Стандарт c++11

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)

r-value ссылки

rvalue и lvalue - это, в первую очередь, свойства выражений.

Если можно получить адрес выражения, то оно lvalue.

Если нельзя - rvalue.

```
int foo();
int& bar();
int main(){
    int i = 0;
    &++i;//Ok, lvalue
    &i++;//error C2102: '&' requires l-value(VC++ 2010)
    &foo();//error C2102: '&' requires l-value(VC++ 2010)
    &bar();//Ok, lvalue
}
int foo(){    return 0;}
int& bar(){
    static int value = 0;
    return value;
}
```

rvalue ссылки

Меняется синтаксис объявления. Для rvalue:

<тип> && <имя ссылки>;

Например:

```
A a;

A&& a_ref = a;  // это rvalue ссылка
```

Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

```
A&  a_ref3 = A();  // Ошибка!

A&& a_ref4 = A();  // Ok
```

Семантика переноса (перемещения, move semantics)

По стандарту C++ временный объект можно передавать в функцию только по константной ссылке.

Но:

- непонятно, временный это объект или константный;
- копирование для неплоских классов может занять много времени.

```
String concat (const String & s1, const String & s2) {
    String res=s1;
    return res+s2;
}
String one(“abc”), two(“def”);
String s=concat (one, two);
```

Сколько раз создается и уничтожается копия объекта res?
А сколько раз она действительно необходима?

По стандарту C++ временный объект можно передавать в функцию только по константной ссылке.

Но:

- непонятно, временный это объект или константный;
- копирование для неплоских классов может занять много времени.

Может, не копировать объект, а просто переместить его?

Именно для этого в языке вводятся rvalue-ссылки.

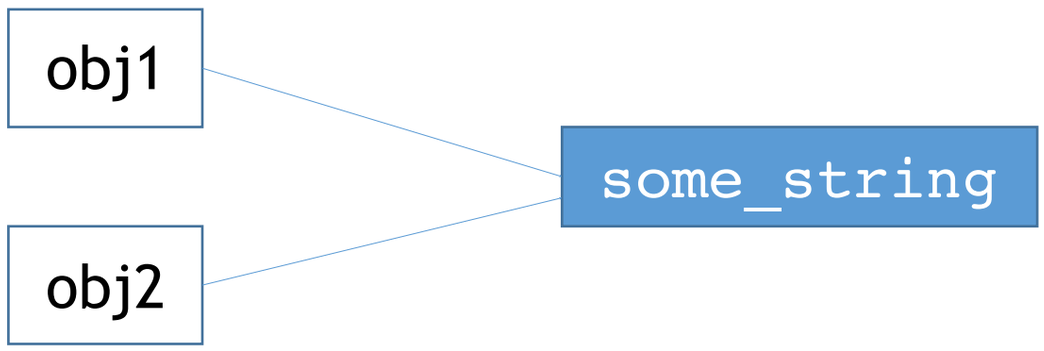
Семантика переноса

- Еще два специфических метода класса, которые появились в стандарте C++11 -- конструктор переноса и оператор переноса.
- Оба этих метода позволяют не создавать, явно выделяя память, копию старого объекта, а «своровать» данные у того объекта, на базе которого вызываются.
- При этом значения соответствующих полей базового объекта устанавливаются в нули.

- Прототипы этих методов таковы:

```
MyClass (MyClass && obj);
MyClass & operator= (MyClass && obj);
```

- Для методов, обладающих семантикой переноса в языке C++ свойственна та картина, о которой мы говорили, когда обсуждали недостатки автоматически сгенерированного конструктора копирования:



- Однако при этом один из этих объектов обычно временный, и его поле теряет связь с общими данными сразу после переноса.

На примере неплоского класса: класс String

```
class String {
    char * p;
    int len;
public:
    String (const Str &);
    String (String && x) {                  // move constructor
        p  = x.p;
        x.p = nullptr;                      // !!!
        len = p.len;
    }
    String & operator = (const String & x);
    String & operator = (String && x) { // move assignment operator
        p = x.p;
        x.p = nullptr;                      // !!!
        len = x.len;
        return *this;
    }
    ...
};
```

Пример. Какой оператор будет вызван?

```
...
String a(“abc”), b(“def”), c;
c = b+a;
...
```

Пример. Какой оператор будет вызван?

```
...
String a(“abc”), b(“def”), c;
c = b+a; // String operator= (String &&);
...
```

Пример. Какой оператор будет вызван?

```
String f (String a ) {
     String b;
     ...
     return a;
}
...
String d = f (String (“dd”) );
...
```

Пример. Какой оператор будет вызван?

```
String f (String a ) {
     String b;
     ...
     return a;
}
...
String d = f (String (“dd”) ); //String (String &&);
...
```

Об умолчаниях

- Операция и конструктор переноса генерируются по умолчанию, если выполнены все следующие условия:
  - в классе не объявлен пользователем конструктор копирования;
  - в классе не объявлен пользователем оператор присваивания;
  - в классе не объявлен пользователем декструктор.
  - однако если объявлен конструктор переноса, то оператор не генерируется, и наоборот.

- Разработчики стандарта рекомендуют всегда явно описывать все методы класса, связанные с переносом и копированием, а также деструктор.

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;
```

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!
int i = NULL;             // вообще говоря, корректно

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

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 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.

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;
};
```

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*** .