Введение в C++11 (стандарт ISO/IEC 14882:2011)

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

- ведение понятий лямбда-функций и выражений,
- внешние шаблоны,
- альтернативный синтаксис шаблонных функций,
- расширение возможностей использования угловых скобок в шаблонах,
- typedef для шаблонов,
- шаблоны с переменным числом аргументов,
- статическая диагностика,
- изменения в STL,
- регулярные выражения.

Не рассматриваются также новые понятия

- ✓ тривиального класса,
- ✓ класса с простым размещением,
- ✓ ***explicit*** перед функциями преобразования,
- ✓ 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 C++11
(стандарт ISO/IEC 14882:2011)

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

Для компиляции программы в соответствии с новым стандартом в командной строке в качестве опции компилятору g++ надо указать:

-std=c++0x (или в новых версиях -std=c++11) :

g++ …… -std=c++0x
(или g++ …… -std=c++11)

Введение в C++11

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.

Например: **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 конструктор 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:

```
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
                s = x.s;
                x.s = NULL; // !!!
                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
                               // элемнтов, так можно в 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 константой,
- 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) …
```

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 новый стандарт C++11 позволяет 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) { }
};
```

Улучшение конструкторов объектов.

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

Пример:

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

Любой конструктор класса A будет инициализировать n значением 14, если сам не присвоит n другое значение.

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.

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.

В 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 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 C++ :

✓перечислимый тип данных фактически совпадает с целым типом,

✓если перечисления заданы в одной области видимости, то имена их констант не могут совпадать.

В 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.

В 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.
```

В 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, в C++11 узаконен тип ***long long int*** .