Введение в C++ 2011-2020 (стандарт ISO/IEC 14882:2011)

Вне рассмотрения в рамках курса остаются нововведения для работы с шаблонами:

- внешние шаблоны,
- альтернативный синтаксис шаблонных функций,
- расширение возможностей использования угловых скобок в шаблонах,
- ***typedef*** для шаблонов,
- шаблоны с переменным числом аргументов,
- статическая диагностика,
- регулярные выражения.

Не рассматриваются также новые понятия

- тривиального класса,
- класса с простым размещением,
- новшества в ограничениях для ***union***,
- новые строковые литералы,
- новые символьные типы ***char16_t*** и ***char32_t*** для хранения UTF-16 и UTF-32 символов,
- некоторое другое...

Введение в современный C++ (стандарт ISO/IEC 14882:2011)

Полностью стандарт 2011 поддерживают компиляторы g++ начиная с версии 4.7. …

Для компиляции программы в соответствии с конкретными стандартами в командной строке в качестве опции компилятору g++ можно указать, например:

g++ …… -std=c++17

По умолчанию используется компилятор, поддерживающий самый свежий стандарт C++, установленный на компьютере

rvalue- ссылки

В C++11 появился новый тип данных – **rvalue-ссылка**:

<тип> && <имя> = <временный объект>;

В C++11 можно использовать перегруженные функции для неконстантных временных объектов, обозначаемых посредством rvalue-ссылок.

Пример: ***class*** *A; … A a; …*

```
void f (A & x);                    ~  f (a);
void f (A && y);                   ~  f ( A() );
…
A && rr1 = A();
// A && rr2 = a; // Err!
int && n = 1+2;
n++;
…
```

Семантика переноса (Move semantics).

При создании/уничтожении временных объектов **неплоских** классов, как правило, требуется выделение-освобождение динамической памяти, что может отнимать много времени.

Однако, можно оптимизировать работу с временными объектами неплоских классов, если не освобождать их динамическую память, а просто перенаправить указатель на нее в объект, который **копирует значение временного объекта неплоского класса** (посредством поверхностного копирования). При этом после копирования надо обнулить соответствующие указатели у временного объекта, чтобы его деструктор ее не зачистил.

Это возможно сделать с помощью перегруженных конструктора копирования и операции присваивания с параметрами – **rvalue-ссылками**. Их называют конструктором переноса (*move constructor*) и операцией переноса (*move assignment operator*).

При этом компилятор сам выбирает нужный метод класса, если его параметром является временный объект.

Семантика переноса (Move semantics). Пример

```
#include <utility>      // <string_view>   c 2017
class Str {
        char * s;
        int len;
public:
        Str (const char * sss = NULL); // обычный конструктор неплоского класса
        Str (const Str &);              // традиционный конструктор копирования
        Str (Str && x) {                // move constructor
                s = x.s;
                x.s = NULL; // !!!
                len = x.len;
        }
        Str & operator = (const Str & x);     // обычная перегруженная операция =
        Str & operator = (Str && x) {         // move assignment operator
                std::swap(s, x.s); // !!!
                std::swap(len, x.len);
                return *this;
        }
        ~Str();                 // традиционный деструктор неплоского класса
        Str operator + ( Str  x);       ...
};
```

Семантика переноса (Move semantics).

Использование rvalue-ссылок в описании методов класса Str приведет к более эффективной работе, например, следующих фрагментов программы:

```
... Str a("abc"), b("def"), c;
    c = b+a; //  Str& operator= (Str &&);
...

Str f (Str a ) {
        Str b;  ... return a;
}
...  Str d = f (Str ("dd") ); ... //  Str (Str &&);
```

Обобщенные константные выражения.

Введено ключевое слово

constexpr,

которое указывает компилятору, что обозначаемое им выражение является константным, что в свою очередь позволяет компилятору вычислить его еще на этапе компиляции и использовать как константу.

Пример:

```
constexpr int give5 () {
        return 5;
}
int mas [give5 () + 7];    // создание массива из 12
                           // элементов, так можно с C++11.
```

Обобщенные константные выражения.

Однако, использование ***constexpr*** накладывает жесткие ограничения на функцию:

- она не может быть типа ***void***;
- тело функции должно быть вида ***return*** *выражение*;
- *выражение* должно быть ***constexpr*-выражением**,
- функция, специфицированная ***constexpr*** не может вызываться до ее определения.

В константных выражениях можно использовать не только переменные целого типа, но и переменные других числовых типов, перед определением которых стоит ***constexpr*.**

Пример:

constexpr double *a = 9.8;*

constexpr double *b = a/6;*

Вывод типов.

Описание ***явно инициализируемой*** переменной может содержать ключевое слово ***auto*****:** при этом типом созданной переменной будет тип инициализирующего выражения.

Пример:

Пусть *ft(....)* – шаблонная функция, которая возвращает значение шаблонного типа, тогда при описании

auto *var1 = ft(....);*

переменная *var1* будет иметь соответствующий шаблонный тип.

Возможно также:

auto *var2 = 5;* // *var2* имеет тип ***int***

Вывод типов.

Для определения типа выражения во время компиляции при описании переменных можно использовать ключевое слово **decltype**.

Пример:

```
int v1;
decltype (v1) v2 = 5;  // тип переменной v2 такой же, как у v1.
```

Вывод типов наиболее интересен при работе с шаблонами, а также для уменьшения избыточности кода.

Пример: Вместо

```
for(vector <int>::const_iterator itr = myvec.cbegin(); itr != myvec.cend(); ++itr)
  ...
```

можно написать:

```
for(auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr) ...
```

cbegin(); cend(),... – возвращают константный итератор, введены в C++11.

For-цикл по коллекции.

Введена новая форма цикла for, позволяющая автоматически осуществлять перебор элементов коллекции (массивы и любые другие коллекции, для которых определены функции *begin () и end()*).

Пример:

```
int arr [5] = {1, 2, 3, 4, 5};
for (int &x : arr) {
        x *= 2;
} ...
```

При этом каждый элемент массива увеличится вдвое.

```
for (int x : arr) {
        cout << x << ' ';
} ...
```

Улучшение конструкторов объектов.

В отличие от старого стандарта стандарты с C++11 позволяют вызывать одни конструкторы класса (так называемые делегирующие конструкторы) из других, что в целом позволяет избежать дублирования кода.

Пример:

```
class A {
        int n;
public:
        A (int x) : n (x) { }
        A ( ) : A (14) { }
};
```

Замечание: Если до конца проработал хотя бы один делегирующий конструктор, его объект уже считается **полностью созданным**. (*Однако, объекты производного класса начнут конструироваться только после выполнения всех конструкторов (основного и его делегирующих) базовых классов*).

Инициализация в современном C++

Стало возможно инициализировать члены-данные класса в области их объявления в классе.

Пример:

```
struct B {
        int t {1};
        B (int k ): t(k) { }
        B(){ }
};
```

```
struct A {
        int  n = 14;
        B b = B(3);
        int m [4] = {1,2,3,4};
        A(int x=0, int y=0) n(x), b(y) { }
        static const int cc = 8; //для
}; // интегральных типов стат. констант
```

```
int  main() {
        A a1{7,9}, a2;
        std::cout << a1.n << a1.b.t << a2.n << a1.m[2] << std::endl     // 7903
        return 0;
}
```

```
std::vector<int> vec = {0, 1, 2, 3, 4};
std::vector<int> v(3, 0); // вектор содержит 0, 0, 0
std::vector<int> v{3, 0}; // вектор содержит 3, 0
```

Явное замещение виртуальных функций и финальность .

В C++11 добавлена возможность (с помощью спецификатора ***override***) отследить ситуации, когда виртуальная функция в базовом классе и в производных классах имеет разные прототипы, например, в результате случайной ошибки (что приводит к тому, что механизм виртуальности для такой функции работать не будет).

Кроме того, введен спецификатор ***final*****,** который обозначает следующее:

- *в описании классов* - то, что они не могут быть базовыми для новых классов,
- *в описании виртуальных функций* — то, что возможные производные классы от рассматриваемого не могут иметь виртуальные функции, которые бы замещали финальные функции.

Замечание: спецификаторы ***override и final*** имеют специальные значения только в приведенных ниже ситуациях, в остальных случаях они могут использоваться как обычные идентификаторы.

Явное замещение виртуальных функций и финальность. Пример

```
struct B {
   virtual void  some_func ();
   virtual void  f (int);
   virtual void g () const;
};

struct D1 : public B {
   virtual void  sone_func ()  override; // Err: нет такой функции в B
   virtual void  f (int) override;            // OK!
   virtual void  f (long) override;       // Err: несоответствие типа параметра
   virtual void  f (int) const  override;
                                  // Err: несоответствие квалификации функции
   virtual int  f (int) override;     // Err: несоответствие типа результата
   virtual void g () const final;   // OK!
   virtual void  g (long);            // OK: новая виртуальная функция
};
```

Явное замещение виртуальных функций и финальность. Пример

```
struct D2 : D1 {            // см. предыдущий слайд

        virtual void g () const;    // Err: замещение финальной функции
};

struct F final {
    int x,y;
};

struct D : F {     // Err: наследование от  финального класса
    int z;
};
```

Константа нулевого указателя.

В C++ NULL — это константа 0, что может привести к нежелательному результату при перегрузке функций:

```
void f (char *);
void f (int);
```

При обращении *f (NULL)* будет вызвана *f (**int**);* , что, вероятно, не совпадает с планами программиста.

С C++11 введено новое ключевое слово ***nullptr*** для описания константы нулевого указателя:

```
std::nullptr_t nullptr;
```

где тип *nullptr_t* можно неявно конвертировать в тип любого указателя и сравнивать с любым указателем.

Неявная конверсия *nullptr_t* в целочисленный тип **недопустима**, за исключением ***bool*** (в целях совместимости).

Для обратной совместимости константа 0 также может использоваться в качестве нулевого указателя.

Константа нулевого указателя.

Пример:

```
char * pc = nullptr;     // OK!

int * pi = nullptr;      // OK!

bool   b = nullptr;      // OK: b = false;

int  i = nullptr;        // Err!

f (nullptr);       // вызывается  f(char*)  а не  f(int).
```

Перечисления со строгой типизацией.

В старом стандарте C++ :

- перечислимый тип данных фактически совпадает с целым типом,
- если перечисления заданы в одной области видимости, то имена их констант не могут совпадать.

С C++11 наряду с обычным перечислением предложен также способ задания перечислений, позволяющий избежать указанных недостатков. Для этого надо использовать объявление ***enum class*** (или, как синоним, ***enum struct***). Например,

```
enum class E { V1, V2, V3 = 100, V4 /*101*/};
```

Элементы такого перечисления нельзя неявно преобразовать в целые числа (выражение *E:: V4 == 101* приведет к ошибке компиляции).

Перечисления со строгой типизацией.

С C++11 тип констант перечислимого типа не обязательно ***int*** (только по умолчанию), его можно задать явно следующим образом:

```
enum class E2 : unsigned int  { V1, V2 };
```

// значение *E2:: V1* определено, а *V1* – не определено.

Или:

```
enum  E3 : unsigned long { V1 = 1, V2 };
```

// в целях обеспечения обратной совместимости определены и значение E3:: V1 , и V1.

С C++11 возможно предварительное **объявление** перечислений, но только если указан размер перечисления (явно или неявно):

```
enum E1;                    // Err: низлежащий тип не определен
enum E2 : unsigned int;     // OK!
enum  class E3 ;            //  OK: низлежащий тип  int
enum class E4 : unsigned long;      // OK!
enum E2 : unsigned short;   // Err: E2 ранее объявлен
                            // с другим низлежащим типом.
```

sizeof — для членов-данных классов без создания объектов.

В C++11 разрешено применять операцию **sizeof** к членам-данным классов независимо от объектов классов.

Пример:

```
struct A {
        some_type a;
};
... sizeof (A::a) ... // OK!
```

Кроме того, с C++11 узаконен тип ***long long int*** .

explicit — для функций преобразования

Начиная с C++11 ключевое слово ***explicit*** может применяться и к функциям преобразования типа, запрещая их неявный вызов:

```
class Ex_explicit {
public:
        int n;
        explicit Ex_explicit (int a = 0) : n(a) { }
        explicit operator bool() const { return true; }
};
 int main() {
            Ex_explicit  se7 = 1;  // Err - неявный вызов конструктора преобразования
        Ex_explicit  se7(1);
        bool be7 = se7; // Err – неявный вызов функции преобразования
        bool be7 = static_cast <bool> (se7);
        return 0;
}
```

delete — для запрещения ненужных методов. Пример

```
struct ex {
        ex () {}
        ex (const ex &) = delete;
        ex & operator=(const ex &) = delete;
        void f(int) = delete;
        void f(double f) {cout << "ex::f(double)\n"; }
};

int main() {
        ex  d1, d2;
        ex  d3 = d1;            // Err – вызов запрещенного метода
        d1.f(1);                // Err – вызов запрещенного метода
        d1.f(1.5);
        d1.f('a');              // Err – вызов запрещенного метода
        d2 = d1;                // Err – вызов запрещенного метода
        return 0;
}
```

default — для автоматической генерации специальных методов класса. Пример

```
class FFF {
        float x;
public:
        FFF() = default;
        FFF(float  y) : x(y) { }
        FFF (const  FFF & ) = default;
        FFF & operator= (const  FFF &) = default;
        FFF (FFF && ) = default;
        FFF & operator= (FFF &&) = default;
        virtual void f() {std::cout << "Default\n";}
        virtual ~FFF() = default;
};

int main () {
        FFF a, b(1), c = b;
}
```

Правила автоматической генерации специальных методов класса

Лямбда-функции

Лямбда-функции — анонимные функции, которые можно определить в любом месте программы, где требуется указать функцию.

Ex1: ***auto*** *mylambda =* *[ ]( )->int{ std::cout << "Hello, lambda!" << std::endl; return 1;};* *mylambda ();*

[ ] — «захват», список переменных из текущей области видимости,
[x] — по значению, запрещено менять в теле лямбда-функции,
[&x] — по ссылке, можно изменять в теле лямбда-функции,
( ) — список формальных параметров функции

Ex2:

```
int main() {
    int n =2;
    std::vector< int > v = { 2, 4, 5, 6, 7, 9, 11, 14 };
    auto newend = std::remove_if ( v.begin(), v.end(), [n]( int x ) { return x % n == 0; } );
    std::erase (newend, v.end());
    std::for_each( v.begin(), v.end(), []( int x ) { std::cout << x << " "; } ); // 5 7 9 11
}
```