**C++ 11**

C++11,之前被称作C++0x，即ISO/IEC 14882:2011，是目前的C++编程语言的正式标准。它取代第二版标准ISO/IEC 14882:2003(第一版ISO/IEC 14882:1998发布于1998年，第二版于2003年发布，分别通称C++98以及C++03，两者差异很小)。新的标准包含了几个核心语言增加的新特性，而且扩展C++标准程序库，并入了大部分的C++ Technical Report 1程序库(数学的特殊函数除外)。最新的消息被公布在 ISO C++ 委员会网站(英文)。

ISO／IEC JTC1/SC22/WG21 C++ 标准委员会计划在2010年8月之前完成对最终委员会草案的投票，以及于2011年3月召开的标准会议完成国际标准的最终草案。然而，WG21预期ISO将要花费六个月到一年的时间才能正式发布新的C++标准。为了能够如期完成，委员会决定致力于直至2006年为止的提案，忽略新的提案。最终,于2011年8月12日公布，并于2011年9月出版。2012年2月28日的国际标准草案(N3376)是最接近于现行标准的草案，差异仅有编辑上的修正。

像C++这样的编程语言，通过一种演化的过程来发展其定义。这个过程不可避免地将引发与现有代码的兼容问题。不过根据Bjarne Stroustrup(C++的创始人,标准委员会的一员)表示，新的标准将几乎100%兼容现有标准。

* **auto and decltype**

C++11引入了auto和decltype关键字，使用他们可以在编译期就推导出变量或者表达式的类型，方便开发者编码也简化了代码。

**auto** a **=** 10; *// 10是int型，可以自动推导出a是int*

cont **int** **&**i **=** 1;

**int** a **=** 2;

**decltype**(i) b **=** 2; *// b是const int&*

auto不能用作函数参数

在类中auto不能用作非静态成员变量

auto不能定义数组，可以定义指针

C++11 中 auto 和 decltype 结合再借助「尾置返回类型」还可推导函数的返回类型。⽰例：

// 利⽤ auto 关键字将**返回类型后置**

template<typename T, typename U>

auto add1(T x, U y) -> decltype(x + y) {

return x + y;

}

从 **C++14** 开始⽀持仅⽤ auto 并实现返回类型推导，见下⽂ C++14 章节。

* **右值引用**

右值引用是C++11中一个非常重要的特性。

右值引用的提出，是为了在语法层面，解决向C++中添加**移动语义**时难以解决的问题，而必须扩展语法，而不是扩展标准库。

一、什么是右值引用

1. 什么是左值右值？

左值和右值是从C语言继承而来的名词，这个概念在C++中一直存在，只是大多时候被忽视了，原本是为了帮助记忆：**左值可以位于赋值语句的左侧，而右值则不能**。

然而在C++中，二者的区别就没有那么简单了，但是也可以总结出一个简单的特征：  
**左值具有持久的状态，而右值要么是字面常量，要么是在表达式求值过程中创建的临时对象**。  
**也有一种快捷的分辨方式，左值可以取地址，右值不可以取地址。**

int i = 42; // i是左值，字面常量42是右值

Object o;

Object p = 2 \* o; // p是左值，

// 而2\*o的计算结果会创建一个临时对象，是右值，再拷贝给p

2. 什么是右值引用？

对于在C++11之前的引用类型，可以称之为**左值引用**，即只能绑定至左值的引用。  
那么右值引用就是**绑定至右值**的引用，通过类型&&表示。虽然和左值引用类型符号相似，但却是**完全不同**的两个类型。

**T&可以绑定至左值，const T&和&&可以绑定至右值**

int &lRef = 0; // 错误。 字面常量0是右值，左值引用lRef不能直接绑定至右值

int obj = 0;

int &lRef = obj; // 正确。

// 可以先使用字面常量0构造一个obj对象，对象可以作为左值。

// 然后将左值引用lRef绑定至对象obj

const int &lRef = 0; // 正确。 可以将const左值引用绑定至右值

int &&lRef = 0; // 正确。 可以将右值引用绑定至右值

P.S. 需要注意的是，虽然右值引用是绑定至右值，但是其本身是个左值。可以理解为，右值引用不是字面常量也不是临时变量，所以其为左值。

二、右值引用有什么用?

**提升性能**。

前面说到右值引用的提出，是为了在语法层面，解决向C++中添加移动语义时难以解决的问题，那么为什么要添加移动语义，还要从深浅拷贝说起。  
对于一个C++11之前的自定义类，拷贝构造函数中只能指定为深拷贝或者浅拷贝。

举个栗子：

#include <iostream>

#include <memory.h>

#define BUFF\_LEN (128)

class Demo

{

public:

Demo()

{

printf("construct: %p\n", this);

m\_p = (char \*)malloc(BUFF\_LEN);

}

Demo(const Demo &other)

{

printf("copy from %p to %p\n", &other, this);

// 深拷贝

m\_p = (char \*)malloc(BUFF\_LEN);

memcpy(m\_p, other.m\_p, BUFF\_LEN);

}

~Demo()

{

if (m\_p) {

printf("destruct: %p\n", this);

free(m\_p);

m\_p = nullptr;

}

}

public:

char \*m\_p;

};

Demo func()

{

Demo a;

return a;

}

int main()

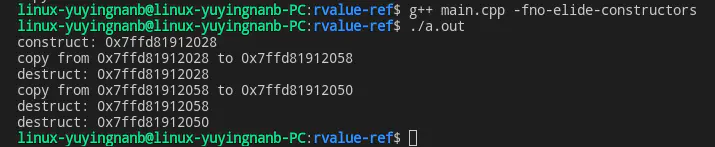
{

Demo b = func();

return 0;

}

自定义一个Demo类，类内有个指针成员变量m\_p，在构造的时候申请内存，拷贝构造的时候执行深拷贝。  
因此在main中，将func的返回值对象赋值给b的时候，会调用拷贝构造函数执行拷贝。  
通过g++ main.cpp -fno-elide-constructors 编译程序并执行，可以得到以下结果。（其中的-fno-elide-constructors编译选项为指定编译器不对构造相关过程进行优化）



拷贝构造性能缺陷

可以看到，执行了一次构造和两次拷贝构造，即一共发生了三次内存申请。但是其中有两次是函数返回时构造的临时对象，也执行了拷贝构造函数申请内存，这两次对于性能而言是浪费的，这就是在部分场景下拷贝构造的性能缺陷。

可见这类性能缺陷的根本原因是临时对象的拷贝构造，而临时对象生成的根因是C++的值语义，如果修改会导致一系列的连锁反应并很难向后兼容。所以，只能向临时对象的拷贝构造中寻找解决方案。

**如果通过临时对象构造新对象时，可以直接把临时对象已申请的内存浅拷贝，然后临时对象析构时不释放该段内存，那么新对象不就无需重新申请内存，解决这类性能缺陷了吗。**

这就是移动语义的含义，把一个对象（包括临时对象）内部的资源移动给另一个对象。

结合右值引用，给Demo类添加一个构造函数。

#include <iostream>

#include <memory.h>

#define BUFF\_LEN (128)

class Demo

{

public:

Demo()

{

printf("construct: %p\n", this);

m\_p = (char \*)malloc(BUFF\_LEN);

}

// 拷贝构造

Demo(const Demo &other)

{

printf("copy from %p to %p\n", &other, this);

// 深拷贝

m\_p = (char \*)malloc(BUFF\_LEN);

memcpy(m\_p, other.m\_p, BUFF\_LEN);

}

// 移动构造

Demo(Demo &&other)

{

printf("move from %p to %p\n", &other, this);

// 浅拷贝

m\_p = other.m\_p;

other.m\_p = nullptr;

}

~Demo()

{

if (m\_p) {

printf("destruct: %p\n", this);

free(m\_p);

m\_p = nullptr;

}

}

public:

char \*m\_p;

};

Demo func()

{

Demo a;

return a;

}

int main()

{

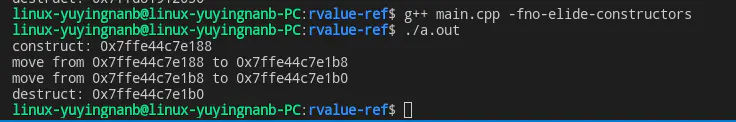
Demo b = func();

return 0;

}

因为右值引用与左值引用是完全不同的类型，所以可以重载一个构造函数Demo::Demo(Demo &&other)，也可以称为移动构造函数。

此时再重新编译并执行，打印结果如下。



使用移动构造函数

可见，依旧是调用了三次构造函数，但是其中两次调用的是我们新加的移动构造，其内部并没有进行内存申请，而且被拷贝的临时对象在析构时也没有进行已申请内存的释放。由此上述的性能缺陷被解决了。

小结：

1. 通过右值引用类型重载构造函数，对于由其他对象构造的新对象，可以根据拷贝源的属性不同（左值/右值），分别调用原本的拷贝构造，或者移动拷贝构造，执行不同的深/浅拷贝。
2. C++11只是通过右值引用类型，提供了移动拷贝实现的机制，但是不会生成默认移动拷贝构造函数，需要开发者实现。
3. 右值匹配的入参类型优先是T&&，然后才是const T&。

三、右值引用还能怎么用？

1. 移动语义

上述使用移动构造的前提是，拷贝源为临时对象，即为右值时。  
那么广义上来思考，如果在程序执行中，某个对象在某个时刻之后再也不会被使用了，那么是不是也可以从这个时刻起，将其视为临时对象呢？

于是标准库提供了移动函数std::move，其内部会把传入的左值强转成右值再返回出来。于是原本的一个左值，可以由此传入移动构造函数。

举个栗子，依旧使用上面定义的Demo类，只修改一下main函数：

int main()

{

Demo a;

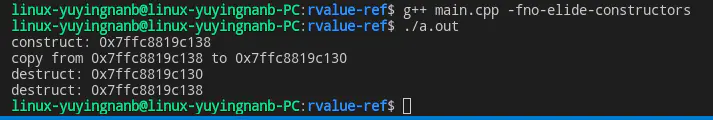
// 从此时刻起，a不再被使用

Demo b = a;

return 0;

}

打印结果：



未使用std::move

通过std::move将a对象内部的m\_p通过移动构造，直接浅拷贝移动给b：

int main()

{

Demo a;

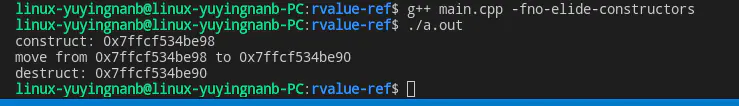
// 从此时刻起，a不再被使用

Demo b = std::move(a);

return 0;

}

打印结果：



使用std::move

注意事项：

C++11标准中并没有规定被std::move()后的原对象内部资源应当变成什么样，而交给类的开发者自行决定。所以！！被移动的对象应当确认在之后的程序中不会以任何形式被引用，或者会有未定义的行为发生。

**完美转发。**它指的是函数模板可以将自己的参数“完美”地转发给内部调用的其它函数。所谓完美，即不仅能准确地转发参数的值，还能保证被转发参数的左、右值属性不变。

* **lambda 表达式**

lambda 表达式是 [C++](http://c.biancheng.net/cplus/)11 最重要也最常用的一个特性之一，[C#](http://c.biancheng.net/csharp/) 3.5 和 [Java](http://c.biancheng.net/java/) 8 中就引入了 lambda 表达式。  
  
lambda 来源于函数式编程的概念，也是现代编程语言的一个特点。C++11 这次终于把 lambda 加进来了。  
  
lambda表达式有如下优点：

* 声明式编程风格：就地匿名定义目标函数或函数对象，不需要额外写一个命名函数或者函数对象。以更直接的方式去写程序，好的可读性和可维护性。
* 简洁：不需要额外再写一个函数或者函数对象，避免了代码膨胀和功能分散，让开发者更加集中精力在手边的问题，同时也获取了更高的生产率。
* 在需要的时间和地点实现功能闭包，使程序更灵活。

下面，先从 lambda 表达式的基本功能开始介绍它。

## lambda 表达式的概念和基本用法

lambda 表达式定义了一个匿名函数，并且可以捕获一定范围内的变量。lambda 表达式的语法形式可简单归纳如下：

[ capture ] ( params ) opt -> ret { body; };

其中 capture 是捕获列表，params 是参数表，opt 是函数选项，ret 是返回值类型，body是函数体。  
  
因此，一个完整的 lambda 表达式看起来像这样：

1. **auto** f = [](int a) -> int { **return** a + 1; };
2. std::cout << f(1) << std::endl; // 输出: 2

可以看到，上面通过一行代码定义了一个小小的功能闭包，用来将输入加 1 并返回。  
  
在 C++11 中，lambda 表达式的返回值是通过前面介绍的《[C++返回值类型后置](http://c.biancheng.net/view/3727.html)》语法来定义的。其实很多时候，lambda 表达式的返回值是非常明显的，比如这个例子。因此，C++11 中允许省略 lambda 表达式的返回值定义：

auto f = [](int a){ return a + 1; };

这样编译器就会根据 return 语句自动推导出返回值类型。  
  
需要注意的是，初始化列表不能用于返回值的自动推导：

auto x1 = [](int i){ return i; };  // OK: return type is int  
auto x2 = [](){ return { 1, 2 }; };  // error: 无法推导出返回值类型

这时我们需要显式给出具体的返回值类型。  
  
另外，lambda 表达式在没有参数列表时，参数列表是可以省略的。因此像下面的写法都是正确的：

auto f1 = [](){ return 1; };  
auto f2 = []{ return 1; };  // 省略空参数表

## 使用 lambda 表达式捕获列表

lambda 表达式还可以通过捕获列表捕获一定范围内的变量：

* [] 不捕获任何变量。
* [&] 捕获外部作用域中所有变量，并作为引用在函数体中使用（按引用捕获）。
* [=] 捕获外部作用域中所有变量，并作为副本在函数体中使用（按值捕获）。
* [=，&foo] 按值捕获外部作用域中所有变量，并按引用捕获 foo 变量。
* [bar] 按值捕获 bar 变量，同时不捕获其他变量。
* [this] 捕获当前类中的 this [指针](http://c.biancheng.net/c/80/)，让 lambda 表达式拥有和当前类成员函数同样的访问权限。如果已经使用了 & 或者 =，就默认添加此选项。捕获 this 的目的是可以在 lamda 中使用当前类的成员函数和成员变量。

下面看一下它的具体用法，如下所示。  
  
【实例】lambda 表达式的基本用法。

1. **class** A
2. {
3. **public**:
4. int i\_ = 0;
5. void func(int x, int y)
6. {
7. **auto** x1 = []{ **return** i\_; }; // error，没有捕获外部变量
8. **auto** x2 = [=]{ **return** i\_ + x + y; }; // OK，捕获所有外部变量
9. **auto** x3 = [&]{ **return** i\_ + x + y; }; // OK，捕获所有外部变量
10. **auto** x4 = [**this**]{ **return** i\_; }; // OK，捕获this指针
11. **auto** x5 = [**this**]{ **return** i\_ + x + y; }; // error，没有捕获x、y
12. **auto** x6 = [**this**, x, y]{ **return** i\_ + x + y; }; // OK，捕获this指针、x、y
13. **auto** x7 = [**this**]{ **return** i\_++; }; // OK，捕获this指针，并修改成员的值
14. }
15. };
16. int a = 0, b = 1;
17. **auto** f1 = []{ **return** a; }; // error，没有捕获外部变量
18. **auto** f2 = [&]{ **return** a++; }; // OK，捕获所有外部变量，并对a执行自加运算
19. **auto** f3 = [=]{ **return** a; }; // OK，捕获所有外部变量，并返回a
20. **auto** f4 = [=]{ **return** a++; }; // error，a是以复制方式捕获的，无法修改
21. **auto** f5 = [a]{ **return** a + b; }; // error，没有捕获变量b
22. **auto** f6 = [a, &b]{ **return** a + (b++); }; // OK，捕获a和b的引用，并对b做自加运算
23. **auto** f7 = [=, &b]{ **return** a + (b++); }; // OK，捕获所有外部变量和b的引用，并对b做自加运算

从上例中可以看到，lambda 表达式的捕获列表精细地控制了 lambda 表达式能够访问的外部变量，以及如何访问这些变量。  
  
需要注意的是，默认状态下 lambda 表达式无法修改通过复制方式捕获的外部变量。如果希望修改这些变量的话，我们需要使用引用方式进行捕获。  
  
一个容易出错的细节是关于 lambda 表达式的延迟调用的：

1. int a = 0;
2. **auto** f = [=]{ **return** a; }; // 按值捕获外部变量
3. a += 1; // a被修改了
4. std::cout << f() << std::endl; // 输出？

在这个例子中，lambda 表达式按值捕获了所有外部变量。在捕获的一瞬间，a 的值就已经被复制到f中了。之后 a 被修改，但此时 f 中存储的 a 仍然还是捕获时的值，因此，最终输出结果是 0。  
  
如果希望 lambda 表达式在调用时能够即时访问外部变量，我们应当使用引用方式捕获。  
  
从上面的例子中我们知道，按值捕获得到的外部变量值是在 lambda 表达式定义时的值。此时所有外部变量均被复制了一份存储在 lambda 表达式变量中。此时虽然修改 lambda 表达式中的这些外部变量并不会真正影响到外部，我们却仍然无法修改它们。  
  
那么如果希望去修改按值捕获的外部变量应当怎么办呢？这时，需要显式指明 lambda 表达式为 mutable：

1. int a = 0;
2. **auto** f1 = [=]{ **return** a++; }; // error，修改按值捕获的外部变量
3. **auto** f2 = [=]() **mutable** { **return** a++; }; // OK，mutable

需要注意的一点是，被 mutable 修饰的 lambda 表达式就算没有参数也要写明参数列表。

## lambda 表达式的类型

最后，介绍一下 lambda 表达式的类型。  
  
lambda 表达式的类型在 C++11 中被称为“闭包类型（Closure Type）”。它是一个特殊的，匿名的非 nunion 的类类型。  
  
因此，我们可以认为它是一个带有 operator() 的类，即仿函数。因此，我们可以使用 std::function 和 std::bind 来存储和操作 lambda 表达式：

1. std::function<int(int)> f1 = [](int a){ **return** a; };
2. std::function<int(void)> f2 = std::bind([](int a){ **return** a; }, 123);

另外，对于没有捕获任何变量的 lambda 表达式，还可以被转换成一个普通的函数指针：

1. **u**[**sin**](http://c.biancheng.net/ref/sin.html)**g** func\_t = int(\*)(int);
2. func\_t f = [](int a){ **return** a; };
3. f(123);

lambda 表达式可以说是就地定义仿函数闭包的“语法糖”。它的捕获列表捕获住的任何外部变量，最终均会变为闭包类型的成员变量。而一个使用了成员变量的类的 operator()，如果能直接被转换为普通的函数指针，那么 lambda 表达式本身的 this 指针就丢失掉了。而没有捕获任何外部变量的 lambda 表达式则不存在这个问题。  
  
这里也可以很自然地解释为何按值捕获无法修改捕获的外部变量。因为按照 C++ 标准，lambda 表达式的 operator() 默认是 const 的。一个 const 成员函数是无法修改成员变量的值的。而 mutable 的作用，就在于取消 operator() 的 const。  
  
需要注意的是，没有捕获变量的 lambda 表达式可以直接转换为函数指针，而捕获变量的 lambda 表达式则不能转换为函数指针。看看下面的代码：

1. **typedef** void(\*Ptr)(int\*);
2. Ptr p = [](int\* p){**delete** p;}; // 正确，没有状态的lambda（没有捕获）的lambda表达式可以直接转换为函数指针
3. Ptr p1 = [&](int\* p){**delete** p;}; // 错误，有状态的lambda不能直接转换为函数指针

上面第二行代码能编译通过，而第三行代码不能编译通过，因为第三行的代码捕获了变量，不能直接转换为函数指针。

## 声明式的编程风格，简洁的代码

就地定义匿名函数，不再需要定义函数对象，大大简化了标准库算法的调用。比如，在 C++11 之前，我们要调用 for\_each 函数将 vector 中的偶数打印出来，如下所示。  
  
【实例】lambda 表达式代替函数对象的示例。

1. **class** CountEven
2. {
3. int& count\_;
4. **public**:
5. CountEven(int& count) : count\_(count) {}
6. void **operator**()(int val)
7. {
8. **if** (!(val & 1)) // val % 2 == 0
9. {
10. ++ count\_;
11. }
12. }
13. };
14. std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
15. int even\_count = 0;
16. for\_each(v.begin(), v.end(), CountEven(even\_count));
17. std::cout << "The number of even is " << even\_count << std::endl;

这样写既烦琐又容易出错。有了 lambda 表达式以后，我们可以使用真正的闭包概念来替换掉这里的仿函数，代码如下：

1. std::vector<int> v = { 1, 2, 3, 4, 5, 6 };
2. int even\_count = 0;
3. for\_each( v.begin(), v.end(), [&even\_count](int val)
4. {
5. **if** (!(val & 1)) // val % 2 == 0
6. {
7. ++ even\_count;
8. }
9. });
10. std::cout << "The number of even is " << even\_count << std::endl;

lambda 表达式的价值在于，就地封装短小的功能闭包，可以极其方便地表达出我们希望执行的具体操作，并让上下文结合得更加紧密。

* **std::function and std::bind**

1. 可调用对象

可调用对象有一下几种定义：

* 是一个函数指针，参考 [C++ 函数指针和函数类型](https://www.jianshu.com/p/6ecfd541ec04)；
* 是一个具有operator()成员函数的类的对象；
* 可被转换成函数指针的类对象；
* 一个类成员函数指针；

C++中**可调用对象**的虽然都有一个比较统一的操作形式，但是定义方法五花八门，这样就导致使用统一的方式保存可调用对象或者传递可调用对象时，会十分繁琐。C++11中提供了std::function和std::bind统一了可调用对象的各种操作。

不同类型可能具有相同的调用形式，如：

// 普通函数

int add(int a, int b){return a+b;}

// lambda表达式

auto mod = [](int a, int b){ return a % b;}

// 函数对象类

struct divide{

int operator()(int denominator, int divisor){

return denominator/divisor;

}

};

上述三种可调用对象虽然类型不同，但是共享了一种调用形式：

int(int ,int)

std::function就可以将上述类型保存起来，如下：

std::function<int(int ,int)> a = add;

std::function<int(int ,int)> b = mod ;

std::function<int(int ,int)> c = divide();

2. std::function

* std::function 是一个可调用对象包装器，是一个类模板，可以容纳除了类成员函数指针之外的所有可调用对象，它可以用统一的方式处理函数、函数对象、函数指针，并允许保存和延迟它们的执行。
* 定义格式：std::function<函数类型>。
* std::function可以取代函数指针的作用，因为它可以延迟函数的执行，特别适合作为回调函数使用。它比普通函数指针更加的灵活和便利。

3. std::bind

可将std::bind函数看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来“适应”原对象的参数列表。

std::bind将可调用对象与其参数一起进行绑定，绑定后的结果可以使用std::function保存。std::bind主要有以下两个作用：

* 将可调用对象和其参数绑定成一个防函数；
* 只绑定部分参数，减少可调用对象传入的参数。

3.1 std::bind绑定普通函数

double my\_divide (double x, double y) {return x/y;}

auto fn\_half = std::bind (my\_divide,\_1,2);

std::cout << fn\_half(10) << '\n'; // 5

* bind的第一个参数是函数名，普通函数做实参时，会隐式转换成函数指针。因此std::bind (my\_divide,\_1,2)等价于std::bind (&my\_divide,\_1,2)；
* \_1表示占位符，位于<functional>中，std::placeholders::\_1；

## std::bind函数

调用bind函数的方式为：

cpp

auto newCallable = bind(callable, arg\_list);

说明：  
callable : 本身就是一个可调用的函数  
arg\_list : 用逗号分割的参数列表，对应给定的callable参数

当我们调用newCallable时，会调用callable函数，并且将arg\_list的参数传递给callable函数。

### 参数占位符

arg\_list中有许多占位符，形如\_n的名字,其中n是一个整数，表示newCallable的参数的位置比如：\_1为newCallable的第一个参数，\_2为第二个参数...。

举例说明：  
现在有一个函数：

bool checkSize(const string &s, string::size\_type sz)

{

return s.size() >= sz;

}

使用bind生成一个调用checkSize的对象：

auto check = bind(checkSize, \_1, 6);

其中的\_1表示需要给第一个参数传递一个const string&。

string s = "hello";

bool b1 = check(s); //check(s) 会调用checkSize(s, 6)

### \_n参数说明

\\_n参数都定义在一个名字为placeholders命名空间中，这个命名空间又在std命名空间中。那么\\_1就是：

using std::placeholders::\_1;

对每一个占位符都要使用using声明，非常麻烦，可以使用:

using namespace std::placeholders;

placeholders的明明空间也定义在functional.h头文件中

### bind的参数顺序

bind绑定的给定函数的参数的顺序。比如func是一个函数，它有5个参数，下面对bind的调用：

hljs

auto g = bind(fun, a, b, \_2, c, \_1);

生成一个新的可调用对象，有两个占位符，传递给g的第一个参数绑定到\_1,第二个参数绑定到\_2,bind调用会将g(\_1, \_2)映射为f(a, b, \_2, c, \_1),如：调用g(X, Y)就会调用f(a, b, Y, c, X);

### 绑定引用参数

默认情况下，bind的非占位符的参数是拷贝到bind返回的可调用的函数中，与Lambda类似，有时候绑定的参数希望用引用的方式传递，或不让绑定的参数类型拷贝。  
例如：

for\_each(words.begin(), words.end(), [&os, c](const string &s) {os << s << c;});

可以编写一个函数完成同样的功能：

hljs

ostream &print(ostream &os, const strint &s, char c)

{

return os << s << c;

}

但是：不能直接用bind来代替os的捕获：

for\_each(words.begin(), words.end(), bind(print, os, \_1, ' ');

由于不能拷贝一个ostream。如果我们希望bind一个函数而不希望拷贝参数，就需要用标准库的ref函数。

for\_each(words.begin(), words.end(), bind(printf, ref(os), \_1, ' '));

函数ref返回一个对象，包含给定的引用，此对象是可以拷贝的。标准库中还有cref函数，生成一个保存const引用的类。ref和cref都在头文件"functional.h"中

3.2 std::bind绑定一个成员函数

struct Foo {

void print\_sum(int n1, int n2)

{

std::cout << n1+n2 << '\n';

}

int data = 10;

};

int main()

{

Foo foo;

auto f = std::bind(&Foo::print\_sum, &foo, 95, std::placeholders::\_1);

f(5); // 100

}

* bind绑定类成员函数时，第一个参数表示对象的成员函数的指针，第二个参数表示对象的地址。
* 必须显示的指定&Foo::print\_sum，因为编译器不会将对象的成员函数隐式转换成函数指针，所以必须在Foo::print\_sum前添加&；
* 使用对象成员函数的指针时，必须要知道该指针属于哪个对象，因此第二个参数为对象的地址 &foo；

3.3 绑定一个引用参数

默认情况下，bind的那些不是占位符的参数被拷贝到bind返回的可调用对象中。但是，与lambda类似，有时对有些绑定的参数希望以引用的方式传递，或是要绑定参数的类型无法拷贝。

#include <iostream>

#include <functional>

#include <vector>

#include <algorithm>

#include <sstream>

using namespace std::placeholders;

using namespace std;

ostream & print(ostream &os, const string& s, char c)

{

os << s << c;

return os;

}

int main()

{

vector<string> words{"helo", "world", "this", "is", "C++11"};

ostringstream os;

char c = ' ';

for\_each(words.begin(), words.end(),

[&os, c](const string & s){os << s << c;} );

cout << os.str() << endl;

ostringstream os1;

// ostream不能拷贝，若希望传递给bind一个对象，

// 而不拷贝它，就必须使用标准库提供的ref函数

for\_each(words.begin(), words.end(),

bind(print, ref(os1), \_1, c));

cout << os1.str() << endl;

}

* **模板的别名**

C++11引入了using，可以轻松的定义别名，而不是使用繁琐的typedef。

**typedef** std**::**vector**<**std**::**vector**<int>>** vvi; *// before c++11*

**using** vvi **=** std**::**vector**<**std**::**vector**<int>>**; *// c++11*

**template<class** **T>**

**struct** **Alloc** { };

**template<class** **T>**

**using** Vec **=** vector**<**T, Alloc**<**T**>>**; *// 类型标识为 vector<T, Alloc<T>>*

Vec**<int>** v; *// Vec<int> 同 vector<int, Alloc<int>>*

使用using明显简洁并且易读，大家可能之前也见过使用typedef定义函数指针之类的操作，那烂代码我就不列出来了，反正我是看不懂也不想看...以后都可以使用using，额还是列出来吧。

**typedef** **void** (**\***func)(**int**, **int**); *// 啥玩意，看不懂*

**using** func **=** **void** (**\***)(**int**, **int**); *// 起码比typedef容易看的懂吧*

上面的代码使用using起码比typedef容易看的懂一些吧，但是我还是看不懂，因为我从来不用这种来表示函数指针，用std::function()、std::bind()、std::placeholder()、lambda表达式它不香吗。

* **基于范围的for循环**

直接看代码

vector**<int>** vec;

**for** (**auto** iter **=** vec.begin(); iter **!=** vec.end(); iter**++**) { *// before c++11*

cout **<<** **\***iter **<<** endl;

}

**for** (**int** i : vec) { *// c++11基于范围的for循环*

cout **<<** "i" **<<** endl;

}

* **委托构造函数**

委托构造函数允许在同一个类中一个构造函数调用另外一个构造函数，可以在变量初始化时简化操作，通过代码来感受下委托构造函数的妙处吧：

不使用委托构造函数：

**struct** **A** {

A(){}

A(**int** a) { a\_ **=** a; }

A(**int** a, **int** b) { *// 好麻烦*

a\_ **=** a;

b\_ **=** b;

}

A(**int** a, **int** b, **int** c) { *// 好麻烦*

a\_ **=** a;

b\_ **=** b;

c\_ **=** c;

}

**int** a\_;

**int** b\_;

**int** c\_;

};

使用委托构造函数：

**struct** **A** {

A(){}

A(**int** a) { a\_ **=** a; }

A(**int** a, **int** b) **:** A(a) { b\_ **=** b; }

A(**int** a, **int** b, **int** c) **:** A(a, b) { c\_ **=** c; }

**int** a\_;

**int** b\_;

**int** c\_;

};

* **继承构造函数**

继承构造函数可以让派生类直接使用基类的构造函数，如果有一个派生类，我们希望派生类采用和基类一样的构造方式，可以直接使用基类的构造函数，而不是再重新写一遍构造函数，老规矩，看代码：

不使用继承构造函数：

**struct** **Base** {

Base() {}

Base(**int** a) { a\_ **=** a; }

Base(**int** a, **int** b) **:** Base(a) { b\_ **=** b; }

Base(**int** a, **int** b, **int** c) **:** Base(a, b) { c\_ **=** c; }

**int** a\_;

**int** b\_;

**int** c\_;

};

**struct** **Derived** **:** Base {

Derived() {}

Derived(**int** a) **:** Base(a) {} *// 好麻烦*

Derived(**int** a, **int** b) **:** Base(a, b) {} *// 好麻烦*

Derived(**int** a, **int** b, **int** c) **:** Base(a, b, c) {} *// 好麻烦*

};

**int** **main**() {

Derived a(1, 2, 3);

**return** 0;

}

使用继承构造函数：

**struct** **Base** {

Base() {}

Base(**int** a) { a\_ **=** a; }

Base(**int** a, **int** b) **:** Base(a) { b\_ **=** b; }

Base(**int** a, **int** b, **int** c) **:** Base(a, b) { c\_ **=** c; }

**int** a\_;

**int** b\_;

**int** c\_;

};

**struct** **Derived** **:** Base {

**using** Base**::**Base;

};

**int** **main**() {

Derived a(1, 2, 3);

**return** 0;

}

只需要使用using Base::Base继承构造函数，就免去了很多重写代码的麻烦。

* **nullptr**

nullptr是c++11用来表示空指针新引入的常量值，在c++中如果表示空指针语义时建议使用nullptr而不要使用NULL，因为NULL本质上是个int型的0，其实不是个指针。举例：

**void** **func**(**void** **\***ptr) {

cout **<<** "func ptr" **<<** endl;

}

**void** **func**(**int** i) {

cout **<<** "func i" **<<** endl;

}

**int** **main**() {

func(NULL); *// 编译失败，会产生二义性*

func(**nullptr**); *// 输出func ptr*

**return** 0;

}

* **Final & override**

c++11关于继承新增了两个关键字，final用于修饰一个类，表示禁止该类进一步派生和虚函数的进一步重载，override用于修饰派生类中的成员函数，标明该函数重写了基类函数，如果一个函数声明了override但父类却没有这个虚函数，编译报错，使用override关键字可以避免开发者在重写基类函数时无意产生的错误。

示例代码1：

**struct** **Base** {

**virtual** **void** **func**() {

cout **<<** "base" **<<** endl;

}

};

**struct** **Derived** **:** **public** Base{

**void** **func**() **override** { *// 确保func被重写*

cout **<<** "derived" **<<** endl;

}

**void** **fu**() **override** { *// error，基类没有fu()，不可以被重写*

}

};

示例代码2：

**struct** **Base** **final** {

**virtual** **void** **func**() {

cout **<<** "base" **<<** endl;

}

};

**struct** **Derived** **:** **public** Base{ *// 编译失败，final修饰的类不可以被继承*

**void** **func**() **override** {

cout **<<** "derived" **<<** endl;

}

};

* **Default，Delete, Explict**

c++11引入default特性，多数时候用于声明构造函数为默认构造函数，如果类中有了自定义的构造函数，编译器就不会隐式生成默认构造函数，如下代码：

**struct** **A** {

**int** a;

A(**int** i) { a **=** i; }

};

**int** **main**() {

A a; *// 编译出错*

**return** 0;

}

上面代码编译出错，因为没有匹配的构造函数，因为编译器没有生成默认构造函数，而通过default，程序员只需在函数声明后加上“=default;”，就可将该函数声明为 defaulted 函数，编译器将为显式声明的 defaulted 函数自动生成函数体，如下：

**struct** **A** {

A() **=** **default**;

**int** a;

A(**int** i) { a **=** i; }

};

**int** **main**() {

A a;

**return** 0;

}

编译通过。

c++中，如果开发人员没有定义特殊成员函数，那么编译器在需要特殊成员函数时候会隐式自动生成一个默认的特殊成员函数，例如拷贝构造函数或者拷贝赋值操作符，如下代码：

**struct** **A** {

A() **=** **default**;

**int** a;

A(**int** i) { a **=** i; }

};

**int** **main**() {

A a1;

A a2 **=** a1; *// 正确，调用编译器隐式生成的默认拷贝构造函数*

A a3;

a3 **=** a1; *// 正确，调用编译器隐式生成的默认拷贝赋值操作符*

}

而我们有时候想禁止对象的拷贝与赋值，可以使用delete修饰，如下：

**struct** **A** {

A() **=** **default**;

A(**const** A**&**) **=** **delete**;

A**&** **operator=**(**const** A**&**) **=** **delete**;

**int** a;

A(**int** i) { a **=** i; }

};

**int** **main**() {

A a1;

A a2 **=** a1; *// 错误，拷贝构造函数被禁用*

A a3;

a3 **=** a1; *// 错误，拷贝赋值操作符被禁用*

}

delele函数在c++11中很常用，std::unique\_ptr就是通过delete修饰来禁止对象的拷贝的。

explicit专用于修饰构造函数，表示只能显式构造，不可以被隐式转换，根据代码看explicit的作用：

不用explicit：

**struct** **A** {

A(**int** value) { *// 没有explicit关键字*

cout **<<** "value" **<<** endl;

}

};

**int** **main**() {

A a **=** 1; *// 可以隐式转换*

**return** 0;

}

使用explicit:

**struct** **A** {

**explicit** **A**(**int** value) {

cout **<<** "value" **<<** endl;

}

};

**int** **main**() {

A a **=** 1; *// error，不可以隐式转换*

A aa(2); *// ok*

**return** 0;

}

* **const, constexpr**
* **enum class**

c++11新增有作用域的枚举类型，看代码

不带作用域的枚举代码：

**enum** **AColor** {

kRed,

kGreen,

kBlue

};

**enum** **BColor** {

kWhite,

kBlack,

kYellow

};

**int** **main**() {

**if** (kRed **==** kWhite) {

cout **<<** "red == white" **<<** endl;

}

**return** 0;

}

如上代码，不带作用域的枚举类型可以自动转换成整形，且不同的枚举可以相互比较，代码中的红色居然可以和白色比较，这都是潜在的难以调试的bug，而这种完全可以通过有作用域的枚举来规避。

有作用域的枚举代码：

**enum** **class** **AColor** {

kRed,

kGreen,

kBlue

};

**enum** **class** **BColor** {

kWhite,

kBlack,

kYellow

};

**int** **main**() {

**if** (AColor**::**kRed **==** BColor**::**kWhite) { *// 编译失败*

cout **<<** "red == white" **<<** endl;

}

**return** 0;

}

使用带有作用域的枚举类型后，对不同的枚举进行比较会导致编译失败，消除潜在bug，同时带作用域的枚举类型可以选择底层类型，默认是int，可以改成char等别的类型。

**enum** **class** **AColor** **:** **char** {

kRed,

kGreen,

kBlue

};

我们平时编程过程中使用枚举，一定要使用有作用域的枚举取代传统的枚举。

* **thread, mutex, conditional var**

std::thread相关

c++11之前你可能使用pthread\_xxx来创建线程，繁琐且不易读，c++11引入了std::thread来创建线程，支持对线程join或者detach。直接看代码：

#include <iostream>

#include <thread>

using namespace std;

int main() {

auto func = []() {

for (int i = 0; i < 10; ++i) {

cout << i << " ";

}

cout << endl;

};

std::thread t(func);

if (t.joinable()) {

t.detach();

}

auto func1 = [](int k) {

for (int i = 0; i < k; ++i) {

cout << i << " ";

}

cout << endl;

};

std::thread tt(func1, 20);

if (tt.joinable()) { // 检查线程可否被join

tt.join();

}

return 0;

}

上述代码中，函数func和func1运行在线程对象t和tt中，从刚创建对象开始就会新建一个线程用于执行函数，调用join函数将会阻塞主线程，直到线程函数执行结束，线程函数的返回值将会被忽略。如果不希望线程被阻塞执行，可以调用线程对象的detach函数，表示将线程和线程对象分离。

如果没有调用join或者detach函数，假如线程函数执行时间较长，此时线程对象的生命周期结束调用析构函数清理资源，这时可能会发生错误，这里有两种解决办法，一个是调用join()，保证线程函数的生命周期和线程对象的生命周期相同，另一个是调用detach()，将线程和线程对象分离，这里需要注意，如果线程已经和对象分离，那我们就再也无法控制线程什么时候结束了，不能再通过join来等待线程执行完。

std::mutex相关

std::mutex是一种线程同步的手段，用于保存多线程同时操作的共享数据。

mutex分为四种：

* std::mutex：独占的互斥量，不能递归使用，不带超时功能
* std::recursive\_mutex：递归互斥量，可重入，不带超时功能
* std::timed\_mutex：带超时的互斥量，不能递归
* std::recursive\_timed\_mutex：带超时的互斥量，可以递归使用

拿一个std::mutex和std::timed\_mutex举例吧，别的都是类似的使用方式：

std::mutex:

#include <iostream>

#include <mutex>

#include <thread>

using namespace std;

std::mutex mutex\_;

int main() {

auto func1 = [](int k) {

mutex\_.lock();

for (int i = 0; i < k; ++i) {

cout << i << " ";

}

cout << endl;

mutex\_.unlock();

};

std::thread threads[5];

for (int i = 0; i < 5; ++i) {

threads[i] = std::thread(func1, 200);

}

for (auto& th : threads) {

th.join();

}

return 0;

}

std lock

这里主要介绍两种RAII方式的锁封装，可以动态的释放锁资源，防止线程由于编码失误导致一直持有锁。

c++11主要有std::lock\_guard和std::unique\_lock两种方式，使用方式都类似，如下：

#include <iostream>

#include <mutex>

#include <thread>

#include <chrono>

using namespace std;

std::mutex mutex\_;

int main() {

auto func1 = [](int k) {

// std::lock\_guard<std::mutex> lock(mutex\_);

std::unique\_lock<std::mutex> lock(mutex\_);

for (int i = 0; i < k; ++i) {

cout << i << " ";

}

cout << endl;

};

std::thread threads[5];

for (int i = 0; i < 5; ++i) {

threads[i] = std::thread(func1, 200);

}

for (auto& th : threads) {

th.join();

}

return 0;

}

std::lock\_gurad相比于std::unique\_lock更加轻量级，少了一些成员函数，std::unique\_lock类有unlock函数，可以手动释放锁，所以条件变量都配合std::unique\_lock使用，而不是std::lock\_guard，因为条件变量在wait时需要有手动释放锁的能力，具体关于条件变量后面会讲到。

**理解条件变量**

条件变量可以用来管理thread间的通信。一个线程可以等待在一个条件变量上，直到发生某个事件。

考虑一个场景，一个线程访问一个队列时，发现队列为空，他只能等待 直到其他线程将一个节点添加到队列中。这种情况就需要用到条件变量。（当然，用低效的轮询也可以，不停的去判断队列中是否有节点）

**C++11 标准库提供的 condition\_variable 概览**

condition\_variable cv{};  *//默认构造函数*

cv.~condition\_variable();  *//析构函数*

*----------*

cv.notify\_one();  *//随机通知一个等待该条件变量的线程*

cv.notify\_all();  *//通知全部等待该条件变量的线程*

*----------*

cv.wait(lck);  *//等待 直到 被唤醒 （存在为唤醒 ）*

cv.wait(lck,pred);  *// 相当于 while(!pared())wait(lck)*

x=cv.wait\_util(lck,tp);  *//等待 直到 被唤醒或者到达某个时间点*

*//返回值 x 如果超时 则 x==timeout,否者x==cv\_status::no\_timeou*

b=cv.wait\_until(lck,tp,pred) *// 等价于： while(!pared())*

*// { if(wait\_until(lck,tp)==cv\_sattus::timeout);}b=pred()*

x=cv.wait\_for(lck,d);  *//等待 直到被唤醒或者超时 @d 超时时间*

b=cv.wait\_for(lck,d,pred);  *//等价于 b=cv.wait\_until(lck,steady\_clock::now()+d,std::move(pre));*

**剖析等待操作**

cv.wait(lck) 需要与一个互斥锁搭配使用  
代码通常如下

//伪代码

...

condition cond; //条件变量

mutex mu; //互斥锁

/\*

进入临界区

\*/

mu.lock();

while(v.is\_empty())

{

cond.wait(&mu);

}

...do something ..

mu.unlock();

/\*

退出临界区

\*/

**在调用wait时：**

**首先该线程需要获取到保护v的锁 进入临界取，因为 v 应该是多个线程可以访问的。**

**在wait函数执行如下操作：**

**1. wait函数内部首先 mu.unlock() 释放锁 ；**

**2. 然后进入等待；**

**3. 如果被唤醒，则调用mu.lock() 再次获取锁。**

**为什么wait函数要在正真开始等待之前释放锁？**

**因为 :**

**1. 变量 v 需要其他线程 需要获取锁 然后给v添加元素，以便可以更改v.is\_empty()这个条件；**

**2. 让其他线程也有机会进入临界区 等待相同的条件**

**生产者-消费者 示例代码**

*//sync\_queue*

#include <list>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <system\_error>

#include <iostream>

template<typename T>

class Sync\_queue

{

public:

Sync\_queue();

~Sync\_queue();

void put(const T& val);

void put(T&& val);

void get(T& val);

private:

std::condition\_variable m\_cond;

std::mutex m\_mutex;

std::list<T> m\_q;

};

template<typename T>

Sync\_queue<T>::~Sync\_queue()

{

}

template<typename T>

void Sync\_queue<T>::get(T& val)

{

std::unique\_lock<std::mutex> lck(m\_mutex);

while (m\_q.empty()) *//使用while 而不用 if 是为了防止虚假唤醒*

{

if (m\_cond.wait\_for(lck, std::chrono::milliseconds{ 2000 }) == std::cv\_status::timeout)

{

break;

}

}

if (m\_q.empty())

{

throw "超时";

}

else

{

val = m\_q.front();

m\_q.pop\_front();

std::cout << "取出一个元素\n";

}

}

template<typename T>

Sync\_queue<T>::Sync\_queue()

{

}

template<typename T>

void Sync\_queue<T>::put(T&& val)

{

std::lock\_guard<std::mutex> lck(m\_mutex);

m\_q.push\_back(std::forward<T>(val));

m\_cond.notify\_one();

}

template<typename T>

void Sync\_queue<T>::put(const T& val)

{

{

std::lock\_guard<std::mutex> lck(m\_mutex);

m\_q.push\_back(val);

}

m\_cond.notify\_one(); *//通知时可以不需要mutex的保护*

}

*//main.cpp*

#include "Sync\_queue.h"

Sync\_queue<int> g\_queue;

void produce()

{

static int i = 0;

while (true)

{

g\_queue.put(++i);

std::this\_thread::sleep\_for(std::chrono::milliseconds(1000));

}

}

void consumer()

{

while (true)

{

try

{

int val;

g\_queue.get(val);

std::cout << "获取 ：" << val << std::endl;

}

catch (const char \* e)

{

std::cout << e << std::endl;

}

}

}

int main()

{

std::thread t\_c{ consumer };

std::thread t\_p{ produce };

t\_c.join();

t\_p.join();

return 0;

}

*// 在多个消费者的情况 put 函数还可以进行一些改进*

*// if(m\_q.size()>n)*

*// {m\_cond.notify\_all();}*

*// else*

*// {m\_cond.notify\_one();}*

**std::future, std::async, std::promise, std::packaged\_task**

* **新增数据结构**
* std::forward\_list：单向链表，只可以前进，在特定场景下使用，相比于std::list节省了内存，提高了性能

std**::**forward\_list**<int>** fl **=** {1, 2, 3, 4, 5};

**for** (**const** **auto** **&**elem : fl) {

cout **<<** elem;

}

* std::unordered\_set：基于hash表实现的set，内部不会排序，使用方法和set类似
* std::unordered\_map：基于hash表实现的map，内部不会排序，使用方法和set类似
* std::array：数组，在越界访问时抛出异常，建议使用std::array替代普通的数组
* std::tuple：元组类型，类似pair，但比pair扩展性好
* **Other common c++**

As the name suggests, Preprocessors are programs that process our source code before compilation. There are a number of steps involved between writing a program and executing a program in C / C++. Let us have a look at these steps before we actually start learning about Preprocessors.

图示

描述已自动生成

You can see the intermediate steps in the above diagram. The source code written by programmers is first stored in a file, let the name be “**program.c**“. This file is then processed by preprocessors and an expanded source code file is generated named “program.i”. This expanded file is compiled by the compiler and an object code file is generated named “program.obj”. Finally, the linker links this object code file to the object code of the library functions to generate the executable file “program.exe”.

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a ‘#’ (hash) symbol. The ‘#’ symbol indicates that whatever statement starts with a ‘#’ will go to the preprocessor program to get executed. Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef* etc. Remember that the **#** symbol only provides a path to the preprocessor, and a command such as include is processed by the preprocessor program. For example, #include will include extra code in your program. We can place these preprocessor directives anywhere in our program.

**There are 4 Main Types of Preprocessor Directives:**

1. Macros
2. File Inclusion
3. Conditional Compilation
4. Other directives

**- overload override**

1. Overload（重载）

　　重载的概念最好理解，在同一个类声明范围中，定义了多个名称完全相同、参数（类型或者个数）不相同的函数，就称之为Overload（重载）。重载的特征如下：

（1）相同的范围（在同一个类中）；

（2）函数名字相同；

（3）参数不同；

（4）virtual 关键字可有可无。

2. Override（覆盖）

　　覆盖的概念其实是用来实现C++多态性的，即子类重新改写父类声明为virtual的函数。Override（覆盖）的特征如下：

（1）不同的范围（分别位于派生类与基类）；

（2）函数名字相同；

（3）参数列表完全相同；

（4）基类函数必须有virtual 关键字。

3. Overwrite（改写）

　　改写是指派生类的函数屏蔽（或者称之为“隐藏”）了与其同名的基类函数。正是这个C++的隐藏规则使得问题的复杂性陡然增加，这里面分为两种情况讨论：

（1）如果派生类的函数与基类的函数同名，但是参数不同。那么此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。

（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。那么此时，基类的函数被隐藏（注意别与覆盖混淆）。

**Exception handling**

void processAdoptions(istream& dataSource) {

while (dataSource) { // 还 有 数 据 时 ,继 续循环 ALA

\*pa = readALA(dataSource); //得 到下一个 动 物

pa->processAdoption(); //处 理 收 容 动 物

delete pa; //删 除 readALA返 回 的对象 }

}

这 个 函 数 循环遍历 dataSource内 的 信 息 ，处 理 它 所 遇 到 的 每 个 项 目 ，唯 一 要 记 住 的 一 点 是 在 每 次 循 环 结 尾 处 删 除 ps。 这是必 须 的 因 为 每 次 调 用 readALA都 建 立 一 个 堆 对 象， 如果不 删 除 对 象 循 环 将产生 资 源 泄 漏。 现 在 考 虑 一 下 如 果 pa->processAdoption抛 出 了 一 个 异 常 将会发生 什 么, processAdoptions没 有 捕 获 异 常 所 以 异 常 将 传 递 给 processAdoptions的 调 用 者, 转 递 中 processAdoptions函 数 中 的 调 用 pa->processAdoption语 句 后 的 所 有 语 句 都 被 跳 过, 这 就 是 说 pa没 有 被 删 除, 结 果 任何时候 pa->processAdoption抛 出 一 个 异 常 都 会 导 致 processAdoptions内 存 泄 漏. 使用auto\_ptr用局部变量可以解决这个问题，

在异常处理机制终结某个函数之前，C++保证 随着栈的展开 尽管局部类对象的生命期是 因为抛出异常而被结束 但是这些局部类对象的析构函数也会被调用。

。

**函数指针：**

int (\*pFunc) (int a, int b);

int test(int a) { return a; }

int main(int argc, const char \* argv[])

{ typedef int (\*fp)(int a);

fp f = test;

cout<<f(2)<<endl; return 0; }

**Functor / 函数对象:**

1. **class** CAverage
2. {
3. **public**:
4. double **operator**()(int a1, int a2, int a3)
5. { //重载()运算符
6. **return** (double)(a1 + a2 + a3) / 3;
7. }
8. };
9. int main()
10. {
11. CAverage average; //能够求三个整数平均数的函数对象
12. cout << average(3, 2, 3); //等价于 cout << average.operator(3, 2, 3);
13. **return** 0;
14. }

函数对象与函数指针比较函数对象可以把附加对象保存在函数对象中是它最大的优点。

**构造函数不能是虚函数**

1. 从vptr角度解释

虚函数的调用是通过虚函数表来查找的，而虚函数表由类的实例化对象的vptr指针(vptr可以参考C++的虚函数表指针vptr)指向，该指针存放在对象的内部空间中，需要调用构造函数完成初始化。如果构造函数是虚函数，那么调用构造函数就需要去找vptr，但此时vptr还没有初始化！

## [析构函数](https://so.csdn.net/so/search?q=%E6%9E%90%E6%9E%84%E5%87%BD%E6%95%B0&spm=1001.2101.3001.7020)可以且常常是虚函数

      这个原理上就很好理解啦，因为此时 vtable 已经初始化了，完全可以把析构函数放在虚函数表里面来调用。

**C++类有继承时，析构函数必须为虚函数。**如果不是虚函数，则使用时可能存在**内存泄漏**的问题。

===========================================================

-基类、子类构造函数的调用，拷贝、赋值函数的调用

-函数定义与声明，全局变量和全局函数的定义与声明

Account( const char \*name, double opening\_bal ) : \_name( name ), \_balance( opening\_bal )更高效？

#include 指示符读入指定文件的内容 它有两种格式 #include <> #include "my\_file.h" 如果文件名用尖括号 < 和 > 括起来 表明这个文件是一个工程或标准头文件 查 找过程会检查预定义的目录 我们可以通过设置搜索路径环境变量或命令行选项来修改这些 目录 在不同的平台上这些方法大不相同 建议你请教同事或查阅编译器手册以获得更进 一步的信息 如果文件名用一对引号括起来 则表明该文件是用户提供的头文件 查找该 文件时将从当前文件目录开始。

#ifdef DEBUG

cout << "word read: " << word << "\n";

#endif

$ CC -DDEBUG main.C

// 相等与不相等操作

bool operator==( const IntArray& ) const;

bool operator!=( const IntArray& ) const;

// 赋值操作符

IntArray& operator=( const IntArray& );

在整型文字常量前面加一个 0 该值将被解释成一个八进制数 而在前面加一个 0x 或 0X 则会使一个整型文字常量被解释成十六进制数

逗号表达式是一系列由逗号分开的表达式 这些表达式从左向右计算 逗号表达式的结 果是最右边表达式的值。

Hashset, unordered\_set

static\_cast, dynamic\_cast

template <typename T>

T myMax(T x, T y)

{

return (x>y)? x: y;

}

template <class T> void bubbleSort(T a[], int n)

{

for (int i = 0; i < n - 1; i++)

for (int j = n - 1; i < j; j--)

if (a[j] < a[j - 1])

swap(a[j], a[j - 1]);

}

template <class T, class U> class A {

T x;

U y;

public:

A() { cout << "Constructor Called" << endl; }

};

Speed of execution: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.

**STL**

[**array:**](https://www.geeksforgeeks.org/array-class-c/)Static contiguous array (class template)

array<**int**,6> ar = {1, 2, 3, 4, 5, 6};

[**vector:**](https://www.geeksforgeeks.org/vector-in-cpp-stl/)Dynamic contiguous array (class template)

vector<**int**> g1;

[**deque:**](https://www.geeksforgeeks.org/deque-cpp-stl/)Double-ended queue (class template)

[**forward\_list:**](https://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/)Singly-linked list (class template)

Forward list in STL implements singly linked list. It differs from the [**list**](https://www.geeksforgeeks.org/list-cpp-stl/) by the fact that the forward list keeps track of the location of only the next element while the list keeps track of both the next and previous elements, thus increasing the storage space required to store each element. The drawback of a forward list is that it cannot be iterated backward and its individual elements cannot be accessed directly.

[**list:**](https://www.geeksforgeeks.org/list-cpp-stl/)Doubly-linked list (class template)

Lists are [sequence containers](https://www.geeksforgeeks.org/containers-cpp-stl/) that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick.

[**Set:**](https://www.geeksforgeeks.org/set-in-cpp-stl/)Collection of unique keys, sorted by keys   
(class template)

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it.

**Properties:**

1. **Storing order –**The set stores the elements in **sorted** order.
2. **Values Characteristics** – All the elements in a set have **unique values**.
3. **Values Nature**– The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element. Thus, the valuesare **immutable**.
4. **Search Technique** – Sets follow the **Binary search tree** implementation.
5. **Arranging order –**The values in a set are **unindexed**.

[**unordered\_set:**](https://www.geeksforgeeks.org/unorderd_set-stl-uses/)Collection of unique keys, hashed by keys. (class template)

An **unordered\_set** is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the **unordered\_set** takes constant time **O(1)** on an average which can go up to linear time **O(n)** in worst case which depends on the internally used hash function, but practically they perform very well and generally provide a constant time lookup operation.

**Difference between Set and Unordered Set**

| Set | Unordered Set |
| --- | --- |
| Set stores elements in a sorted order | Unordered Set stores elements in an unsorted order |
| Set stores/acquire unique elements only | Unordered Set stores/acquire only unique values |
| Set uses Binary Search Trees for implementation | Unordered Set uses Hash Tables for implementation |
| More than one element can be erased by giving the starting and ending iterator | We can erase that element for which the iterator position is given |
| **set<datatype> Set\_Name;** | **unordered\_set<datatype> UnorderedSet\_Name;** |

[**Map:**](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/)Collection of key-value pairs, sorted by keys, keys are unique (class template).

[**multiset:**](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)Collection of keys, sorted by keys (class template)

[**multimap:**](https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/)Collection of key-value pairs, sorted by keys   
(class template)

[**unordered\_map:**](https://www.geeksforgeeks.org/unordered_map-in-stl-and-its-applications/)Collection of key-value pairs, hashed by keys, keys are unique. (class template)

[**unordered\_multiset:**](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/)Collection of keys, hashed by keys (class template)

[**unordered\_multimap:**](https://www.geeksforgeeks.org/unordered_multimap-and-its-application/)Collection of key-value pairs, hashed by keys (class template)

**unordered\_set**  
总的来说，unordered\_set 容器具有以下几个特性：

1. 不再以键值对的形式存储数据，而是直接存储数据的值；
2. 容器内部存储的各个元素的值都互不相等，且不能被修改。
3. 不会对内部存储的数据进行排序（这和该容器底层采用哈希表结构存储数据有关，可阅读《[C++ STL无序容器底层实现原理](http://c.biancheng.net/view/7235.html" \t "_blank)》一文做详细了解）

**Garbage Collector in Java and C#**

**Shared memory, mutex, Semaphore**

**Mutex is for inter thread communication**

**Exceptional handling**

**Solarflare**

**Make:** [**https://www.gnu.org/software/make/manual/make.html**](https://www.gnu.org/software/make/manual/make.html)

**Premake:** PreMake可以自动生成多个平台的MakeFile，比起其它的MakeFile生成软件（qmake, cmake），它最大的特点应该是它的工程文件其实是使用LUA语言编写的脚本，这样一来，它的工程文件可以拥有非常强大的表达能力。

**Jenkins: CI/CD**

**Train: MS thing, integrate SCM, CI/CD etc**

**C++11 multi-thread programing, std::future, 条件变量**

#### 名词解释SFINAE

    SFINAE是英文Substitution failure is not an error的缩写，意思是匹配失败不是错误。这句话什么意思呢？当调用模板函数时编译器会根据传入参数推导最合适的模板函数，在这个推导过程中如果某一个或者某几个模板函数推导出来是编译无法通过的，只要有一个可以正确推导出来，那么那几个推导得到的可能产生编译错误的模板函数并不会引发编译错误。这段话很绕，我们接下来用代码说明一下，一看便知。

struct Test {

typedef int foo;

};

template <typename T>

void f(typename T::foo) {} // Definition #1

template <typename T>

void f(T) {} // Definition #2

int main() {

f<Test>(10); // Call #1.

f<int>(10); // Call #2. Without error (even though there is no int::foo) thanks to SFINAE.

}

    这是wiki上SFINAE的一个经典示例，注释已经解释的相当明白，由于推导模板函数过程中可以找到一个正确的版本，所以即时int::foo是一个语法错误，但是编译器也不会报错。这就是SFINAE要义。在C++11中，标准确立了这种编译的行为，而不像C++98未明确定义它的行为。通过std::enable\_if和SFINAE的共同使用，会产生很多很奇妙的实现，STL库中大量的应用了这种组合，下面我们来看看他们组合一起是如何工作的。

#### std::enable\_if

    这个模板类的实现相当的简单，看一下一个版本的实现。

template<bool B, class T = void>

struct enable\_if {};

template<class T>

struct enable\_if<true, T> { typedef T type; };

    一个普通版本的模板类定义，一个偏特化版本的模板类定义。它在第一个模板参数为false的时候并不会定义type，只有在第一模板参数为true的时候才会定义type。看一下下面的模板实例化代码

typename std::enable\_if<true, int>::type t; //正确

typename std::enable\_if<true>::type; //可以通过编译，没有实际用处，推导的模板是偏特化版本，第一模板参数是true，第二模板参数是通常版本中定义的默认类型即void

typename std::enable\_if<false>::type; //无法通过编译，type类型没有定义

typename std::enable\_if<false, int>::type t2; //同上

    我们可以看到，通过typename std::enable\_if<bool>::type这样传入一个bool值，就能推导出这个type是不是未定义的。那么这种用法有什么用途呢？结合上面的SFINAE来看代码

template <typename T>

typename std::enable\_if<std::is\_trivial<T>::value>::type SFINAE\_test(T value)

{

std::cout<<"T is trival"<<std::endl;

}

template <typename T>

typename std::enable\_if<!std::is\_trivial<T>::value>::type SFINAE\_test(T value)

{

std::cout<<"T is none trival"<<std::endl;

}

    这两个函数如果是普通函数的话，根据重载的规则是不会通过编译的。即便是模板函数，如果这两个函数都能推导出正确的结果，也会产生重载二义性问题，但是正因为std::enable\_if的运用使这两个函数的返回值在同一个函数调用的推导过程中只有一个合法，遵循SFINAE原则，则可以顺利通过编译。

SFINAE\_test(std::string("123"));

SFINAE\_test(123);

    当第一个函数调用进行模板函数推导的时候，第一个版本的模板函数std::is\_trivial<T>::value为false，继而std::enable\_if<std::is\_trivial<T>::value>::type这个类型未定义，不能正确推导，编译器区寻找下一个可能的实现，所以接下来找到第二个模板函数，!std::is\_trivial<T>::value的值是true，继而std::enable\_if<std::is\_trivial<T>::value>::type是void类型，推导成功。这时候SFINAE\_test(std::string("123"));调用有了唯一确定的推导，即第二个模板函数，所以程序打印T is none trival。与此相似的过程，第二个函数调用打印出T is trival。  
    这样写的好处是什么？这个例子中可以认为我们利用SFINAE特性实现了通过不同返回值，相同函数参数进行了函数重载，这样代码看起来更统一一些。还有一些其他应用std::enable\_if的方式，比如在模板参数列表里，在函数参数列表里，都是利用SFINAE特性来实现某一些函数的选择推导。来看一下cpprefrence上的例子代码

#include <type\_traits>

#include <iostream>

#include <string>

namespace detail { struct inplace\_t{}; }

void\* operator new(std::size\_t, void\* p, detail::inplace\_t) {

return p;

}

// enabled via the return type

template<class T,class... Args>

typename std::enable\_if<std::is\_trivially\_constructible<T,Args&&...>::value>::type

construct(T\* t,Args&&... args)

{

std::cout << "constructing trivially constructible T\n";

}

// enabled via a parameter

template<class T>

void destroy(T\* t,

typename std::enable\_if<std::is\_trivially\_destructible<T>::value>::type\* = 0)

{

std::cout << "destroying trivially destructible T\n";

}

// enabled via a template parameter

template<class T,

typename std::enable\_if<

!std::is\_trivially\_destructible<T>{} &&

(std::is\_class<T>{} || std::is\_union<T>{}),

int>::type = 0>

void destroy(T\* t)

{

std::cout << "destroying non-trivially destructible T\n";

t->~T();

}

int main()

{

std::aligned\_union\_t<0,int,std::string> u;

construct(reinterpret\_cast<int\*>(&u));

destroy(reinterpret\_cast<int\*>(&u));

construct(reinterpret\_cast<std::string\*>(&u),"Hello");

destroy(reinterpret\_cast<std::string\*>(&u));

}

**C++ metaprogramming**