**[C++11 常用语法](https://www.cnblogs.com/benxintuzi/p/4613144.html)**

**1 新类型**

C++ 11新增了long long和unsigned long long，以支持64bit宽度；

新增char16\_t和char32\_t以支持16位和32位字符表示；

增加了“原始”字符串。

**2 初始化**

C++ 11扩展了大括号{}的适用范围，既可以用于基本类型，也可以用于自定义类型：

int x = {5};

couble y{3.5};

short quar[5]{1, 2, 3, 4, 5};

int\* p = new int[5]{1, 2, 3, 4, 5};

创建对象时，也可以使用大括号链表来调用构造函数：

class Stump

{

Public:

    Stump(int r, double w) : roots(r), weight(w){}

Private:

    int roots;

    double weight;

};

Stump s1(3, 4.5); // old style

Stump s2{3, 4.5}; // new style

Stump s3 = {3, 4.5};     // new style

注意：使用初始化列表可以防止向下转型，如：

char c1 = 1.57e27;   // double-to-char, undefined behavior

char c1{1.57e27}; // double-to-char, compile error

std::initializer\_list可以作为函数参数，如果一个类中的构造函数使用了这种方法，则其他函数将不能再使用其作为参数，这是为了防止调用二义性。

Initializer\_list模板类提供了两个成员函数begin()和end()，用来指定列表的范围，如：

double sum(std::initializer\_list<double> il)

{

    double total = 0;

    for(auto p = il.begin(); p != il.end(); p++)

       total += \*p;

    return total;

}

**double total = sum({1, 2.5, 8.3, 2.2, 1.0});**

**3 声明**

C++提供了多种简化声明的方式，尤其在使用模板时更加方便。

**1 auto**：用来实现自动类型推断，如：

auto i = 112;     // i is type int

auto pt = &i;     // pt is type int\*

double fm(double, int);

auto pf = fm;     // pf if type double(\*)(double, int);

用于模板时形式简洁，如：

for(std::initializer\_list<double>::iterator p = il.begin(); p != il.end(); p++)

改写为：

for(auto p = il.begin(); p != il.end(); p++)

**2 decltype：**将变量的类型指定为表达式的类型。

decltyte (x) y; 将y设为与x相同的类型，其中x是一个表达式。

**3 返回类型后置：**在函数名和参数列表后面指定返回类型。

double f1(double int); 　　　　　　　// return double type

auto f2(double, int) -> double; 　　// new syntax, return double type

如果结合上模板表示返回类型，那就更好了，如下：

template<typename T, typename U>

auto eff(T t, U u) -> decltype(T \* U)

{

}

**4 模板别名：using =**

C++中创建别名一般用typedef，如：

typedef std::vector<std::string>::iterator itType;

还可以这样做：

using itType = std::vector<std::string>::iterator;

二者的差别在于，using可以使模板具体化，如：

using arr = std::array<T, **12**>;，此时typedef不行。

**5 nullptr：**空指针

之前，C++使用0表示空指针，同样的0既可以表示整型，又可以表示空指针，比较混乱；新增的nullptr是指针类型，不能转换为整型。为了兼容性，C++目前仍然允许0表示空指针，即nullptr == 0的结果为true。

**4 智能指针**

C++ 11摒弃了auto\_ptr，新增了三种：unique\_ptr、shared\_ptr、weak\_ptr。

**5 异常**

之前C++的语法中可以指出函数可能引发哪些异常，如：

void f1(int) throw(bad\_alloc);  // 可能抛出bad\_alloc异常

void f2(long long) throw();     // 不抛异常

**C++摒弃了异常规范，新增了如下规则：**

**Void f3(short, short) noexcept; // 不抛异常**

**6 作用域内枚举**

传统的C++枚举的作用域在所属的域内，就是说同一作用域内不能出现两个同名的枚举变量。

C++ 11新增了一种枚举，使用class或者struct定义：

enum Old{yes, no};   // old style

enum class New{yes, no}; // new style

enum struct New{yes, no};   // new style

由于允许同名存在，因此引用时需要使用枚举名限定：New::yes。

**7 对类的修改**

在扩展类的设计方面，C++ 11很多改进，比如允许构造函数被继承、彼此调用、移动构造函数、移动赋值运算符等。

**7.1 显示转换运算符**

早期的C++会导致自动类型转换，比如：

class Plebe

{

    Plebe(int);

    explicit Plebe(double);

    ...

};

Plebe a, b;

**a = 5;        // 发生隐式类型转换，实则调用Plebe(5);**

**b = 0.5;      // 不允许**

b = Plebe(0.5);   // 允许

C++ 11扩展了explicit，使得可以如下这样做：主要是针对转换函数：

比如：

operator int() const;

explicit operator double() const;

int n = a;    // allow

double x = b;     // not allow

x = double(b);    // allow

**7.2 类内成员初始化**

class Session

{

    int mem = 10;

    double mem2{2.35};

    ...

};

**8 模板和STL方面**

为了改善模板和标准化方面的易用性，C++ 11做了多个改进：

**8.1 改进的for循环**

double prices[5] = {1, 2, 3, 4, 5};

for(double x : prices)

for(auto x : prices)

如果要在循环中修改数组或容器中的每个元素，可以使用引用：

for(auto& x : prices)

**8.2 新增的STL容器**

C++ 11新增了forward\_list、unordered\_map、unordered\_multimap、unordered\_set、unordered\_multiset。

新增了模板array，实例化时指定元素类型和个数：std::array<int, 10> ar; // 10个int

**8.3 新增STL方法**

cbegin()、cend()、crbegin()、crend()。这几个方法将容器元素视为const。

**8.4 valarray升级版**

对于valarray模板，C++ 11新增了两个方法begin()和end()。

**8.5 摒弃了export**

C++ 98增加的关键字export可以让程序员将模板定义放在接口文件中，实现文件中放置方法体，但是实践证明这一点也不现实，因此C++ 11把它摒弃了。

**8.6 尖括号**

旧时的C++要求在定义嵌套的模板时，两个尖括号之间必须有空格，比如：vector<list<int> >，但是C++ 11中不再需要这样了，比如vector<list<int>>也可以通过的。

**9 右值引用**

C++新增了右值引用，使用&&表示。【相对于左值，右值表示字面常量、表达式、函数的非引用返回值等】

如：

int&& r1 = 12;

int x = 5;

int y = 8;

int&& r2 = x + y;

我们可以通过r1来修改12，很方便。

**10 移动语义和右值引用**

**10.1 为何需要移动语义**

移动语义就是为了避免多余的复制工作，就是说与其复制，还不如将源地址传给使用者，因为有时复制工作确是没什么用。

要实现移动语义，需要采用某种措施让编译器知道到底什么情况下需要复制，什么情况下不必复制。这时，右值引用就可以配上用场了。

传统的复制构造函数执行深复制，并且不改变实参，因此，参数为const类型；

移动构造函数只是更改了引用记录，并且可能会改变实参，因此，参数必须是非const类型；

如下实例：

class Use

{

public:

    Use();

    Use(const Use& f);   // copy constructor

    Use(Use&& f);        // move constructor

    ...

private:

int n;

char\* pc;

};

...

Use::Use(const Use& f) : n(f.n) // 深复制

{

    pc = new char[n];

    for(int i = 0; i < n; i++)

       pc[i] = f.pc[i];

}

Use::Use(Use&& f) : n(f.n)

{

    pc = f.pc;    // 移动构造，转移控制权

    f.pc = nullptr;

    f.n = 0;

}

**10.2 移动构造解析**

虽然移动构造定义好了，但是如何调用呢，也就是说什么情况下才会发生移动构造呢？

必须**使用右值引用调用**：

Use two = one;           // match Use::Use(const Use&);

Use four(one + three);  // match Use::Use(Use&&);

因为one是左值，而one + three是右值。

移动赋值情况类似，不记录了。

**10.3 强制移动**

移动构造和移动赋值使用右值引用，如果需要让他们操作左值呢？

答案是使用static\_cast<>将对象强制转换为Use&&即可。不过在C++ 11种提供了更便捷的操作，在头文件utility中声明的函数std::move(...)。

如果使用了std::move()函数调用，但是类中却没有定义移动相关函数，那么编译器会调用传统版本的移动构造函数和移动赋值操作符。

**11 新的类功能**

假设你为类定义了构造函数,那么类就不会自动提供默认的构造函数了,然而,如果你仍然想使用类提供的默认版本,那么可以使用default关键字：

class Some

{

public:

    Some(Some&&);

**Some() = default; // use default constructor**

    ...

};

相反地，如果要禁用编译器提供的默认函数，可以使用delete:

class Some

{

public:

    Some(Some&&);

    Some() = default; // use default constructor

**Some(const Some&) = delete; //disable copy constructor**

    ...

};

当然要想禁用某个编译器提供的函数也可以显式声明为private，但是使用delete更方便且不易出错。

注意：default关键字只能用于6个特殊函数，而delete却能够用于任何成员函数。

**委托构造**

如果一个类包含多个构造函数，C++ 11允许在一个构造函数中的定义中使用另一个构造函数，但这必须通过初始化列表进行操作，如下：

class Notes

{

    int k; double x; string s;

public:

    Notes(int kk, double xx, string ss) : k(kk), x(xx), s(ss){ }

    Notes(int kk) : Notes(0, 0.5, “benxin”){ k = kk; }

    ...

};

**继承构造**

C++ 11允许派生类继承基类的构造函数。C++ 98提供了一种让某个名称空间中的同名重载函数都可用的语法，如下：

namespace Box

{

    int fn(){}

    int fn(int){}

    int fn(double){}

}

using Box::fn;该语句使得fn的所有重载版本都可用。我们一般使用这种方法在派生类中调用基类的同名函数。之所以提供这种语法，就是因为覆盖是以函数名为基础的，不论参数是否对应都将被覆盖。

C++ 11将这种语法用于构造函数中，使得派生类将继承基类的构造函数（默认构造函数、复制构造函数、移动构造函数除外）。

**管理虚方法：override和final**

传统的虚函数是为了实现多态调用，但这必须是派生类与基类的虚函数签名完全一致的情况下才会发生多态，如果不一致，假设如下：

[复制代码](javascript:void(0);)

1 class Base

2 {

3 public:

4 virtual void **Fn()**{ cout << "Base::Fn()" << endl; }

5 };

6

7 class Derived : public Base

8 {

9 public:

10 virtual void **Fn(int)**{ cout << "Derived::Fn()" << endl; }

11 };

12

13 int main()

14 {

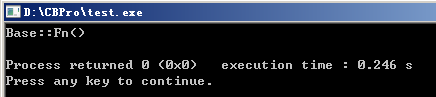
15 **Base\* d = new Derived;**

**16 d->Fn();**

17

18 return 0;

19 }



[复制代码](javascript:void(0);)

结果如上，编译器竟然会根据指针的静态类型发生了调用，并无多态发生。

好在C++ 11提供了override指出了该虚函数是为了覆盖基类的虚函数而存在的，此时如果不小心与基类中的虚函数不一致了，那么编译器不会让你通过的。

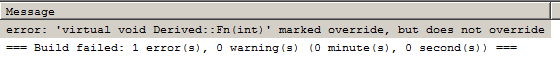
1 class Derived : public Base

2 {

3 public:

4 virtual void Fn(int) **override**{ cout << "Derived::Fn()" << endl; }

5 };



如果想禁止派生类覆盖基类的虚函数，可在基类的虚函数参数列表后加上final。

[复制代码](javascript:void(0);)

1 class Base

2 {

3 public:

4 virtual void Fn() **final**{ cout << "Base::Fn()" << endl; }

5 };

6

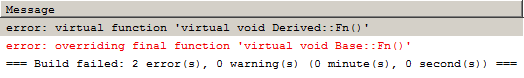
7 class Derived : public Base

8 {

9 public:

10 virtual void Fn() **override**{ cout << "Derived::Fn()" << endl; }

11 };



[复制代码](javascript:void(0);)

最后需要指出的是：override和final并非关键字，而是具有特殊含义的标识符，编译器会根据上下文确定其到底是否代表特殊性。

**12 Lambda函数**

如下示例使用了三种方法给STL算法传递信息：**函数指针、函数符、lambda函数**。出于方便性，我们将其统称为函数对象。

假设生成一个包含1000个数的容器，并判断其中有多少个可以被3整除，有多少个可以被13整除：

vector<int> numbers(1000);

generate(numbers.begin(), numbers.end(), **rand**);

说明：generate函数前两个参数指定容器区间，并将每个元素设置为第三个参数返回的值。

bool f3(int x){ return x % 3 == 0; }

bool f13(int x){ return x % 13 == 0; }

接下来可以借助于count\_if函数，该函数前两个参数也是指定区间，第三个参数是一个返回true或false的函数对象，该函数计算使得返回true的元素个数。

int count = count\_if(numbers.begin(), numbers.end(), **f3**);

如上述所示，rand和f3都是作为**函数指针**传递过去的。

下面看一下**函数符**如何使用：

我们知道函数符是一个类对象，主要利用重载()操作符来完成任务：

class f\_mod

{

public:

    f\_mod(int d = 1) : dv(d){ }

    bool operator()(int x){ return x % dv == 0; }

private:

    int dv;

};

创建对象：

f\_mod obj(3);     　　// set dv to 3

使用：

bool rv = obj(7); 　　// **same as obj.operator()(7);**

                     　　　　// return 7 % 3 == 0

因此：

int count = count\_if(numbers.begin(), numbers.end(), **f\_mod(3))**;

最后看看如何使用**lambda**：C++ 11中，对于接受函数指针或函数符的函数，也可以使用**匿名函数定义(lambda)**作为其参数：

bool f3(int x){ return x % 3 == 0; }

上述函数f3用lambda来表示即为：

[](int x){ return x % 3 == 0; }

**lambda和一般函数的区别如下：**

使用[]代替函数名；没有返回类型。返回类型相当于使用decltype推断而得。如果表达式中不包括返回语句，则推断出类型为void。**但要特别注意，仅当lambda表达式中仅包含一条返回语句时，自动推断类型才起作用；否则，需要使用新增的返回类型后置语法，**如下：

[](double x)**-> double**{ int y = x; return x – y; }

int count = count\_if(numbers.begin(), numbers.end(), [](int x){ return x % 3 == 0; });

**我们也可以给lambda指定一个名称，如：**

**auto mod3 = [](int x){ return x % 3 == 0; }**

int count = count\_if(numbers.begin(), numbers.end(), **mod3**);

当然，我们也可以像普通函数一样使用lambda，如：mod3(z); ---> return z % 3 == 0;

还有，**lambda还可以操作变量**，如：

[count]：表示以传值方式访问变量；

[&count]：表示以引用方式访问变量；

[count1, &count2]：以传值方式访问count1，以引用方式访问count2;

[&, count]：表示以传值方式访问count，以引用方式访问其他变量；

[=]：表示以传值方式访问所有变量；

[&]：表示以引用方式访问所有变量；

[复制代码](javascript:void(0);)

1 　　int count3 = 0;

2 int count13 = 0;

3 vector<int> numbers(100);

4 generate(numbers.begin(), numbers.end(), rand);

5 for\_each(numbers.begin(), numbers.end(), **[&]**(int x){ count3 += x % 3 == 　　　　　　　　0; count13 += x % 13 == 0;});

6

7 　　cout << count3 << " " << count13 << endl;

https://images0.cnblogs.com/blog2015/759985/201507/011353012816152.png

[复制代码](javascript:void(0);)

**注意：**

C++引入lambda的主要目的是为了简化程序的编写，**典型的lambda是测试表达式或者比较表达式**，因为它们一般只有一条返回语句，因此可以使用返回类型自动推断。

**13 可变参数模板**

C++ 11提供了一个用省略号表示的元运算符,使得可以声明表示模板参数列表,其语法如下:

template<typename... Args>

void show(Args... args){ }

其中Args表示模板参数列表，args表示函数参数列表。

这里有一个问题，我们如何访问具体的某个参数呢？使用args[2]吗?答案是否定的，我们在可变参数中不能使用索引来访问，这里需要递归地访问每个参数：

template<typename T, typename... Args>

void **show(**T value, Args... args)

{

    cout << value << “, ”;

**show(args...);**

}

每次调用show，可变参数将减少一个，直到调用show()时，调用参数为空，不接受，则退出递归。

**14 断言**

C++提供了调试工具assert，这是一个宏，用于在**运行阶段**对断言进行检查，如果为true，则显示一条消息，否则调用abort();

C++ 11新增了关键字static\_assert，可用于在**编译阶段**对断言进行测试。