# c++11学习笔记

#include <iostream>

/\*//传值调用（类似于传文件，但不改变原文件）

void Sum(int a, int b);

int main() {

int a(2),b(3);

Sum(a,b);

std::cout << a<< std::endl;

std::cout << b<< std::endl;

return 0;

}

void Sum(int a, int b)

{

a=5;

b=10;

}\*/

/\*

//传址调用（类似于传ip地址，改变原文件）

void Sum(int \*a, int \*b);

int main() {

int a(2),b(3);

Sum(&a,&b);

std::cout << a<< std::endl;

std::cout << b<< std::endl;

return 0;

}

void Sum(int \*a, int \*b)

{

\*a=5;

\*b=10;

}\*/

void swap(int \*a,int \*b);

int main() {

int a=1,b=2;

swap(&a,&b);

cout<<a<<b<<endl;

return 0;

}

void swap(int \*a,int \*b)

{

int temp=0;

temp=\*a;

\*a=\*b;

\*b=temp;

cout<<\*a<<\*b<<endl;

}

//引用调用（实参与形参具有同一变量名，省去指针传递的复杂性）

int Sum(int a, int b, int sum);

int main() {

int a(2),b(3);

int sum(2);

sum=Sum(a,b,sum);

std::cout << sum<< std::endl;

return 0;

}

int Sum(int a, int b, int sum)

{

return sum=a+b;

}

#include <iostream>

using namespace std;

void swap(int &a,int &b);

int main() {

int a=1,b=2;

swap(a,b);

cout<<a<<b<<endl;

return 0;

}

void swap(int &a,int &b)

{

int temp=0;

temp=a;

a=b;

b=temp;

}

## C++的三大函数

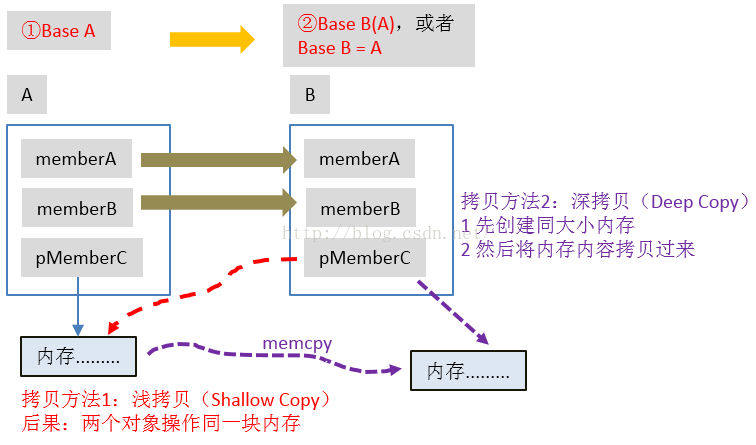
### 1.拷贝法

浅拷贝（拷贝构造函数）和深拷贝（拷贝赋值函数）

注：关于深拷贝和浅拷贝

* memberA和memberB是值拷贝。所以，A对象的memberA和memberB将赋给B的memberA和memberB。此后，A、B对象的memberA和memberB值分别相同。
* 而对pMemberC来说，情况就不一样了。B.pMemberC和A.pMemberC将指向同一块内存。如果A对这块内存进行了操作，B知道吗？更有甚者，如果A删除了这块内存，而B还继续操作它的话，岂不是会崩溃？所以，对于这种情况，拷贝构造函数中使用了所谓的深拷贝（deepcopy），也就是将A.pMemberC的内容拷贝到B对象中（B先创建一个大小相同的数组，然后通过memcpy进行内存的内容拷贝），而不是简单的进行赋值（这种方式叫浅拷贝，shallow copy）。
* 值拷贝、内容拷贝和浅拷贝、深拷贝

由上述内容可知，浅拷贝对应于值拷贝，而深拷贝对应于内容拷贝。对于非指针变量类型而言，值拷贝和内容拷贝没有区别，但对于指针型变量而言，值拷贝和内容拷贝差别就很大了。

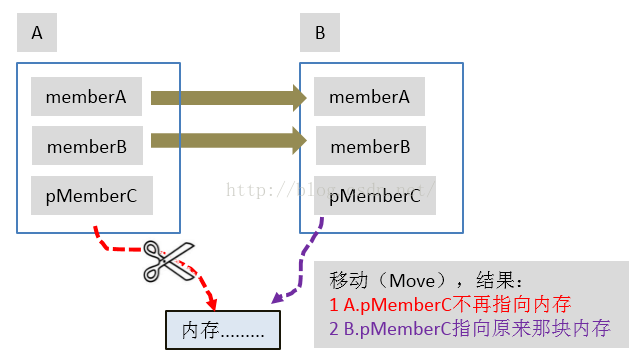


### 2.移动法

* 对于memberA和memberB而言，由于它们是非指针类型的变量，移动和拷贝没有不同。
* 但对于pMemberC而言，差别就很大了。如果使用拷贝之法，A和B对象将各自有一块内存。如果使用移动之法，A对象将不再拥有这块内存，反而是B对象拥有A对象之前拥有的那块内存。

移动的含义好像不是很难。不过，让我们更进一步思考一个问题：移动之后，A、B对象的命运会发生怎样的改变？

很简单，B自然是得到A的全部内容。A则掏空自己，成为无用之物。注意，A对象还存在，但是你最好不要碰它，因为它的内容早已经移交给了B。移动之后，A居然无用了。



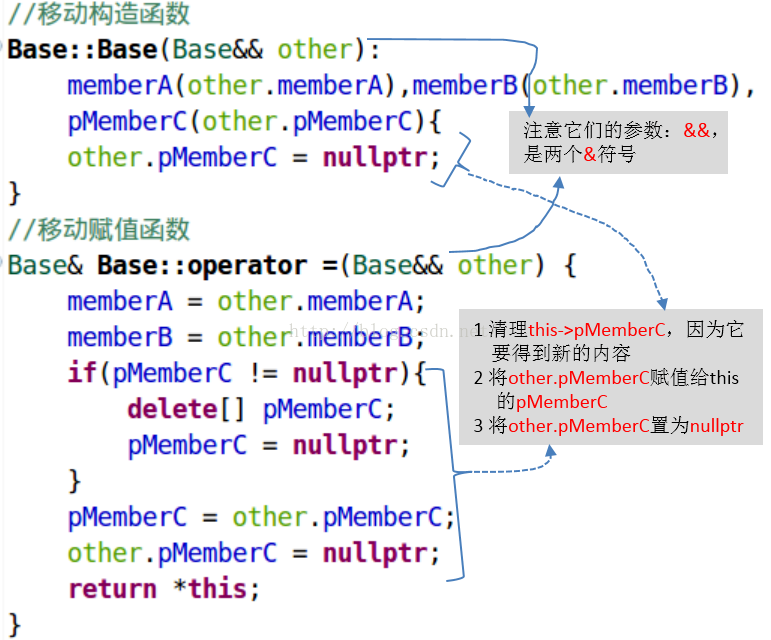
#### 移动法的优点

结合前文所述内容，我们发现tmp确实是一种转移出去（不管是采用移动还是拷贝）后就不需要再使用的对象了。对于这种情况，移动构造所带来的好处是显而易见的。

注意：

对于图中的测试函数，现在的编译器已经能做到高度优化，以至于图中列出的移动或拷贝调用都不需要了。为了达到图中的效果，编译时必须加上-fno-elide-constructors标志以禁止这种优化。

#### 移动法的代码实现和左右值类型



特别注意Base类移动构造和移动赋值函数的参数的类型，它是Base&&。没错，是两个&&符号：

如果是Base&&（两个&&符号），则表示是Base的右值引用类型。

如果是Base&（一个&符号），则表示是Base的引用类型。和右值引用相比，这种引用也叫左值引用。

什么是左值，什么是右值？笔者不拟讨论它们详细的语法和语义。不过，根据参考文献[5]所述，读者掌握如下识即可：

左值是有名字的，并且可以取地址。

右值是无名的，不能取地址。比如图19中getTemporyBase返回的那个临时对象就是无名的，它就是右值。

例如：

int a,b,c; //a,b,c都是左值

c = a+b; //c是左值，但是(a+b)却是右值，因为&(a+b)取地址不合法

getTemporyBase();//返回的是一个无名的临时对象，所以是右值

Base && x = getTemoryBase();//通过定义一个右值引用类型x，getTemporyBase函数返回的这个临时无名对象从此有了x这个名字。不过，x还是右值吗？答案为否：

Base y = x;//此处不会调用移动构造函数，而是拷贝构造函数。因为x是有名的，所以它不再是右值。

#### 移动构造/赋值函数使用场合

第一，如果确定被转移的对象（比如图中的tmp对象）不再使用，就可以使用移动构造/赋值函数来提升运行效率。

第二，我们要保证移动构造/赋值函数被调用，而不是拷贝构造/赋值函数被调用。例如，上述代码中Base y = x这段代码实际上触发了拷贝构造函数，这不是我们想要的。为此，我们需要强制使用移动构造函数，方法为Base y = std::move(x)。move是std标准库提供的函数，用于将参数类型强制转换为对应的右值类型。通过move函数，我们表达了强制使用移动函数的想法。

如果没有定义移动函数怎么办？

如果类没有定义移动构造或移动赋值函数，编译器会调用对应的拷贝构造或拷贝赋值函数。所以，使用std::move不会带来什么副作用，它只是表达了要使用移动之法的愿望。

### 3.析构函数

实例

~test()

{

if(pmber!= nullptr)

{

delete[]pmber;

pmber= nullptr;

}

} //析构函数

完整代码

#include <iostream>

#include <string.h>

using namespace std;

class test

{

public:

test(); // 默认构造函数

test(int a,int b)

{

mbera=a;

mberb=b;

}//构造函数

test(int a,int b,int temp)

{

mbera=a;

mberb=b;

pmber=new int(temp);

}//重载构造函数

test(const test &other)

{

mbera=other.mbera;

mberb=other.mberb;

} //拷贝构造函数

test &operator=(const test &other)

{

/\*

\* 浅拷贝other对象成员变量的值

\*/

this->mbera=other.mbera;

(\*this).mberb=other.mberb;

/\*

\* 深拷贝other对象指针变量的值

\*/

if(pmber!= nullptr)

{

delete []pmber;

pmber= nullptr;

}

if(other.pmber!= nullptr)

{

pmber=new int[size]; //创建新的内存空间

/\*头文件：#include <string.h>

memcpy() 用来复制内存，其原型为：

void \* memcpy ( void \* dest, const void \* src, size\_t num )\*/;

memcpy(pmber,other.pmber,size); //把other对象指针指向的内存拷贝给调用对象

}

} //拷贝赋值函数

test(test &&other) // &&是 test右值引用类型

{

mbera=other.mbera;

mberb=other.mberb;

pmber=other.pmber;

other.pmber= nullptr;

}//移动构造函数

test &operator=(test &&other)

{

mbera=other.mbera;

mberb=other.mberb;

if(pmber!= nullptr)

{

delete []pmber;

pmber= nullptr;

}

pmber=other.pmber;

other.pmber= nullptr;

return \*this;

} //移动赋值函数

void set\_mbera(int value\_a) //成员变量的封装

{

mbera=value\_a;

}

int get\_mbera()

{

return this->mbera;

}

void set\_mberb(int value\_b)

{

mberb=value\_b;

}

int get\_mberb()

{

return this->mberb;

}

int get\_pmber()

{

return \*pmber;

}

/\*

\* 析构函数,加了就报错了？

\*/

/\*~test()

{

if(pmber!= nullptr)

{

delete[]pmber;

pmber= nullptr;

}

cout << "destructing it" << endl;

} \*/

private:

int mbera;

int mberb;

static const int size=512;

int \*pmber;

};

int main()

{

/\*test p(1,2); //创建实例，调用构造函数实例化该对象

cout<<p.get\_mbera()<<endl;

cout<<p.get\_mberb()<<endl;

test q(p); //创建实例，调用拷贝构造函数实例化该对象

cout<<q.get\_mbera()<<endl;

cout<<q.get\_mberb()<<endl;\*/

/\*

\* 测试拷贝赋值函数

\*/

test p(3,4),q(10,20,30); //创建实例，调用拷贝赋值函数实例化该对象

p.operator=(q);

cout<<"P对象的数据"<<endl;

cout<<p.get\_mbera()<<endl;

cout<<p.get\_mberb()<<endl;

cout<<p.get\_pmber()<<endl;

cout<<"q对象的数据"<<endl;

cout<<q.get\_mbera()<<endl;

cout<<q.get\_mberb()<<endl;

cout<<q.get\_pmber()<<endl;

return 0;

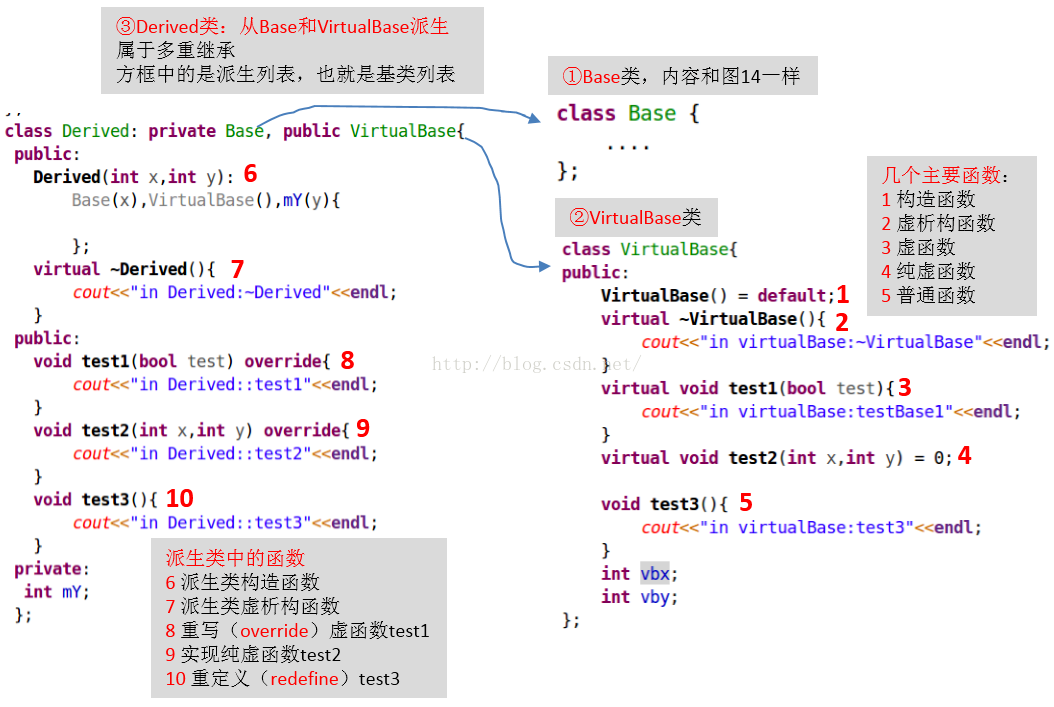
}

### 总结：

* 构造函数，分为默认构造，普通构造，拷贝构造和移动构造。
* 赋值函数，分为拷贝赋值和移动赋值。请读者先从原理上理解拷贝和移动的区别和它们的目的。
* 析构函数。

## 类的派生和继承

### 虚函数和纯虚函数（描述多态和抽象的利器）

虚函数的声明

virtual void test1(bool test);//虚函数由virtual标示

虚函数：基类定义虚函数，派生类可以重写（override）它。当我们拥有一个派生类对象，但却是通过基类引用类型或者基类指针类型的变量来调用该对象的虚函数时，被调用的虚函数是派生类重写过的虚函数（如果该虚函数被派生类重写了的话）。

虚函数被派生类重写

void test1(bool test)override

{

cout<<"in Dirve:test1"<<endl;

}

纯虚函数的声明

virtual void test2(int x,int y)=0;//纯虚函数由"virtual"和"=0"同时标示

纯虚函数：拥有纯虚函数的类不能实例化。从这一点看，它和Java的抽象类和接口非常类似。

纯虚函数被派生类实现

void test2(int x,int y)override

{

cout<<"in Dirve:test2"<<endl;

}

注：1.建议在派生类中重写基类虚函数时都加上override和virtual关键字

2.virtual和override标示只在类中声明函数时需要。如果在类外实现该函数，则并不需要这些关键词

### **虚析构函数**

除了上述两类虚函数外，C++中还有虚析构函数。虚析构函数其实就是虚函数，不过它稍微有一点特殊，需要开发者注意：

1.虚函数被override的时候，基类和派生类声明的虚函数在函数名，参数等信息上需保持一致。但对析构函数而言，由于析构函数的函数名必须是"~类名"，所以派生类和基类的析构函数名肯定是不同的。

但是，我们又希望多态对于析构函数（注意，析构函数也是函数，和普通函数没什么区别）也是可行的。比如，当通过基类指针来删除派生类对象时，是派生类对象的析构函数被调用。所以，当基类中如果有虚函数时候，一定要记得将其析构函数变成虚析构函数。

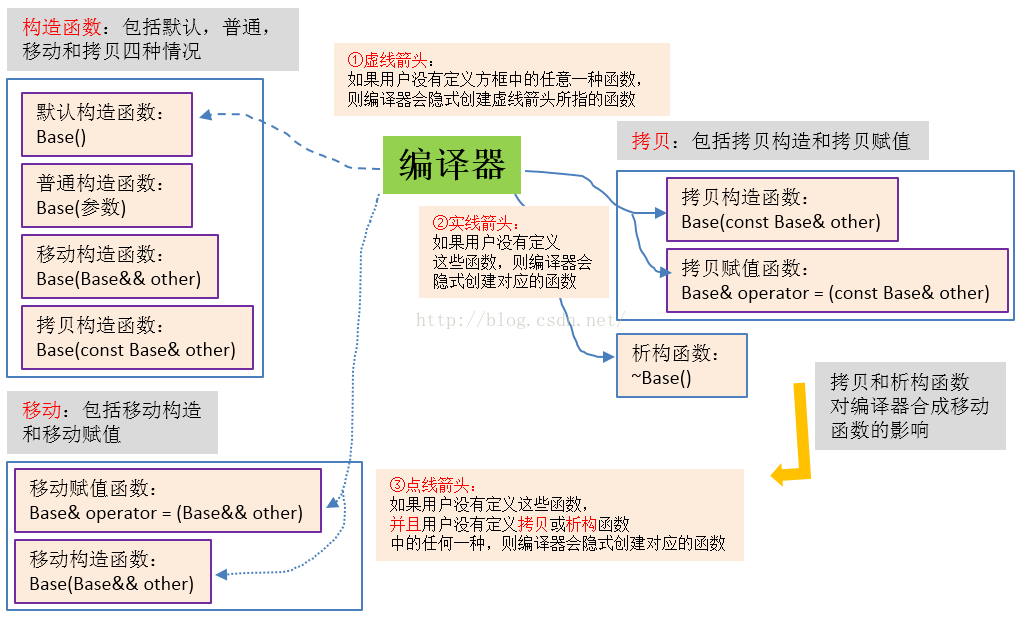
2.阻止虚函数被override。C++中，也可以阻止某个虚函数被override，方法和Java类似，就是在函数声明后添加final关键词，比如

virtual void test1(boolean test) final;//如此，test1将不能被派生类override了

最后，我们通过一段示例代码来加深对虚函数的认识

* 如果想实现多态，就在基类中为需要多态的函数增加virtual关键词。
* 如果基类中有虚函数，也请同时为基类的析构函数添加virtual关键词。只有这样，指向派生类对象的基类指针变量被delete时，派生类的析构函数才能被调用。

## 编译器合成函数

可简单总结为：

* 如果程序员定义了任何一种类型的构造函数（拷贝构造、移动构造，默认构造，普通构造），则编译器将不再隐式创建默认构造函数。
* 如果程序没有定义拷贝（拷贝赋值或拷贝构造）函数或析构函数，则编译器将隐式合成对应的函数。
* 如果程序没有定义移动（移动赋值或移动构造）函数，并且，程序没有定义析构函数或拷贝函数（拷贝构造和拷贝赋值），则编译器将合成对应的移动函数。

### 1. =delete和=default

有些时候我们需要一种方法来控制编译器这种自动合成的行为，控制的目的无外乎两个：

* 让编译器必须合成某些函数。
* 禁止编译器合成某些函数。

借助=default和=delete标识，这两个目的很容易达到，来看一段代码：

//定义了一个普通的构造函数，但同时也想让编译器合成默认的构造函数，则可以使用=default标识

Base(int x); //定义一个普通构造函数后，编译器将停止自动合成默认的构造函数

//=default后，强制编译器合成默认的构造函数。注意，开发者不用实现该函数

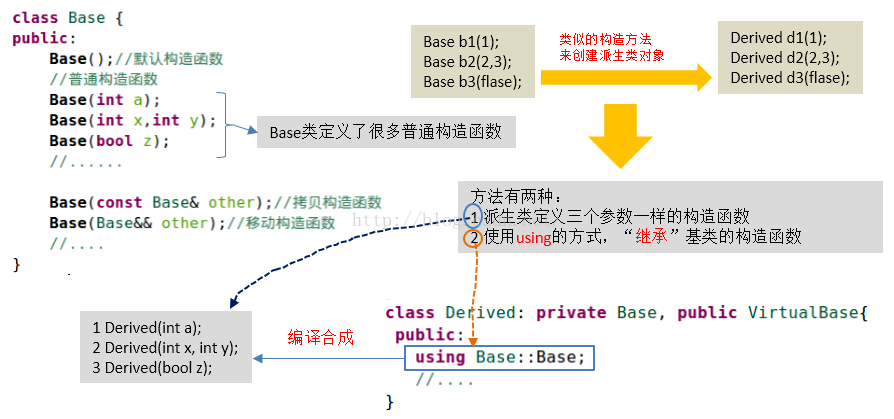
Base() = default;//通知编译器来合成这个默认的构造函数

//如果不想让编译器合成某些函数，则使用= delete标识

Base&operator=(const Base& other) = delete;//阻止编译合成拷贝赋值函数

注意，这种控制行为只针对于构造、赋值和析构等三类特殊的函数。

### 2.“继承”基类的构造函数



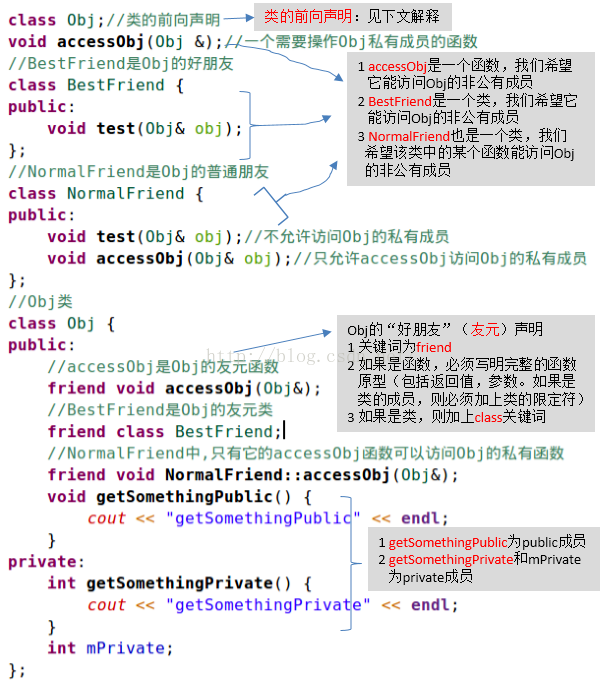
* 第一种方法就是在Derived派生类中手动编写三个构造函数，这三个构造函数和Base类里的一样。
* 另外一种方法就是通过使用using关键词“继承”基类的那三个构造函数。继承之后，编译器会自动合成对应的构造函数。

注意，这种“继承”其实是一种编译器自动合成的规则，它仅支持合成普通的构造函数。而默认构造函数，移动构造函数，拷贝构造函数等遵循正常的规则来合成。

## 友元和类的前向声明

关于友元：

C++中，借助友元，我们可以做到小范围的公开信息以减少沟通成本。从编程角度来看，友元的作用无非是：提供一种方式，使得类外某些函数或者某些类能够访问一个类的私有成员变量或成员函数。对被访问的类而言，这些类外函数或类，就是被访问的类的朋友。



C++中，类的友元可以是：

* 一个类外的函数或者一个类中的某些成员函数。如果友元是函数，则必须指定该函数的完整信息，包括返回值，参数，属于哪个类等。
* 一个类。

基类的友元会变成从该基类派生得来的派生类的友元吗？

C++中，友元关系不能继承，也就是说：

1 基类的友元可以访问基类非公开成员，也能访问派生类中属于基类的非公开成员。

2 但是不能访问派生类自己定义的非公开成员。

友元比较简单，此处就不拟多说。现在我们介绍下图中提到的类的前向声明，先来回顾下代码：

class Obj;//类的前向声明

void accessObj(Obj& obj);

C++中，数据类型应该先声明，然后再使用。但这会带来一个“先有鸡还是先有蛋”的问题：

accessObj函数的参数中用到了Obj。但是类Obj的声明却放在图的最后。

如果把Obj的声明放在accessObj函数的前面，这又无法把accessObj指定为Obj的友元。因为友元必须要指定完整的函数。

怎么破解这个问题？这就用到了类的前向声明，以图为例，Obj前向声明的目的就是告诉类型系统，Obj是一个class，不要把它当做别的什么东西。一般而言，类的前向声明的用法如下：

假设头文件b.h中需要引入a.h头文件中定义的类A。但是我们不想在b.h里包含a.h。因为a.h可能太复杂了。如果b.h里包含a.h，那么所有包含b.h的地方都间接包含了a.h。此时，通过引入A的前向声明，b.h中可以使用类A。

注意，类的前向声明一种声明，真正使用的时候还得包含类A所在的头文件a.h。比如，b.cpp（b.h相对应的源文件）是真正使用该前向声明类的地方，那么只要在b.cpp里包含a.h即可。

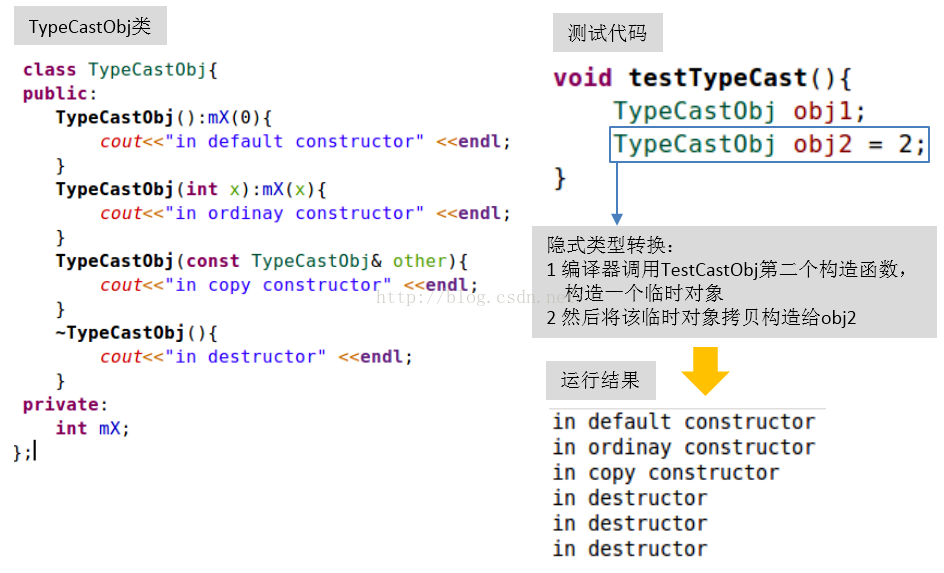
这就是类的前向声明的用法，即在头文件里进行类的前向声明，在源文件里去包含该类的头文件。

类的前向声明的局限

前向声明好处很多，但同时也有限制。以Obj为例，在看到Obj完整定义之前，不能声明Obj类型的变量（包括类的成员变量），但是可以定义Obj引用类型或Obj指针类型的变量。比如，你无法在图中class Obj类代码之前定义ObjaObj这样的变量。只能定义Obj& refObj或Obj\* pObj。之所以有这个限制，是因为定义Obj类型变量的时候，编译器必须确定该变量的大小以分配内存，由于没有见到Obj的完整定义，所以编译器无法确定其大小，但引用或者指针则不存在此问题。

## explicit构造函数

explicit构造函数和类型的隐式转换有关。

图中测试代码里，编译器进行了隐式类型转换，即先用常量2构造出一个临时TypeCastObj对象，然后再拷贝构造为obj2对象。注意，支持这种隐式类型转换的类的构造函数需要满足一个条件：

* 类的构造函数必须只能有一个参数。如果构造函数有多个参数，则不能隐式转换。

注意：

TypeCastObj obj3(3) ;//这样的调用是直接初始化，不是隐式类型转换

如果程序员不希望发生这种隐式类型转换该怎么办？只需要在类声明中构造函数前添加explicit关键词即可，比如：

explicit TypeCastObj(intx) :mX(x){

cout<<"in ordinay constructor"<<endl;

}

## 操作符重载

对Java程序员而言，操作符重载是一个陌生的话题，因为Java语言并不支持它[⑥]。相反，C++则灵活很多，它支持很多操作符的重载。为什么两种语言会有如此大相径庭的做法呢？关于这个问题，前文也曾从面向对象和面向数据类型的角度探讨过：

从面向对象的角度看，两个对象进行加减乘除等操作会得到什么？不太好回答，而且现实生活中好像也没有可以类比的案例。

但如果从数据类型的角度看，既然普通的数据类型可以支持加减乘除，类这种自定义类型为什么又不可以呢？

### 操作符重载的实现方式

操作符重载说白了就是将操作符当成函数来对待。当执行某个操作符运算时，对应的操作符函数被调用。和普通函数比起来，操作符对应的函数名由“operator 操作符的符号”来标示。

既然是函数，那么就有类的成员函数和非类的成员函数之分，C++中：

有一些操作符重载必须实现为类的成员函数，比如->，\*操作符。

有一些操作符重载必须实现为非类的成员函数，比如<<和>>操作符。

有一些操作符即可以实现为类的成员函数，也可以实现为非类的成员函数，比如加减乘除运算符。具体采用哪种方式，视习惯做法或者方便程度而定。

本节先来看一个可以采用两种方式来重载的加操作符的示例，如图所示：



Obj类定义了两个+号重载函数，分别实现一个Obj类型的变量和另外一个Obj类型变量或一个int型变量相加的操作。同时，我们还定义了一个针对Obj类型和布尔类型的+号重载函数。+号重载为类成员函数或非类成员函数均可，程序员应该根据实际需求来决定采用哪种重载方式。下面是一段测试代码：

Obj obj1, obj2;

obj1 = obj1+obj2;//调用Obj类第一个operator+函数

int x = obj1+100;//调用Obj类第二个operator+函数

x = obj1.operator+(1000); //显示调用Obj类第二个operator+成员函数

int z = obj1+true;//调用非类的operator+函数

强调：

实际编程中，加操作符一般会重载为类的成员函数。并且，输入参数和返回值的类型最好都是对应的类类型。因为从“两个整型操作数相加的结果也是整型”到“两个Obj类型操作数相加的结果也是Obj类型”的推导是非常自然的。上述示例中，笔者有意展示了操作符重载的灵活性，故而重载了三个+操作符函数。

### 输入和输出操作符重载

"cout<<....<<..."和"cin>>...>>.."这样的代码得以成功实现。

C++的>>和<<操作符已经实现了内置数据类型和某些类类型（比如STL标准类库中的某些类）的输出和输入。如果想实现用户自定义类的输入和输出则必须重载这两个操作符。来看一个例子，如图所示：



通过图中的重载，我们可以通过标准输入输出来操作Obj类型的对象了。

比较：

<<输出操作符重载有点类似于我们在Java中为某个类重载toString函数。toString的目的是将类实例的内容转换成字符串以方便打印或者别的用途。

### ->和\*操作符的重载

->和\*操作符重载一般用于所谓的智能指针类，它们必须实现为类的成员函数

注：这两个操作符如果操作的是指针类型的对象，则并不是重载。

//假设Object类重载了->和\*操作符

Object \*pObject =new Object();//new一个Object对象

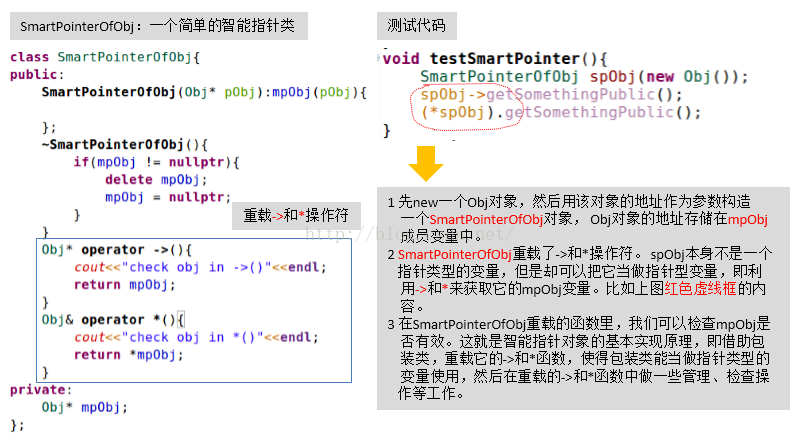
//下面的->操作符并非重载。因为pObject是指针类型，所以->只是按照标准语义访问它的成员

pObject->getSomethingPublic();

//同理，pObject是指针类型，故\*pObject就是对该地址的解引用，不会调用重载的\*操作符函数

(\*pObject).getSomethingPublic();

按照上述代码所说，对于指针类型的对象而言，->和\*并不能被重载，那这两个操作符的重载有什么作用？



实现了一个用于保护某个new出来的Obj对象的SmartPointerOfObj类，通过重载SmartPointerOfObj的->和\*操作符，我们就好像直接在操作指针型变量一样。在重载的->和\*函数中，程序员可以做一些检查和管理，以确保mpObj指向正确的地址，目的是避免操作无效内存。这就是一个很简单的智能指针类的实现。

### new和delete操作符重载

new和delete操作符的重载与其他操作符的重载略有不同。平常我们所说的new和delete实际上是指new表达式（expression）以及delete表达式，比如：

Object\* pObject =new Object; //new表达式，对于数组而言就是new Object[n];

delete pObject;//delete表达式，对于数组而言就是delete[] pObject

上面这两行代码分别是new表达式和delete表达式，这两个表达式是不能自定义的，但是：

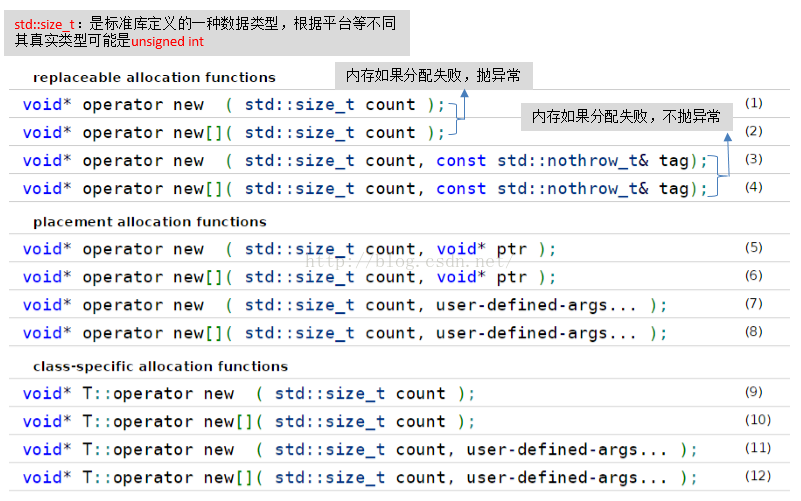
1.new表达式执行过程中将首先调用operator new函数。而C++允许程序员自定义operator new函数。

2.delete表达式执行过程的最后将调用operator delete函数，而程序员也可以自定义operator delete函数。

所以，所谓new和delete的重载实际上是指operator new和operator delete函数的重载。下面我们来看一下operator new和operator delete函数如何重载。

#### 1. new 和delete操作符语法

#### new操作符的语法

new操作符一共有12种形式，用法相当灵活，其中：

* 程序员可以重载（1）到（4）这四个函数。这四个函数是全局的，即它们不属于类的成员函数。有些new函数会抛异常，不过笔者接触的程序中都没有使用过C++中的异常，所以本书不拟讨论它们。
* （5）到（8）为placement new系列函数。placement new其实就是给new操作符提供除内存大小之外（即count参数）的别的参数。比如“new(2,f)T”这样的表达式将对应调用operator new(sizeof(T), 2, f)函数，注意，这几个函数也是系统全局定义的。另外，C++规定（5）和（6）这两个函数不允许全局重载。
* （9）到（12）定义为类的成员函数。注意，虽然上边的（5）和（6）不能进行全局重载，但是在类中却可以重载它们。

务必注意，如果我们在类中重载了任意一种new操作符，那么系统的new操作符函数将被隐藏。隐藏的含义是指编译器如果在类X中找不到匹配的new函数时，它也不会去搜索系统定义的匹配的new函数，这将导致编译错误。

注意：何谓“隐藏”？

http://en.cppreference.com/w/cpp/memory/new/operator\_new提到了只要类重载任意一个new函数，都将导致系统定义的new函数全部被隐藏。关于“隐藏”的含义，经过笔者测试，应该是指编译器如果在类中没有搜索到合适的new函数后，将不会主动去搜索系统定义的new函数，如此将导致编译错误。

如果不想使用类重载的new操作符的话，则必须通过::new的方式来强制使用全局new操作符。其中，::是作用域操作符，作用域可以是类（比如Obj::）、命名空间（比如stl::），或者全局（::前不带名称）。

综上所述，new操作符重载很灵活，也很容易出错。所以建议程序员尽量不要重载全局的new操作符，而是尽可能重载特定类的new操作符（图中的（9）到（12））。

#### delete操作符的语法

delete用法比new还要复杂。此处需要特别说明的是：

* new表达式可以带参数，比如new(2,f)T。
* 但delete表达式不能传递参数。所以像图中带参数的delete操作符函数，比如（7）到（10），（15）、（16）这几个函数将如何调用呢？C++规范里说，当使用对应形式的new操作符构造一个或一组类实例时，如果其中有一个实例的构造函数抛出异常，那么对应形式的delete操作符函数将被调用。

上面的描述不太直观，我们通过一个例子进一步来解释它。



图中：

* 类X的构造函数抛出一个异常。
* 类X重载了一个new操作符和一个delete操作符。这两个操作符函数最后一个参数都是bool型。

main函数中，使用placementnew表达式触发了类X的new操作符被调用。

* 由于X构造函数抛出异常，所以系统会调用X重载的delete函数，也就是最后一个 参数是bool的那个delete函数。

图中还特别指出代码中不能直接使用delete p1这样的表达式，这会导致编译错误，提示没有匹配的delete函数，这是因为：

类重载的delete函数有参数，这个函数只能在类实例构造时抛出异常时调用。而类X没有定义如图中（11）或（13）所示的delete函数。

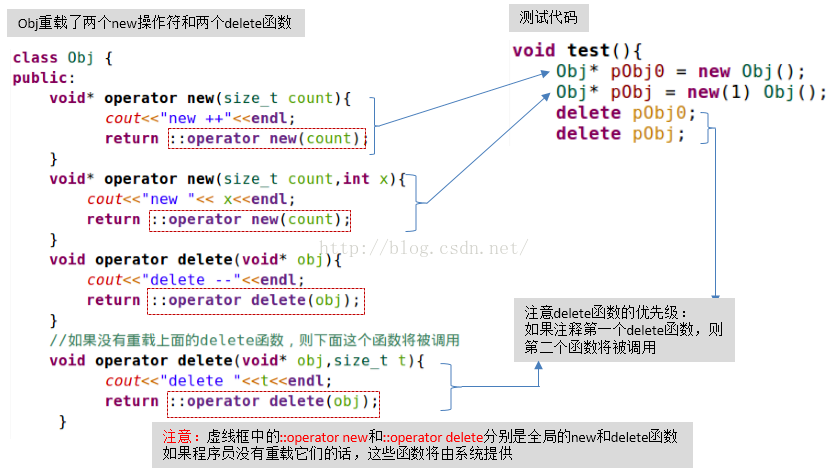
并且，类只要重定义任何一个delete函数，这都将导致系统的delete函数被隐藏。

#### 2.new和delete操作符重载示例

现在我们来看new和delete操作符重载的一个简单示例。如图所示：

强调：

考虑到new和delete的高度灵活性以及和它们和内存分配释放紧密相关的重要性，程序员最好只针对特定类进行new和delete操作符的重载。



图中，笔者为Obj重载了两个new操作符和两个delete操作符：

当像测试代码中那样创建Obj实例时，这两个new操作符重载函数分别会被调用。

delete函数略有特殊，它存在优先级的问题。第一个delete函数优先级高于第二个delete函数。如果第一个delete函数被注释，那么第二个delete函数将被调用。

#### 3.重载new和delete操作符的好处

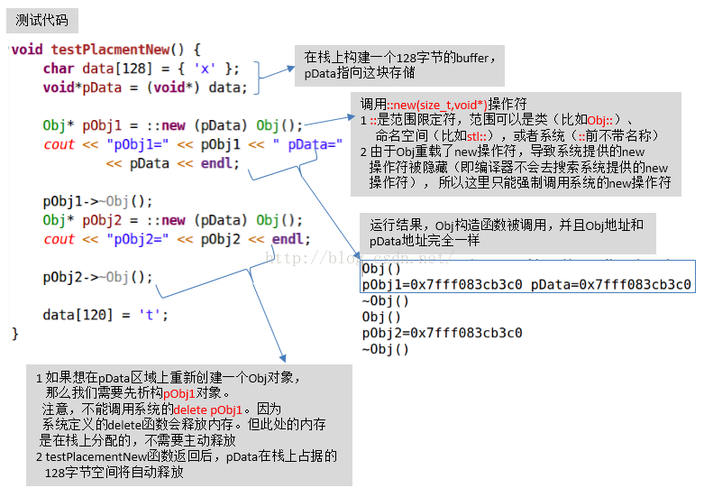
通过重载new和delete操作符，我们有机会在对象创建和释放的时候做一些内存管理的工作。比如，每次new一个Obj对象，我们递增new被调用的次数。delete的时候再递减。当程序退出时，我们检查该次数是否归0。如果不为0，则表示有Obj对象没有被delete，这很可能就是内存泄露的潜在原因。

## 如何在指定内存中构造对象

我们用new表达式创建一个对象的时候，系统将在堆上分配一块内存，然后这个对象在这块内存上被构造。由于这块内存分配在堆上，程序员一般无法指定其地址。这一点和Java中的new类似。但有时候我们希望在指定内存上创建对象，可以做到吗？对于C++这种灵活度很高的语言而言，这个小小要求自然可以轻松满足。只要使用特殊的new即可：

void\* operator new(size\_t count, void\* ptr):它是placement new中的一种。此函数第二个参数是一个代表内存地址的指针。该函数的默认实现就是直接将ptr作为返回的内存地址，也就是将传入的内存地址作为new的结果返回给调用者。

使用这种方式的new操作符时，由于返回的内存地址就是传进来的ptr，这就达到了在指定内存上构造对象的功能。马上来看一个示例，如图所示：



图展示了placement new的用法，即在指定内存中构造对象。这个指定内存是在栈上创建的。另外，对于这种方式创建的对象，如果要delete的话必需小心，因为系统提供的delete函数将回收内存。在本例中，对象是构造在栈上的，其占据的内存随testPlacementNew函数返回后就自动回收了，所以图中没有使用delete。不过请读者务必注意，这种情况下内存不需要主动回收，但是对象是需要析构的。

显然，这种只有new没有delete的使用方法和平常用法不太匹配，有点别扭。如何改进呢？方法很简单，我们只要按如下方式重载delete操作符，就可以在图的实例中使用delete了：

//Class Obj重载delete操作符

void operator delete(void\* obj){

cout<<"delete--"<<endl;

//return ::operator delete(obj);屏蔽内存释放，因为本例中内存在栈上分配的

}//读者可以自行修改测试案例以加深对new和delete的体会。

如果Obj类按如上方式重载了delete函数，我们在图的代码中就可以“delete pObj1”了。

探讨：重载new和delete的好处

一般情况下，我们重载new和delete的目的是将内存创建和对象构造分隔开来。这样有什么好处呢？比如我们可以先创建一个大的内存，然后通过重载new函数将对象构造在这块内存中。当程序退出后，我们只要释放这个大内存即可。

另外，由于内存创建和释放与对象构造和析构分离了开来，对象构造完之后切记要析构，delete表达式只是帮助我们调用了对象的析构函数。如果像本例那样根本不调用delete的话，就需要程序员主动析构对象。

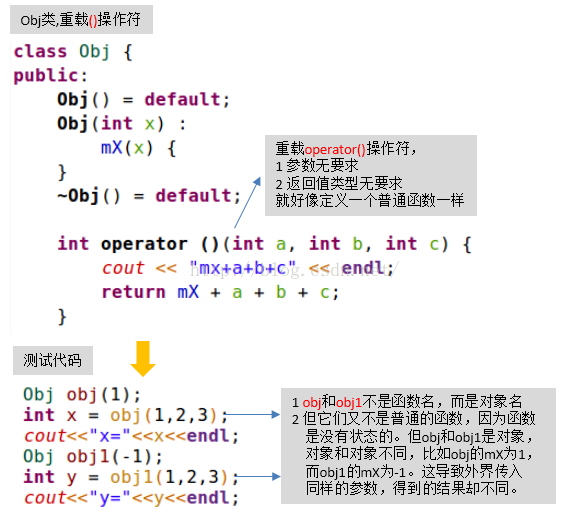
ART中，有些基础性的类重载了new和delete操作符，它们的实例就是用类似方式来创建的。以后我们会见到它们。

最后，new和delete是C++中比较复杂的一个知识点。关于这一块的内容，笔者觉得参考文献里列的几本书都没有说太清楚和全面。请意犹未尽的读者阅读如下两个链接的内容：

<http://en.cppreference.com/w/cpp/memory/new/operator_new>

<http://en.cppreference.com/w/cpp/memory/new/operator_delete>

## 函数调用符重载



函数调用运算符使得对象能像函数一样被调用，什么意思呢？我们知道C++和Java一样，函数调用的写法是“函数名(参数)”。如果我们把函数名换成某个类的对象，即“对象(参数)”，就达到了对象像函数一样被调用的目的。这个过程得以顺利实施的原因是C++支持函数调用运算符的重载，函数调用运算符就是“()”。

此操作符的重载比较简单，就和定义函数一样可以根据需要定义参数和返回值。

函数调用操作符重载后，Obj类的实例对象就可以像函数一样被调用了。我们一般将这种能像函数一样被调用的对象叫做函数对象。普通函数是没有状态的，但是函数对象却不一样。函数对象首先是对象，然后才是可以像函数一样被调用，而对象是有所谓的“状态”的。比如图中的obj和obj1，两个对象的mX取值不同，这将导致外界传入一样的参数却得到不同的调用结果。

## 函数模板和类模板

介绍模板具体知识之前，笔者先分享几点关于模板的非常重要的学习心得：

* C++是面向对象的语言。面向对象最重要的一个特点就是抽象，即将公共的属性、公共的行为抽象到基类中去。这种抽象非常好理解，现实生活中也无处不在。反观模板，它其实也是一种抽象，只不过这种抽象的关注点不在属性，不在行为，而在于数据类型。比如，有一个返回两个操作数相加之和的函数，它即可以处理int型操作数，也可以处理long型操作数。那么，从数据类型的角度进行抽象的话，我们可以用一个代表通用数据类型的T做为该函数的参数类型，该函数内部只对T类型的变量进行相加。至于T具体是什么，此时不用考虑。而使用这个函数的时候，当传入int型变量时，T就变成int。当传入long型变量时，T就变成long。所以，模板的重点在于将它所操作的数据的类型抽象出来！
* C++是强类型的语言，即所有变量（包括函数参数，返回值）都需要有一个明确的类型。这个要求对于模板这种基于数据类型的抽象方式有重大和直接的影响。对于模板而言，定义函数模板或类模板时所用的数据类型只是一个标示，比如前面提到的T。而真正的数据类型只有等使用者用具体的数据类型来使用模板时才能确定。相比非模板编程，模板编程多了一个非常关键的步骤，即模板实例化（英文叫instantiation）。模板实例化是编译器发现使用者用具体的数据类型来使用模板时，它就会将模板里的通用数据类型替换成具体的数据类型，从而生成实际的函数或类。比如前面提到的两个操作数相加的模板函数，当传入int型变量时，模板会实例化出一个参数为int型的函数，当传入long型变量时，模板又会实例化出一个参数为long型的函数。当然，如果没有地方用具体数据类型来使用这个模板，则编译器不会生成任何函数。注意，模板的实例化是由编译器来做的，但触发实例化的原因是因为使用者用具体数据类型来使用了某个模板。

简而言之，对于模板而言，程序员需要重点关注两个事情，一个是对数据类型进行抽象，另一个是利用具体数据类型来绑定某个模板以将其实例化。

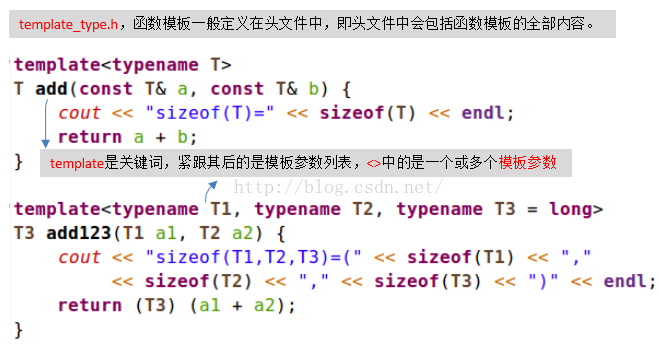
好了，让我们正式进入模板的世界，故事先从简单的函数模板开始。

提示：

模板编程是C++中非常难的部分，参考文献[4]用了六章来介绍与之相关的知识点。不管怎样，模板的核心依然是笔者前面提到的两点，一个是数据类型抽象，一个是实例化。

### 1.函数模板

#### 函数模板的定义

图所示为两个函数模板的定义，其中有几点需要读者注意：

* 函数模板一般在头文件中定义，这和普通函数不太一样。普通函数一般在头文件中声明，在源文件中定义。对函数模板而言，因为编译器在实例化一个模板的时候需要知道函数模板的全部内容（再次强调，实例化就是编译器用具体数据类型套用到模板上去，然后生成具体函数的过程），所以实例化过程中只知道函数模板的声明是不够的。更进一步得说，其实函数模板并不是真正的函数，只有编译器用具体数据类型套用到函数模板时才会生成实际的函数。
* 模板的关键词是template，其后通过<>符号包含一个或多个模板参数。模板参数列表不能为空。模板参数和函数参数有些类似，可以定义默认值。比如图中add123最后一个模板参数T3，其默认值是long。

提示：

图中的函数模板定义中，template可以和其后的代码位于同一行，比如：

template<typename T> T add(const T&a1,const T& a2);

建议开发者将其分成两行，因为这样的代码阅读起来会更容易一些。

下面继续讨论template和模板参数：

首先，可以定义任意多个模板参数，模板参数也可以像函数参数那样有默认值。

其次，函数的参数都有数据类型。类似，模板参数（如上面的T）也有类型之分：

² 代表数据类型的模板参数：用typename关键词标示，表示该参数代表数据类型，实例化时应传入具体的数据类型。比如typename T是一个代表数据类型的模板参数，实例化的时候必须用数据类型来替代T（或者说，T的取值为数据类型，比如int，long之类的）。另外，typename关键词也可以用class关键词替代，所以"template<class T>"和"template<typenameT>"等价。建议读者尽量使用typename作为关键词。

² 非数据类型参数：非数据类型的参数支持整型、指针（包括函数指针）、引用。但是这些参数的值必须在实例化期间（也就是编译期）就能确定。

关于非类型参数，此处先展示一个简单的示例，后续介绍类模板时会碰到具体用法。

//下面这段代码中，T是代表数据类型的模板参数，N是整型，compare则是函数指针

//它们都是模板参数。

template<typename T,int N,bool (\*compare)(constT & a1,const T &a2)>

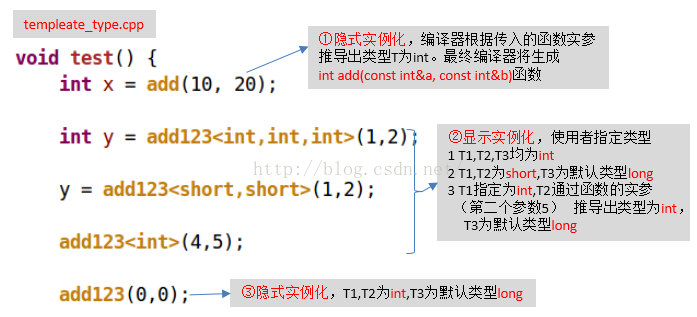
void comparetest(const T& a1,const T& a2){

cout<<"N="<<N<<endl;

compare(a1,a2);//调用传入的compare函数

#### 函数模板的实例化

下图为上图函数模板的实例化

图所示为add和add123这两个函数模板的实例化示意。结合前文反复强调的内容，函数模板的实例化就是当程序用具体数据类型来使用函数模板时，编译器将生成具体的函数：

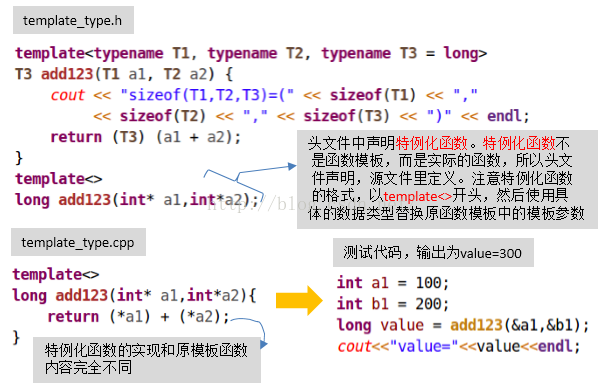
* 比如①，编译器根据传入的函数实参推导出数据类型为T，从而会生成一个"intadd(const int &b,const int &b)"函数，最终调用的也是这个生成的函数。这是编译器根据函数实参自动推导出来的，叫模板实参推导。推导过程有一些规则，属于比较高级的话题，笔者不拟讨论。不过，不论推导规则有多复杂，其目的就是为了确定模板参数的具体取值情况，这一点请读者牢记。
* 使用者也可以显示实例化，即显示指明模板参数的类型。比如②中所示的三个函数。编译器将生成三个不同的add123函数。
* add123函数模板也可以隐式实例化，比如③所示。但请读者注意，模板实参的推导只能根据传入的函数参数来确定，不能根据函数的返回值来确定。如果add123函数模板中没有为T3设置默认类型的话，编译将出错。

#### 函数模板的特例化

上文介绍了函数模板的实例化，实例化就是指编译器进行类型推导，然后得到具体的函数。实例化得到的这些函数除了数据类型不一样之外，函数内部的功能是完全一样的。有没有可能为某些特定的数据类型提供不一样的函数功能？

显然，C++是支持这种做法的，这也被称为模板的特例化（英文简称specialization）。特例化就是当函数模板不太适合某些特定数据类型时，我们单独为它指定一套代码实现。

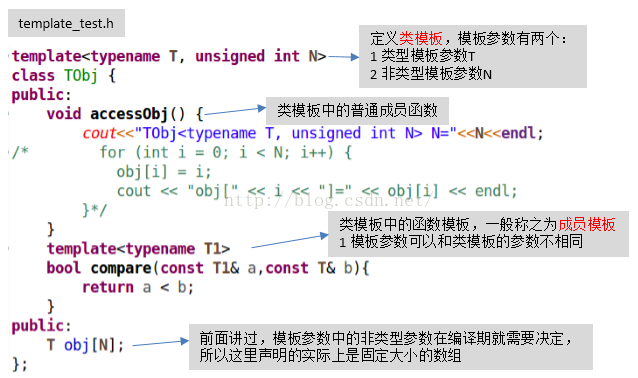
读者可能会觉得很奇怪，为什么会有这种需求？以图中的add123为例，如果程序员传入的参数类型是指针的话，显然我们不能直接使用add123原函数模板的内容（那样就变成了两个指针值的相加），而应该单独实现一个针对指针类型的函数实现。要达到这个目的就需要用到特例化了。来看具体的做法，如图所示：



### 2.类模板

#### 类模板的定义和特例化

类模板的规则比函数模板要复杂，我们来看一个例子



图中定义一个类模板，其语法格式和函数模板类型，class关键字前需要由template<模板参数>来修饰。另外，类模板中可以包含普通的成员函数，也可以有成员模板。这导致类模板的复杂度（包括程序员阅读代码的难度）大大增加。

注意:普通类也能包含成员模板，这和函数模板类似，此处不拟详述。

接着来看类模板的特例化，它分为全特化和偏特化两种情况，如图所示：



图中展示了类模板的全特化和偏特化，其中：

* 全特化和前文介绍的函数模板的特例化类似，即所有模板参数都指定具体类型或值。全特化类模板得到的是一个实例化的类。
* 偏特化就是为模板参数中的几个参数指定具体类型或值，剩下的模板参数依然由使用者来指定。注意，偏特化一个类模板得到的依然是类模板。

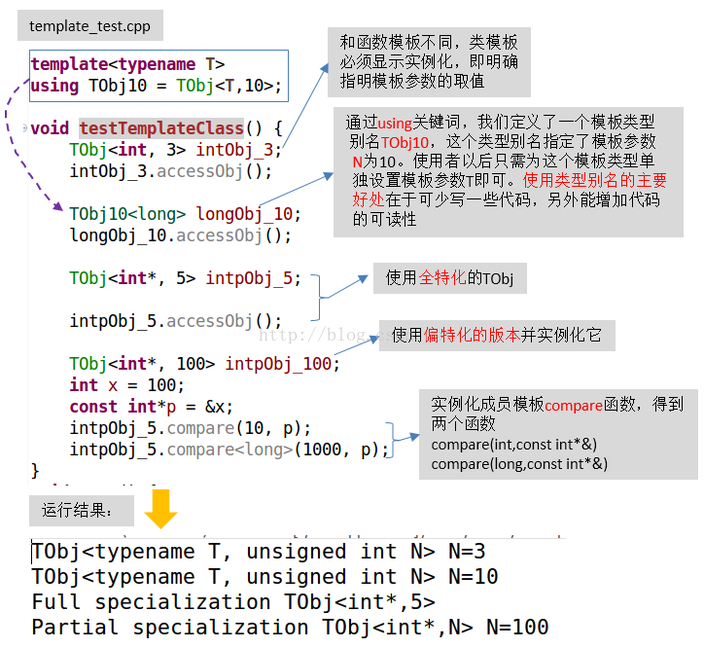
偏特化也叫部分特例化（partial specialization）。但笔者觉得“部分特例化”有些言不尽意，因为偏特化不仅仅包括“为部分模板参数指定具体类型”这一种情况，它还可以为模板参数指定某些特殊类型，比如：

template<typename T> class Test{}//定义类模板Test，包含一个模板参数

//偏特化Test类模板，模板参数类型变成了T\*。这就是偏特化的第二种表现形式

template<typename T> class<T\*> Test{}

#### 类模板的使用



值得关注的是C++11中程序员可通过using关键词定义类模板的别名。并且，使用类模板别名的时候可以指定一个或多个模板参数。

最后，类模板的成员函数也可以在类外（即源文件）中定义，不过这会导致代码有些难阅读，下图展示了如何在类外定义accessObj和compare函数



源文件中定义类模板的成员函数时需要携带类模板的模板参数信息。如果成员函数又是函数模板的话，还得加上函数模板的模板参数信息。这些模板信息放在一起很容易让代码阅读者头晕。

类模板全特化后得到是具体的类，所以它的成员函数前不需要template关键词来修饰。

类模板成员函数内部如果需要定义该类模板类型的变量时，只需使用类名，而不需要再携带模板信息了。

### lambda表达式

C++11引入了lambda表达式（lambda expression），这比Java直到Java 8才正式在规范层面推出lambda表达式要早三年左右。lambda表达式和另一个耳熟能详的概念closure（闭包）密切相关，而closure最早被提出来的目的也是为了解决数学中的lambda演算（λ calculus）问题。从严格语义上来说，closure和lambda表达式并不完全相同，不过一般我们可以认为二者描述得是同一个东西。

提示：closure和lambda的区别

关于二者的区别，读者可参考Effective C++作者Scott Meyers的一篇博文，地址如下：

http://scottmeyers.blogspot.com/2013/05/lambdas-vs-closures.html

我们在“函数调用运算符重载”一节中曾介绍过函数对象，函数对象是那些重载了函数调用操作符的类的实例，和普通函数比起来：

函数对象首先是一个对象，所以它可以通过成员变量来记录状态，保存信息。

然后，函数对象可以被执行。

通过上面的描述，我们知道函数对象的两个特点，一个是可以保存状态，另外一个是可以执行。不过，和函数一样，程序员要使用函数对象的话，首先要定义对应的类，然后才能创建该类的实例并使用它们。

现在我们来思考这样一个问题，可不可以不定义类，而是直接创建某种东西，然后可以执行它们？

Java中有匿名内部类可以做到类似的效果。但Java中的类无法重载函数调用操作符，所以匿名内部类不能像函数调用那样执行。

Java的匿名内部类给了C++一个很好的启示，由于C++是支持重载函数调用操作符的，如果我们能在C++中定义匿名函数对象，就能达到所要求的目标了。

以上问题的讨论就引出了C++中的lambda表达式，规范中没有明确说明lambda表达式是什么，但实际上它就是匿名函数对象。下面的代码展示了创建一个lambda表达式的语法结构：

auto f = [ 捕获列表，英文叫capture list ] ( 函数参数 ) ->返回值类型 { 函数体 }

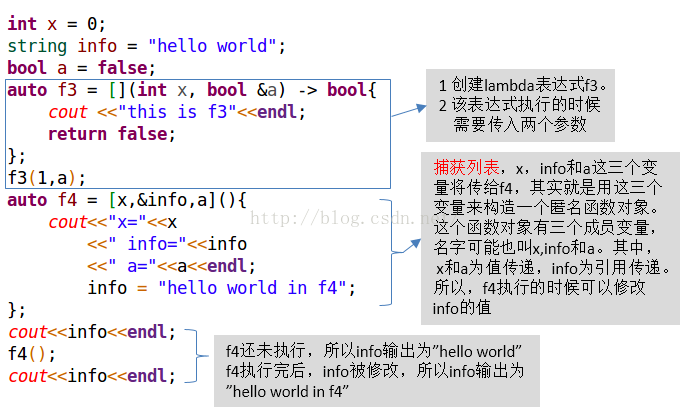
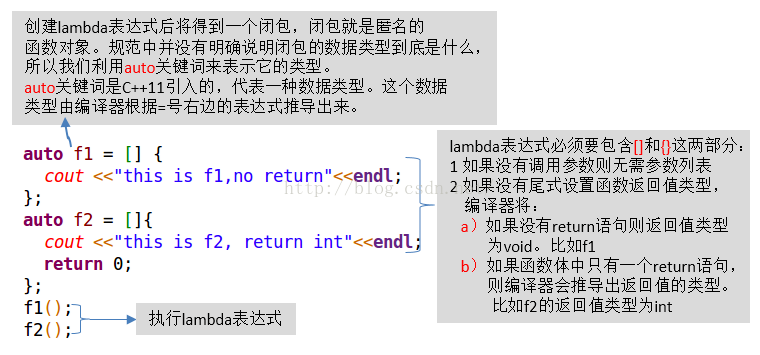
其中：

=号右边是lambda表达式，左边是变量定义，变量名为f。lambda表达式创建之后将得到一个匿名函数对象，规范中并没有明确说明这个对象的具体数据类型是什么，所以一般用auto来表示它的类型。注意，auto并不是类型名，它仅表示把具体类型的推导交给编译器来做。简而言之，lambda表达式得到的这个匿名对象是有类型的，但是类型叫什么不知道，所以程序员只好用auto来表示它的类型，反正它的具体类型会由编译器在编译时推导出来。

捕获列表：lambda表达式一般在函数内部创建。它要捕获的东西也就是函数内能访问的变量（比如函数的参数，在lambda表达式创建之前所定义的变量，全局变量等）。之所以要捕获它们是因为这些变量代表了lambda创建时所对应的上下文信息，而lambda表达式执行的时候很可能要利用这些信息。所以，捕获这个词的使用是非常传神的。

函数参数、返回值类型以及函数体：这和普通函数的定义一样。不过，lambda表达式必须使用尾置形式的函数返回声明。尾置形式的函数返回声明即是把原来位于函数参数左侧的返回值类型放到函数参数的右侧。比如，"int func(int a){...} "的尾置声明形式为"autofunc(int a ) -> int {...}"。其中，auto是关键词，用在此处表明该函数将采用尾置形式的函数返回声明。

下面我们通过例子进一步来认识lambda表达式，来看下图：



lambda表达式实际上就是匿名函数对象，但是一般不知道它到底是什么类型，所以通过auto关键词把这个问题答案交给编译器来回答。

捕获列表可以按值和按引用两种方式来捕获信息。按引用方式进行捕获时需要考虑该变量生命周期的问题。因为lambda表达式作为一个对象是可以当做函数返回值跳出创建它的函数的范围。如果它通过引用方式捕获了一个函数内部的局部变量时，这个变量在跳出函数范围后将变得毫无意义，并且其占据的内存都可能不复存在了。

图所示例子的捕获列表显示指定了要捕获的变量。如果变量比较多的话，要一个一个写上变量名会变得很麻烦，所以lambda表达式还有更简单的方法来捕获所有变量，如下所示：

此处仅关注捕获列表中的内容

[=,&变量a,&变量b] = 号表示按值的方式捕获该lambda创建时所能看到的全部变量。如果有些变量需要通过引用方式来捕获的话就把它们单独列出来（变量前带上&符号）

[&,变量a,变量b] &号表示按引用方式捕获该lambda创建时所能看到的全部变量。如果有些变量需要通过按值方式来捕获的话就把它们单独列出来（变量前不用带上=号）

## STL 介绍

STL是StandardTemplate Library的缩写，英文原意是标准模板库。由于STL把自己的类和函数等都定义在一个名为std（std即standard之意）的命名空间里，所以一般也称其为标准库。标准库的重要意义在于它提供了一套代码实现非常高效，内容涵盖许多基础功能的类和函数，比如字符串类，容器类，输入输出类，多线程并发类，常用算法函数等。虽然和Java比起来，C++标准库涵盖的功能并不算多，但是用法却非常灵活，学习起来有一定难度。

熟练掌握和使用C++标准库是一个合格C++程序员的重要标志。对于标准库，笔者感觉是越了解其内部的实现机制越能帮助程序员更好得使用它。

STL的内容非常多，本节仅从API使用的角度来介绍其中一些常用的类和函数，包括：

string类，和Java中的String类似。

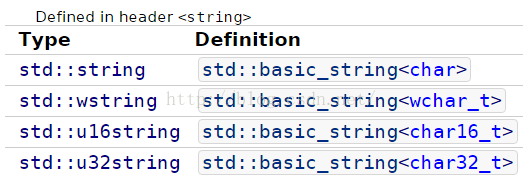
容器类，包括动态数组vector，链表list，map类、set类和对应的迭代器。

算法和函数，比如搜索，遍历算法，STL中的函数对象，绑定等。

智能指针类。

### 1.string类

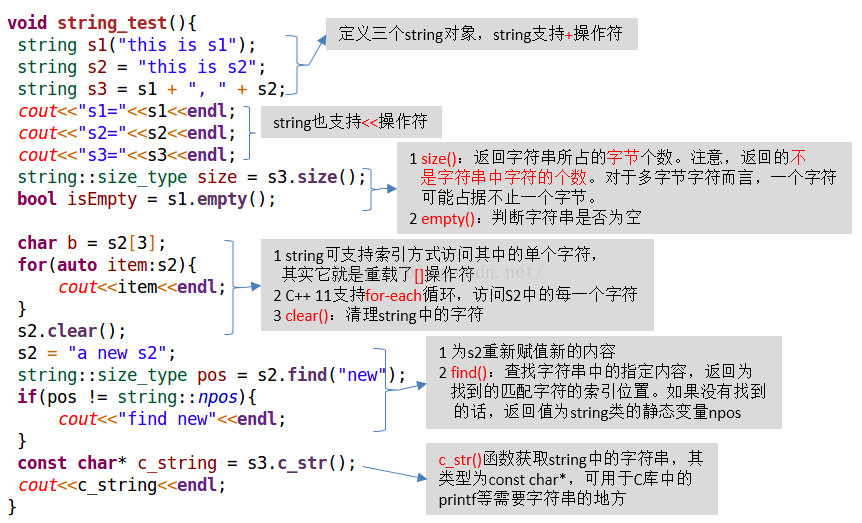
STL string类和Java String类很像。不过，STL的string类其实只是模板类basic\_string的一个实例化产物，STL为该模板类一共定义了四种实例化类，如图所示：



如果要使用其中任何一种类的话，需要包含头文件<string>。

* string对应的模板参数类型为char，也就是单字节字符。而如果要处理像UTF-8/UTF-16这样的多字节字符，程序员可酌情选用其他的实例化类。

string类的完整API可参考<http://www.cplusplus.com/reference/string/string/?kw=string>。其使用和Java String有些类似，所以上手难度并不大。下图中的代码展示了string类的使用：



### 2.容器类

好在Java中也有容器类，所以C++的容器类不会让大家感到陌生，表1对比了两种语言中常见的容器类。

|  |  |  |  |
| --- | --- | --- | --- |
| 容器类型 | STL类名 | Java类 | 说明 |
| 动态数组 | vector | ArrayList | 动态大小的数组，随机访问速度快 |
| 链表 | list | LinkedList | 一般实现为双向链表 |
| 集合 | set，multiset | SortedSet | 有序集合，一般用红黑树来实现。set中没有值相同的多个元素，而multiset允许存储值相同的多个元素 |
| 映射表 | map、multimap | SortedMap | 按Key排序，一般用红黑树来实现。map中不允许有Key相同的多个元素，而multimap允许存储Key相同的多个元素 |
| 哈希表 | unordered\_map | HashedMap | 映射表中的一种，对Key不排序 |

本节主要介绍表1中vector、map这两种容器类的用法以及Allocator的知识。

提示：

list、set和unordered\_map的在线API查询链接：

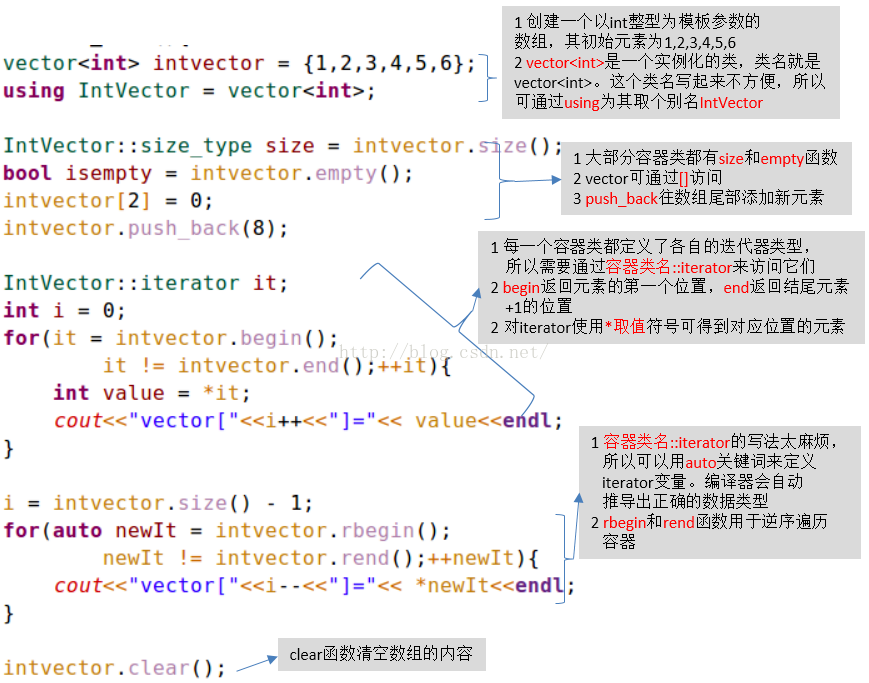
list的API：http://en.cppreference.com/w/cpp/container/list

set的API：http://en.cppreference.com/w/cpp/container/set

unordered\_map的API：http://en.cppreference.com/w/cpp/container/unordered\_map

#### vector类

vector是模板类，使用它之前需要包含<vector>头文件。



图中有三个知识点需要读者注意：

vector是模板类，本例用int作为模板参数实例化后得到一个名为vector<int>的类。这个类的名字写起来比较麻烦，所以我们通过using关键词为它定义了一个类型别名IntVector。IntVector是vector<int>的别名，凡是出现IntVector的地方其实都是vector<int>。

大部分STL容器类中都定义了相对应的迭代器，其类型名为Iterator。C++中没有通用的Iterator类（Java有Iterator接口类），而是需要通过容器类::Iterator的方式定义该容器类对应的迭代器变量。迭代器用于访问容器的元素，其作用和Java中的迭代器类似。

图中再次展示了auto的用法。auto关键词的出现使得程序员不用再写冗长的类型名了，一切交由编译器来完成。

#### Map类

map也叫关联数组。下图展示了map类的情况：

 map是模板类，使用它之前需要包含<map>头文件。map模板类包含四个模板参数，第一个模板参数Key代表键值对中键，第二个模板参数T代表键值对中值的类型，第三个模板参数Compare，它用于比较Key大小的，因为map是一种按key进行排序的容器。第四个参数Allocator用于分配存储键值对的内存。STL中，键值对用pair类来描述。

使用map的时候离不开pair。pair定义在头文件<utility>中。pair也是模板类，有两个模板参数T1和T2。

讨论：Compare和Allocator

map类的声明中，Compare和Allocator虽然都是模板参数，但很明显不能随便给它们设置数据类型，比如Compare和Allocator都取int类型可以吗？当然不行。实际上，Compare应该被设置成这样一种类型，这个类型的变量是一个函数对象，该对象被执行时将比较两个Key的大小。map为Compare设置的默认类型为std::less<Key>。less将按以小到大顺序对Key进行排序。除了std::less外，还有std::greater，std::less\_equal，std::greater\_equal等。

同理，Allocator模板参数也不能随便设置成一种类型。后文将继续介绍Allocator。

map类的用法：

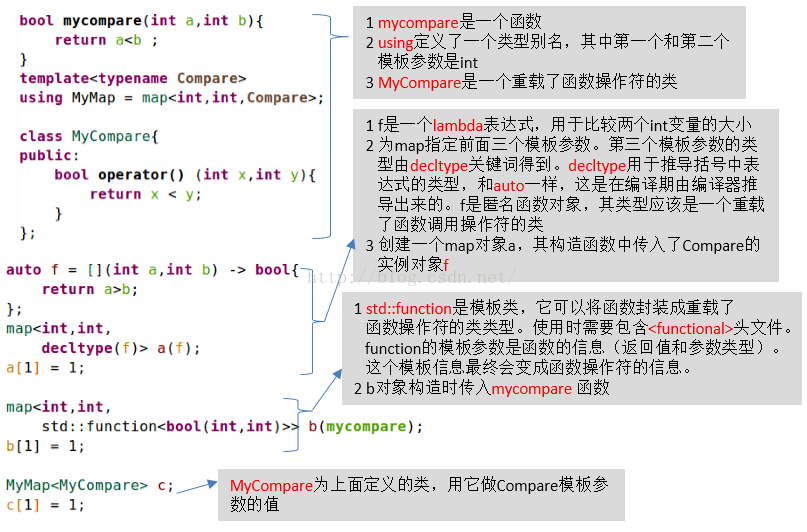


定义了一个key和value类型都是string的map对象，有两种方法为map添加元素：

通过索引Key的方式可添加或访问元素。比如stringMap["4"]="four"，如果stringMap["4"]所在的元素已经存在，则是对它重新设置新的值，否则是添加一个新的键值对元素。该元素的键为"4"，值为"four"。

通过insert添加一个元素。再次强调，map中元素的类型是pair，所以必须构造一个pair对象传递给insert。C++11前可利用辅助函数make\_pair来构造一个pair对象，C++11之后可以利用{}花括号来隐式构造一个pair对象了。

map默认的Compare模板参数是std::less，它将按从小到大对key进行排序，如何为map指定其他的比较方式呢？



图中展示了map中和Compare模板参数有关的用法，其中：

decltype(表达式)：用于推导表达式的数据类型。比如decltype(5)得到的是int，decltype(true)得到的是bool。decltype和auto都是C++11中的关键词，它们的真实类型在编译期间由编译器推导得到。

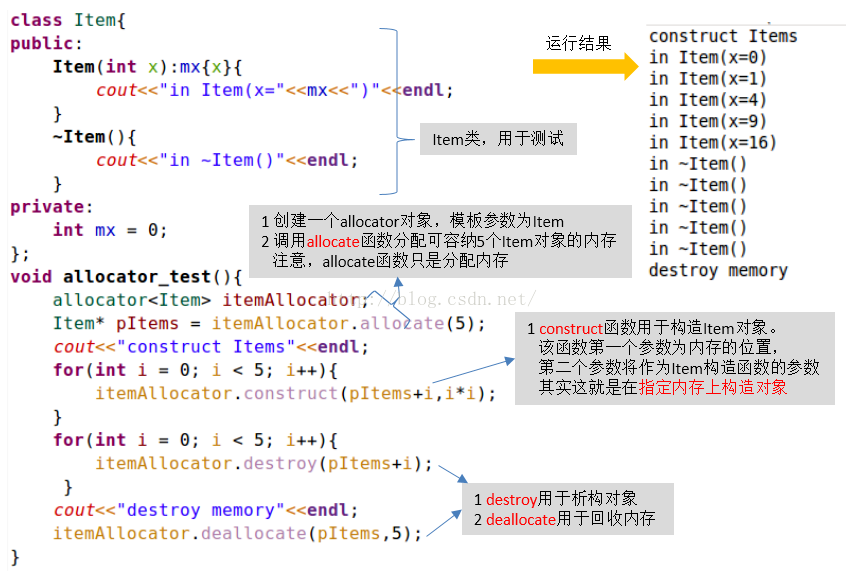
std::function是一个模板类，它可以将一个函数（或lambda表达式）封装成一个重载了函数操作符的类。这个类的函数操作符的信息（也就是函数返回值和参数的信息）和function的模板信息一样。比如图中"function<bool (int,int)>"将得到一个类，该类重载的函数操作符为"booloperator() (int,int)"。

#### Allocator介绍

Java程序员在使用容器类的时候从来不会考虑容器内的元素的内存分配问题。因为Java中，所有元素（除int等基本类型外）都是new出来的，容器内部无非是保存一个类似指针这样的变量，这个变量指向了真实的元素位置。

这个问题在C++中的容器类就没有这么简单了。比如，我们在栈上构造一个string对象，然后把它加到一个vector中去。vector内部是保存这个string变量的地址，还是在内部构造一个新的存储区域，然后将string对象的内容保存起来呢？显然，我们应该选择在内部构造一个区域，这个区域存储string对象的内容。

STL所有容器类的模板参数中都有一个Allocator（译为分配器），它的作用包括分配内存、构造对应的对象，析构对象以及释放内存。STL为容器类提供了一个默认的类，即std::allocator。其用法如图所示：



上图展示了allocator模板类的用法，我们可以为容器类指定自己的分配器，它只要定义图中的allocate、construct、destory和deallocate函数即可。当然，自定义的分配器要设计好如何处理内存分配、释放等问题也是一件很考验程序员功力的事情。

提示：

ART中也定义了类似的分配器，以后我们会碰到它们。

### 3.算法和函数对象介绍

STL还为C++程序员提供了诸如搜索、排序、拷贝、最大值、最小值等算法操作函数以及一些诸如less、great这样的函数对象。本节先介绍算法操作函数，然后介绍STL中的函数对象。

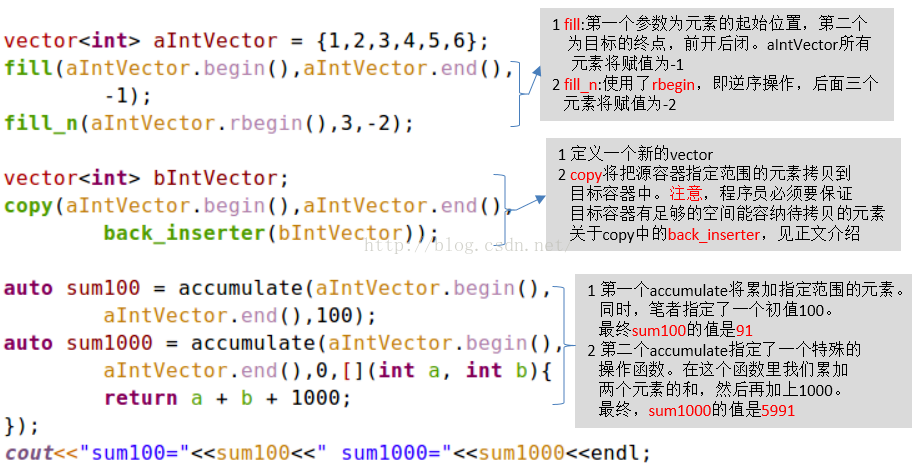
#### 算法

STL中要使用算法相关的API的话需要包含头文件<algorithm>，如果要使用一些专门的数值处理函数的话则需额外包含<numeric>头文件。参考文献[2]在第11章中对STL算法函数进行了细致的分类。不过本节不打算从这个角度、大而全得介绍它们，而是将ART中常用的算法函数挑选出来介绍，如表所示。

ART源码中常用的算法函数

|  |  |
| --- | --- |
| 函数名 | 作用 |
| fill  fill\_n | fill：为容器中指定范围的元素赋值  fill\_n：为容器内指定的n个元素赋值 |
| min/max | 返回容器某范围内的最小值或最大值 |
| copy | 拷贝容器指定范围的元素到另外一个容器 |
| accumulate | 定义于<numerics>，计算指定范围内元素之和 |
| sort | 对容器类的元素进行排序 |
| binary\_search | 对已排序的容器进行二分查找 |
| lexicographical\_compare | 按字典序对两个容器内内指定范围的元素进行比较 |
| equal | 判断两个容器是否相同（元素个数是否相等，元素内容是否相同） |
| remove\_if | 从容器中删除满足条件的元素 |
| count | 统计容器类满足条件的元素的个数 |
| replace | 替换容器类旧元素的值为指定的新元素 |
| swap | 交换两个元素的内容 |

一些函数的用法：



包含一些知识点需要读者了解：

对于操作容器的算法函数而言，它并不会直接操作具体的容器类，而是借助Iterator来遍历一个范围（一般是前开后闭）内的元素。这种方式将算法和容器进行了最大程度的解耦，从此，算法无需关心容器，而是只通过迭代器来获取、操作元素。

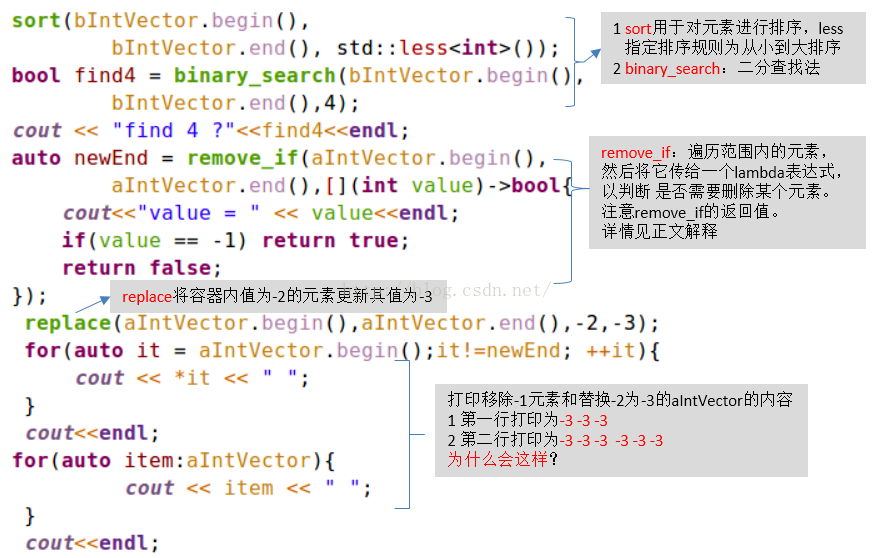
对初学者而言，算法函数并不像它的名字一样看起来那么容易使用。以copy为例，它将源容器指定范围元素拷贝到目标容器中去。不过，目标容器必须要保证有足够的空间能够容纳待拷贝的源元素。比如图中aIntVector有6个元素，但是bIntVector只有0个元素，aIntVector这6个元素能拷贝到bIntVector里吗？copy函数不能回答这个问题，只能由程序员来保证目标容器有足够的空间。这导致程序员使用copy的时候就很头疼了。为此，STL提供了一些辅助性的迭代器封装类，比如back\_inserter函数将返回这样一种迭代器，它会往容器尾部添加元素以自动扩充容器的大小。如此，使用copy的时候我们就不用担心目标容器容量不够的问题了。

有些算法函数很灵活，它可以让程序员指定一些判断、操作规则。比如第二个accumulate函数的最后一个参数，我们为其指定了一个lambda表达式用于计算两个元素之和。

提示：

STL的迭代器也是非常重要的知识点。

接着来看下图，它继续展示了算法函数的使用方法：

图中remove\_if函数向读者生动展示了要了解STL细节的重要性：

remove\_if将vector中值为-1的元素remove。但是这个元素会被remove到哪去？该元素所占的内存会不会被释放？STL中，remove\_if函数只是将符合remove条件的元素挪到容器的后面去，而将不符合条件的元素往前挪。所以，vector最终的元素布局为前面是无需移动的元素，后面是被remove的元素。但是请注意，vector的元素个数并不会发生改变。所以，remove\_if将返回一个迭代器位置，这个迭代器的位置指向被移动的元素的起始位置。即vector中真正有效的元素存储在begin()和newEnd之间，newEnd和end()之间是逻辑上被remove的元素。

如果初学者不知道remove\_if并不会改变vector元素个数的话，就会出现图中最后一个for循环的结果，vector的元素还是有6个，就好像没有被remove一样。

是不是有种要抓狂的感觉？这个问题怎么破解呢？当使用者remove\_if调用完毕后，务必要通过erase来移除容器中逻辑上不再需要的元素，代码如下：

//newEnd和end()之间是逻辑上被remove的元素，我们需要把它从容器里真正移除!

aIntVector.erase(newEnd,aIntVector.end());

最后，关于<algorithm>的全部内容请读者参考：

http://en.cppreference.com/w/cpp/header/algorithm

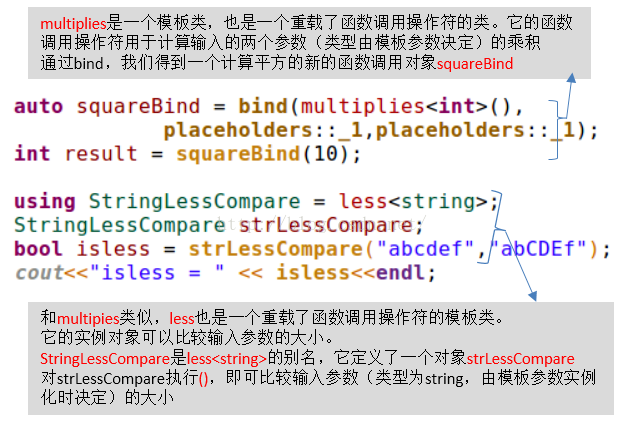
#### 函数介绍

STL中要使用函数对象相关的API的话需要包含头文件<functional>，ART中常用的函数对象如表所示。

|  |  |
| --- | --- |
| 类或函数名 | 作用 |
| bind | 对可调用对象进行参数绑定以得到一个新的可调用对象。详情见正文 |
| function | 模板类，用于得到一个重载了函数调用对象的类 |
| hash | 模板类，用于计算哈希值 |
| plus/minus/multiplies | 模板类，用于计算两个变量的和，差和乘积 |
| equal\_to/greater/less | 模板类，用于比较两个数是否相等或大小 |

函数对象的使用相对比较简单，下图中给出了几个示例：

重点介绍了bind函数的用法。如图中所说，bind是一个很奇特的函数，其主要作用就是对原可调用对象进行参数绑定从而得到一个新的可调用对象。bind的参数绑定规则需要了解。另外，占位符\_X定义在std下的placeholders命名空间中，所以一般要用placeholders::\_X来访问占位符。

 mutiplies模板类：它是一个重载了函数操作符的模板类，用于计算两个输入参数的乘积。输入参数的类型就是模板参数的类型。

less模板类，和mutiplies类似，它用于比较两个输入参数的大小。

最后，关于<algorithm>的全部内容，请读者参考：

http://en.cppreference.com/w/cpp/header/functional

提示：

从容器类和算法以及函数对象来看，STL的全称标准模板库是非常名符其实的，它充分利用了和发挥了模板的威力。

### 4.智能指针类

C++11此次在STL中推出了两个比较常用的智能指针类：

shared\_ptr：共享式指针管理类。内部有一个引用计数，每当有新的shared\_ptr对象指向同一个被管理的内存资源时，其引用计数会递增。该内存资源直到引用计数变成0时才会被释放。

unique\_ptr：独占式指针管理类。被保护的内存资源只能赋给一个unique\_ptr对象。当unique\_ptr对象销毁、重置时，该内存资源被释放。一个unique\_ptr源对象赋值给一个unique\_ptr目标对象时，内存资源的管理从源对象转移到目标对象。

shared\_ptr和unqiue\_ptr的思想其实都很简单，就是借助引用计数的概念来控制内存资源的生命周期。相比shared\_ptr的共享式指针管理，unique\_ptr的引用计数最多只能为1罢了。

注意：环式引用问题

虽然有shared\_ptr和unique\_ptr，但是C++的智能指针依然不能做到Java那样的内存自动回收。并且，shared\_ptr的使用也必须非常小心，因为单纯的借助引用计数无法解决环式引用的问题，即A指向B，B指向A，但是没有别的其他对象指向A和B。这时，由于引用计数不为0，A和B都不能被释放。

下面分别来看shared\_ptr和unique\_ptr的用法。

#### shared\_ptr的介绍

下图为shared\_ptr的用法示例，难度并不大：



STL提供一个帮助函数make\_shared来构造被保护的内存对象以及一个的shared\_ptr对象。

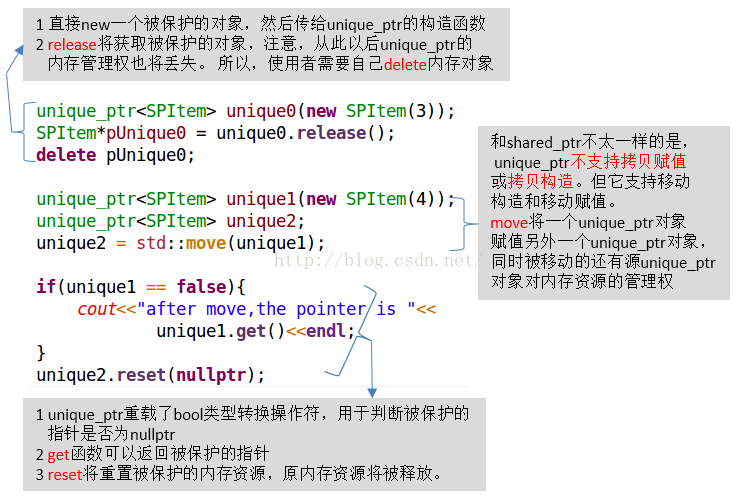
当item0赋值给item1时，引用计数（通过use\_count函数返回）递增。

reset函数可以递减原被保护对象的引用计数，并重新设置新的被保护对象。

关于shared\_ptr更多的信息，请参考：http://en.cppreference.com/w/cpp/memory/shared\_ptr

#### unique\_ptr的介绍

ART中使用unique\_ptr远比shared\_ptr多，它的用法比shared\_ptr更简单，如图所示：

关于unique\_ptr完整的API列表，请参考<http://en.cppreference.com/w/cpp/memory/unique_ptr>

### 5.探讨STL的学习

本章对STL进行了一些非常粗浅的介绍。结合笔者个人的学习和使用经验，STL初看起来是比较容易学的。因为它更多关注的是如何使用STL定义好的类或者函数。从“使用现成的API”这个角度来看，有Java经验的读者应该毫不陌生。因为Java平台从诞生之初就提供了大量的功能类，熟练的java程序员使用它们时早已能做到信手拈来。同理，C++程序员初学STL时，最开始只要做到会查阅API文档，了解API的用法即可。

但是，正如前面介绍copy、remove\_if函数时提到的那样，STL的使用远比掌握API的用法要复杂得多。STL如果要真正学好、用好，了解其内部大概的实现是非常重要的。并且，这个重要性不仅停留在“可以写出更高效的代码”这个层面上，它更可能涉及到“避免程序出错，内存崩溃等各种莫名其妙的问题”上。另外，C++之父编写的第IV部分也对STL进行了大量深入的介绍，读者也可以仔细阅读。

要研究STL的源码吗？

对绝大部分开发者而言，笔者觉得研究STL的源码必要性不大。http://en.cppreference.com网站中会给出有些API的可能实现，读者查找API时不妨了解下它们。

## 其他常用知识

本节介绍ART代码中其他一些常见知识。

### initializer\_list

initializer\_list和C++11中的一种名为“列表初始化”的技术有关。什么是列表初始化呢？来看一段代码：

vector<int>intvec = {1,2,3,4,5};

vector<string>strvec{”one”,”two”,”three”};”

上面代码中，intvect和strvect的初值由两个花括号{}和里边的元素来指定。C++11中，花括号和其中的内容就构成一个列表对象，其类型是initializer\_list，也属于STL标准库。

initializer\_list是一个模板类，花括号中的元素的类型就是模板类型。并且，列表中的元素的数据类型必须相同。

另外，如果类创建的对象实例构造时想支持列表方式的话，需要单独定义一个构造函数。我们来看几段代码：

class Test{

public:

//①定义一个参数为initializer\_list的构造函数

Test(initializer\_list<int> a\_list){

//②遍历initializer\_list，它也是一种容器

for(auto item:a\_list){

cout<<”item=”<<item<<endl;

}

}

}

Test a = {1,2,3,4};//只有Test类定义了①，才能使用列表初始化构造对象

initializer\_list<string> strlist ={”1”,”2”,”3”};

using ILIter =initializer\_list<string>::iterator;

//③通过iterator遍历initializer\_list

for(ILIter iter =strlist.begin();iter != strlist.end();++iter){

cout<<”item = ” << \*iter<< endl;

}

### 带作用域的enum

enum应该是广大程序员的老相识了，它是一个非常古老，使用广泛的关键词。不过，C++11中enum有了新的变化，我们通过两段代码来了解它：

//C++11之前的传统enum，C++11继续支持

enum Color{red,yellow,green};

//C++11之后，enum有一个新的形式：enum class或者enum struct

enum class ColorWithScope{red,yellow,green}

由上述代码可知，C++11为古老的enum添加了一种新的形式，叫enum class（或enum struct）。enum class和Java中的enum类似，它是有作用域的，比如：

//对传统enum而言：

int a\_red = red;//传统enum定义的color仅仅是把一组整型值放在一起罢了

//对enum class而言，必须按下面的方式定义和使用枚举变量。

//注意，green是属于ColorWithScope范围内的

ColorWithScopea\_green = ColorWithScope::green;//::是作用域符号

//还可以定义另外一个NewColor，这里的green则是属于AnotherColorWithScope范围内

enum class AnotherColorWithScope{green,red,yellow};

//同样的做法对传统enum就不行，比如下面的enum定义将导致编译错误，

//因为green等已经在enum Color中定义过了

enum AnotherColor{green,red,yellow};

### constexpr

const一般翻译为常量，它和Java中的final含义一样，表示该变量定义后不能被修改。但C++11在const之外又提出了一个新的关键词constexpr，它是constexpression（常量表达式）的意思。constexpr有什么用呢？很简单，就是定义一个常量。

读者一定会觉得奇怪，const不就是用于定义常量的吗，为什么要再来一个constexpr呢？关于这个问题的答案，让我们通过例子来回答。先看下面两行代码：

const int x = 0;//定义一个整型常量x，值为0

constexpr int y =1; //定义一个整型常量y，值为1

上面代码中，x和y都是整型常量，但是这种常量的初值是由字面常量（0和1就是字面常量）直接指定的。这种情况下，const和constexpr没有什么区别（注意，const和constexpr的变量在指向指针或引用型变量时，二者还是有差别，此处不表）。

不过，对于下面一段代码，二者的区别立即显现了：

int expr(int x){//测试函数

if(x == 1) return 0;

if(x == 2) return 1;

return -1;

}

const int x = expr(9);

x = 8;//编译错误，不能对只读变量进行修改

constexpr int y = expr(1);//编译错误，因为expr函数不是常量表达式

上面代码中：

x定义为一个const整型变量，但因为expr函数会根据输入参数的不同而返回不同的值，所以x其实只是一个不能被修改的量，而不是严格意义上的常量。常量的含义不仅仅是它的值不能被改变，并且它的值必须是固定的。

对于这种情况，我们可以使用constexpr来定义一个货真价实的常量。constexpr将告知编译器对expr函数进行推导，判断它到底是不是一个常量表达式。很显然，编译器判断expr不是常量表达式，因为它的返回值受输入参数的影响。所以上述y变量定义的那行代码将无法通过编译。

所以，constexpr关键词定义的变量一定是一个常量。如果等号右边的表达式不是常量，那么编译器会报错。

提示：

常量表达式的推导工作是在编译期决定的。

### static\_assert

assert，也叫断言。程序员一般在代码中一些关键地方加上assert语句用以检查参数等信息是否满足一定的要求。如果要求达不到，程序会输出一些警告语（或者直接异常退出）。总之，assert是一种程序运行时做检查的方法。

有没有一种方法可以让程序员在代码的编译期也能做一些检查呢？为此，C++11推出了static\_assert，它的语法如下：

static\_assert (bool\_constexpr , message )

当bool\_constexpr返回为false的时候，编译器将报错，报错的内容就是message。注意，这都是在编译期间做的检查。

读者可能会好奇，什么场合需要做编译期检查呢？举个最简单的例子。假设我们编写了一段代码，并且希望它只能在32位的机器上才能编译。这时就可以利用static\_assert了，方法如下：

static\_assert(sizeof(void\*) == 4,”can only be compiled in32bit machine”);

包含上述语句的源码文件在64位机器上进行编译将出错，因为64位机器上指针的字节数是8，而不是4。

### Thread

thread类当中的两个成员函数，join()和detach()。这两个成员的作用就像上面代码的注释那样，启动新生成的线程的，但是区别在于join()函数是启动子线程而阻塞主线程，当子线程运行结束后，才会继续运行主线程。相比之下，detach()函数的作用是启动子线程，并且让子线程和主线程分离，子线程和主线程各运行各的，虽然两个线程会因为共享内存池的原因在操作系统的层面发生发生阻塞等关系，但是在代码层次上，两个线程并不存在谁阻塞谁，很可能主线程已经运行结束了，子线程还在运行。

#### **新建多线程工程**

1. cmake文件 加入编译选项c11和多线程

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -Wall -std=c++11 -pthread -g -march=native")

很重要！！不然会爆出错误

在函数‘std::thread::thread<void (\*)()>(void (\*&&)())’中：

/usr/include/c++/5/thread:137：对‘pthread\_create’未定义的引用

1. 包括头文件#include<thread>
2. 新建线程  
   使用thread 线程名来新建线程

#include <iostream>

#include<thread>

#include<unistd.h>

using namespace std;

void sayHello() {

while (1) {

sleep(1);

cout << endl << "hello" << endl;

}

}

void sayWorld() {

while (1) {

sleep(1);

cout << endl << "world" << endl;

}

}

int main() {

thread threadHello(&sayHello);

thread threadWorld(&sayWorld);

threadHello.join();

threadWorld.join();

return 0;

}

代码解析：

* c++在新建一个线程thread threadHello(&sayHello);，在新建的过程会自动调用线程内容，所以新建的位置需要注意。
* threadHello.join();的意思是外在的线程（一般为主线程）等待\*this线程（这里就是threadHello线程）的完成，阻塞外在的线程。
* thread其他方法.detach()，调用之后，目标线程就成为了守护线程，驻留后台运行，与之关联的std::thread对象失去对目标线程的关联，无法再通过std::thread对象取得该线程的控制权。当线程主函数执行完之后，线程就结束了，运行时库负责清理与该线程相关的资源。  
  当一个thread对象到达生命期终点而关联线程还没有结束时，则thread对象取消与线程之间的关联，目标线程线程则变为分离线程继续运行。  
  注意：在主线程未执行完前并没有销毁，仍会继续执行。
* 另：若子线程不写join阻塞主线程或detach后台运行，不管子线程是否被执行完毕。都会爆出warning:terminate called without an active exception  
  这是指就是子线程还在运行，主进程就退出导致了该错误。即使你认为子线程已经结束，但系统没有通过join之类的阻塞解除之类的方法判断，都会爆出这个warning。

#### 线程死锁方法(unique\_lock的使用)

1. 定义：  
   unique\_lock中的unique表示独占所有权。  
   unique\_lock独占的是mutex对象，就是对mutex锁的独占。
2. 用法：  
   （1）新建一个unique\_lock 对象  
   （2）给对象传入一个std::mutex 对象作为参数;  
   std::mutex mymutex;  
   unique\_lock lock(mymutex);
3. 解锁  
   因此加锁时新建一个对象lockunique\_lock lock(mymutex);，而这个对象生命周期结束后自动解锁。
4. 参考代码

#include <iostream>

#include<thread>

#include<unistd.h>

#include<mutex>

using namespace std;

std::mutex mymutex;

void sayHello() {

int k = 0;

unique\_lock<mutex> lock(mymutex);

while (k < 2) {

k++;

cout << endl << "hello" << endl;

sleep(2);

}

}

void sayWorld() {

unique\_lock<mutex> lock(mymutex);

while (1) {

cout << endl << "world" << endl;

sleep(1);

}

}

int main() {

thread threadHello(&sayHello);

thread threadWorld(&sayWorld);

threadHello.join();

threadWorld.join();

return 0;

}

程序运行步骤是这样的：  
首先同时运行threadHello线程和threadWorld线程  
先进入threadHello线程的sayHello()函数，这个时候加了mymutex锁，另外一个threadWorld线程进入后发现mymutex锁没有释放，只能等待。  
当过去两个循环（每个循环2秒后）threadHello线程结束，unique\_lock lock(mymutex)的生命周期结束，mymutex锁释放，执行threadWorld线程，此时开始一直say world。