0/(60*60) km/s
8     const double local_max = 160.0/(60*60);
9     // error: speed must be a constant
10    static_assert(speed < C, "can't go that fast");
11    // OK
12    static_assert(local_max < C, "can't go that fast");
13    // ...
14 }
```


Ох уж этот C++98

```
1 std::vector<std::string> v = read_file();  
2 for (std::vector<int>::iterator it = v.begin();  
3     it != v.end(); ++it) {  
4     std::cout << *it << ' ';  
5 }
```

Уже можем использовать auto

```
1 std::vector<std::string> v = read_file();  
2 for (auto it = v.begin(); it != v.end(); ++it) {  
3     std::cout << *it << ' ';  
4 }
```

Да здравствует C++11!

```
1 std::vector<std::string> v = read_file();  
2 for (const auto& line : v) {  
3     std::cout << line << ' ';  
4 }
```

Цикл по набору значений

```
1 int v[] = {0,1,2,3,4,5,6,7,8,9};  
2 for (int x : v) // for each x in v  
3     cout << x << '\n';
```

Для изменения значений, используем ссылки

```
1 for (auto& x : v)  
2     ++x;
```

Библиотека STL

Библиотека стандартных шаблонов (STL) — набор **согласованных обобщённых** алгоритмов, контейнеров, средств доступа к их содержимому и различных вспомогательных функций в C++.
(википедия)

По факту же STL является неотъемлемой частью стандарта C++, без которой сложно представить разработку на C++.

Контейнеры

- `std::vector`
- `std::map`
- `std::set`
- `std::deque` (`std::queue`, `std::stack`)
- `std::list`
- `std::string`
- `std::forward_list`
- `std::unordered_map`
- `std::unordered_set`
- `std::array`

std::vector представляет собой динамический массив элементов. Элементы вектора располагаются непрерывно в оперативной памяти.

std::map - отсортированный ассоциативный контейнер, который содержит пары "ключ-значение" с неповторяющимися (уникальными) ключами.

std::unordered_map - неотсортированный ассоциативный контейнер, который содержит пары "ключ-значение" с неповторяющимися ключами.

std::set - это ассоциативный контейнер, который содержит упорядоченный набор уникальных объектов типа

std::list - представление двусвязного списка.

std::forward_list - представление односвязного списка.

std::deque - двусторонняя очередь. Представляет собой последовательный индексированный контейнер, который позволяет быстро вставлять и удалять элементы как с начала, так и с конца.

Итераторы

Итератор — интерфейс, предоставляющий доступ к элементам контейнера и навигацию по ним.

Итераторы в STL позволяют не вдаваться в тонкости реализации контейнера и предоставляют унифицированный доступ к элементам контейнеров.

Типы итераторов

- InputIterator
- ForwardIterator
- BidirectionalIterator
- RandomAccessIterator

Пример

```
1 std::map<std::string, std::string> dict;  
2 dict["sun"] = "Sonne";  
3 dict["car"] = "Auto";  
4 dict["dog"] = "Hund";  
5  
6 std::string ger_dog = dict["dog"];  
7  
8 auto it = dict.begin();  
9  
10 std::cout << it->first << ": " << it->second;
```

Алгоритмы

Алгоритмы STL решают типовые задачи, связанные с обработкой последовательности элементов одного типа (т.е. массивы и контейнеры). Последовательности определяются итераторами, указывающими на первый элемент и **на элемент за последним**. Результаты поиска также возвращаются в виде итератора. Если поиск не успешен, то результирующий итератор равен итератору указывающего на элемент за последним.

Пример

```
1 std::vector v = {10, -10, 0, 5, -5};  
2 std::sort(v.begin(), v.end());  
3 for(auto i : v)  
4     std::cout << i << ", ";
```

```
1 -10,-5,0,5,10
```


Функциональный объект (функтор) - объект, который можно использовать как функцию. В C++ для определения функтора достаточно описать класс, в котором переопределена операция `operator()`.

Пример

Подсчитать количество вызовов функции сравнения.

```
1 template<class T>
2 struct Less {
3     bool operator()(T a, T b) {
4         ++counter;
5         return a < b;
6     }
7     int counter = 0;
8 }
9
10 Less cmp;
11 cmp(10, 100);
12 cmp(20, 150);
13 std::cout << cmp.counter;
```

Примеры использования алгоритмов и функторов

Отсортировать массив чисел по возрастанию по модулю 5.

1 input: 13, 1, 9, 5, 22, 10

2 output: 5, 10, 1, 22, 13, 9

1 `template<class T>`

2 `struct mod_less{`

3 `bool operator()(T a, T b) const {`

4 `return (a % 5) < (b % 5);`

5 `}`

6 `};`

7 `std::vector<int> v = {13, 1, 9, 5, 22, 10};`

8 `std::sort(begin(v), end(v), mod_less<int>{});`

Примеры использования алгоритмов и функторов

Вычислить сумму квадратов массива

```
1 template<class T>
2 struct sum {
3     T operator()(T a, T b) const {
4         return (a*a) + (b*b);
5     }
6 };
7 std::vector<int> v = {13, 1, 9, 5, 22, 10};
8 int ret = std::accumulate(v.begin(), v.end(), 0, sum<int>{});
9 std::cout << ret;
```

Лямбда функции

Лямбда-выражение в C++11 — это удобный способ определения анонимного функтора непосредственно в месте его вызова или передачи его в функцию в качестве аргумента. Обычно лямбда-выражения используются для инкапсуляции нескольких строк кода, передаваемых алгоритмам или асинхронным методам.

Пример

Вычислить сумму квадратов массива

```
1 std::vector<int> v = {13, 1, 9, 5, 22, 10};  
2 int ret = std::accumulate(v.begin(), v.end(), 0,  
3     [](int a, int b) {  
4         return (a*a) + (b*b);  
5     }  
6 );  
7 std::cout << ret;
```

Пример

Отсортировать в порядке убывания абсолютных значений

```
1 input: -13, -1, -9, 5, -22, 10
2 output: -22, -13, 10, -9, 5, -1,
1 std::vector<int> v = {-13, -1, -9, 5, -22, 10};
2 std::sort(begin(v), end(v), [](auto a, auto b) {
3     return std::abs(a) > std::abs(b);
4 });
```

Самостоятельное изучение

- `initializer_list`
- `constexpr`
- `enum class`
- Пользовательские литералы
- Управление выравниванием объектов и запросы на выравнивание
- Ссылочный квалификатор

Список литературы

- С. Мэйерс - Эффективный и современный C++
- C++11 - the new ISO C++ standard
<http://www.stroustrup.com/C++11FAQ.html>
- <http://archive.kalnitsky.org/>
- <http://www.darkraha.com/rus/cpp/stl/stl10.php>
- <http://rstdn.org/forum/cpp/2933384.1>
- <https://msdn.microsoft.com/ru-ru/library/dd293608.aspx>