# 内存模型

# 数据类型

## 内建类型

### 算术类型

算术类型包括基本算术类型和void特殊类型。



### 有符号/无符号类型

有三种不同的char类型：没有修饰的（plain）char、signed char和unsigned char。

注：plain char具体时间unsigned还是signed是由编译器决定的，C++标准没有规定。



确定要使用的整数类型和浮点类型：

1. 确定数值不能为负时，使用无符号类型
2. 整数计算时使用int
3. 不要再算术表达式中使用char或者bool，只可以用它们保留字符和真值
4. 在浮点计算中使用double

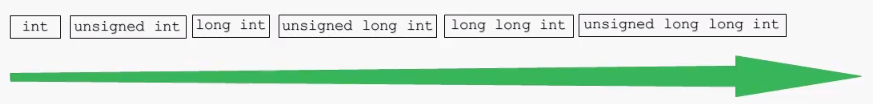
### 类型转换

### 字面量

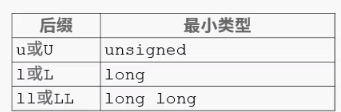
十进制整数字面量类型：



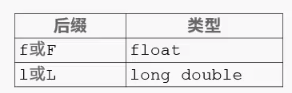
八进制和十六进制整数字面量的类型：



整数字面量后缀：



浮点数字面量后缀：



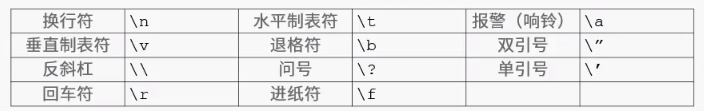
注：double是浮点数默认的类型。

字符和字符串字面量前缀：



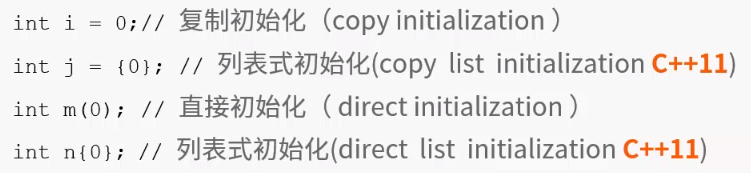
注：目前STL只提供了wchar\_t的输入和输出。

转义字符：



## 变量

变量的初始化：



## 复合类型

指针和引用

# 类型别名

## typedef

## using

别名声明符：using

# 类型推断

## auto

## decltype

# 类型转换

## static\_cast

### 实现

### 使用

template<typename T, typename U>

shared\_ptr<T> static\_pointer\_cast(constshared\_ptr<U> &other) noexcept {

T \*ptr = static\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

// shape\* -> circle\* 使用static\_cast转换后,指针为空 与dynamic\_cast相比,不安全

shared\_ptr<shape> sptr2(new shape);

shared\_ptr<circle> sptr1 = static\_pointer\_cast<circle>(sptr2); // 基类转子类

cout << "use count of sptr1 is now " << dptr1.use\_count() << endl; // 0

cout << "use count of sptr2 is now " << dptr2.use\_count() << endl; // 1

// circle\* -> circle\* 使用dynamic\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2

shared\_ptr<shape> sptr3(new circle);

//shared\_ptr<circle> sptr3(new circle); // 上面或者当前行,后面输出一样！

shared\_ptr<circle> sptr1\_1 = static\_pointer\_cast<circle>(sptr3); // 基类转子类

cout << "use count of sptr1\_1 is now " << sptr1\_1.use\_count() << endl; // 2

cout << "use count of sptr3 is now " << sptr3.use\_count() << endl; // 2

// circle\* -> circle\* 使用static\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2 等价于dynamic\_cast

shared\_ptr<circle> sptr3\_1(new circle);

shared\_ptr<shape> sptr2\_1 = static\_pointer\_cast<shape>(sptr3\_1); //子类转基类，上行转换，安全！

cout << "use count of sptr2\_1 is now " << sptr2\_1.use\_count() << endl; // 2

cout << "use count of sptr3\_1 is now " << sptr3\_1.use\_count() << endl; // 2

输出结果同上dynamic\_cast，不同之处，在下行转换的时候(基类转子类)，是不安全的！

适用场景：

1. 内置数据类型转换；
2. 父子类转换：

**向下类型转换（父类->子类）不安全，向上类型转换（子类->父类）安全**

没有父子关系的两个类之间是无法转换成功的

## dynamic\_cast

### 实现

### 使用

在unique\_ptr实现了子类向基类的转换，但是却没有实现基类向子类的转换，例如：:unique\_ptr<circle>转unique\_ptr<shape>。

实现这种，需要使用dynamic\_cast，实现如下：

首先为了实现这些转换，我们需要添加构造函数，允许在对智能指针内部的指针对象赋值时，使用一个现有的智能指针的共享计数。

// 实现强制类型转换需要的构造函数

template<typename U>

shared\_ptr(constshared\_ptr<U> &other, T \*ptr) noexcept {

ptr\_ = ptr;

if (ptr\_) {

other.shared\_count\_->add\_count();

shared\_count\_ = other.shared\_count\_;

}

}

其次，就是实现转换函数：

template<typename T, typename U>

shared\_ptr<T> dynamic\_pointer\_cast(constshared\_ptr<U> &other) noexcept {

T \*ptr = dynamic\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

// shape\* -> circle\* 使用dynamic\_cast转换后,指针为空.此时资源还是被dptr2拥有,dptr1为0

shared\_ptr<shape> dptr2(new shape);

shared\_ptr<circle> dptr1 = dynamic\_pointer\_cast<circle>(dptr2); // 基类转子类

cout << "use count of dptr1 is now " << dptr1.use\_count() << endl; // 0

cout << "use count of dptr2 is now " << dptr2.use\_count() << endl; // 1

// circle\* -> circle\* 使用dynamic\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2

shared\_ptr<shape> dptr3(new circle);

// shared\_ptr<circle> dptr3(new circle); // 上面或者当前行,后面输出一样！

shared\_ptr<circle> dptr1\_1 = dynamic\_pointer\_cast<circle>(dptr3); // 基类转子类

cout << "use count of dptr1\_1 is now " << dptr1\_1.use\_count() << endl; // 2

cout << "use count of dptr3 is now " << dptr3.use\_count() << endl; // 2

// circle\* -> circle\* 使用dynamic\_cast转换后,指针不为空,此时资源被两者共同使用,引用计数为2

shared\_ptr<circle> dptr3\_1(new circle);

shared\_ptr<shape> dptr2\_1 = dynamic\_pointer\_cast<shape>(dptr3\_1); // 子类转基类 上行转换,安全！

cout << "use count of dptr2\_1 is now " << dptr2\_1.use\_count() << endl; // 2

cout << "use count of dptr3\_1 is now " << dptr3\_1.use\_count() << endl; // 2

dynamic\_cast主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。在类层次间进行上行转换时，dynamic\_cast和static\_cast的效果是一样的；在进行下行转换时，dynamic\_cast具有类型检查的功能，比static\_cast更安全。在多态类型之间的转换主要使用dynamic\_cast，因为类型提供了运行时信息。

1、下行转换，基类转换为子类，例如：智能指针转换类似于shape\* 转换为circle\* 使用dynamic\_cast转换后,指针为空.此时资源还是被dptr2拥有,dptr1为0。比static\_cast安全。

2、平行转换，指向一致的相互转换，例如：智能指针转换类似于circle\*转换为circle\*。此时引用计数为两者共享。

3、上行转换，子类转基类，例如：智能指针转换类似于circle\*转换为shape\*，此时引用技术为两者共享。等价于static\_cast。

适用场景：

1. 内置数据类型不适用（有精度损失或不安全转换不允许）；
2. 如果发生多态，父子之间转换总是安全的。

## const\_cast

### 实现

### 使用

去掉const属性：

template<typename T, typename U>

shared\_ptr<T> const\_pointer\_cast(

constshared\_ptr<U> &other) noexcept {

T \*ptr = const\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

shared\_ptr<circle> s = const\_pointer\_cast<circle>(shared\_ptr<const circle>(new circle));

适用场景：

1. 指针或引用转换；
2. 不可以对非指针或非引用做const\_cast转换。

举例：

int a = 10;

const int& b = a;

int &c = const\_cast<int&>(b);

c = 20;

或：

const int\* p = NULL;

int\* p2 = const\_cast<int\*>(p);

int\* p3 = NULL;

const int\* p4 = const\_cast<const int\*>(p3);

## reinterpret\_cast

### 实现

### 使用

例如：想把一个指针转为整数，就可以用reinterpret\_cast。

template<typename T, typename U>

shared\_ptr<T> reinterpret\_pointer\_cast(

constshared\_ptr<U> &other) noexcept {

T \*ptr = reinterpret\_cast<T \*>(other.get());

returnshared\_ptr<T>(other, ptr);

}

调用：

int a = reinterpret\_pointer\_cast<int>(s);

重新解释转换，最不安全，不建议使用（没有关系的类之间使用）。

# 标准输入输出

## cin

### cin.get

### cin.getline

### cin.ignore

### cin.peek

### cin.put\_back

## cout

### cout.put

### cout.write

### cout.flush

### cout.width

### cout.fill

### cout.self

### cout.unself

## cerr

## clog

## iostream

### istream

### ostream

### fstream

### ifstream

### ofstream

# 内存分配

## new/delete/malloc/free

delete会调用对象的析构函数，和new对应free只会释放内存，new调用构造函数。

malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。

对于非内部数据类型的对象而言，**光用malloc/free无法满足动态对象的要求**。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。

注意：**new/delete不是库函数**。

## delete与delete []

delete只会调用一次析构函数，而delete[]会调用每一个成员的析构函数。

在More Effective  C++中有更为详细的解释：“当delete操作符用于数组时，它为每个数组元素调用析构函数，然后调用operatordelete来释放内存。”delete与New配套，delete []与new []配套：

  MemTest \*mTest1 = new MemTest[10];

  MemTest \*mTest2 = new MemTest;

  int \*pInt1 = new int[10];

  int \*pInt2 = new int;

  delete []pInt1;  //-1-

  delete []pInt2;  //-2-

  delete []mTest1;//-3-

  delete []mTest2;//-4-

  在-4-处报错。

这就说明：对于内建简单数据类型，delete和delete[]功能是相同的。对于自定义的复杂数据类型，delete和delete[]不能互用。delete[]删除一个数组，delete删除一个指针。

简单来说，用new分配的内存用delete删除用new[]分配的内存用delete[]删除delete[]会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大。如果你在用delete时没用括号，delete就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

# 指针与引用

## 引用

### 普通引用

在分析之前我们先介绍一下引用。引用为对象起了一个别名。例如：

int a = 1024；

int &b = a;

这里的b就是指向a，相当于a的另外一个名字。b是一个引用，但并非一个对象，同时引用的类型都要和对象的类型严格匹配（在具有继承关系的类中是例外，这里不展开），因此有下面的操作：

int a = 1024；

int &b = a;

int &c = b; //非法，b在此之前已经是引用，不能定义引用的引用

int &d = 1024; //非法，引用类型的初始值必须是对象，而不能是字面值

const int &e = 1024 //合法，e为常量引用，只要能够转换成引用的类型，可以使用任何表达式作为初始值

double f = 1.024；

int &g = f; //非法，类型不匹配

而对引用赋值，实际上是将值赋给了与引用绑定的对象，例如：

int h = 1024；

int &i = h;

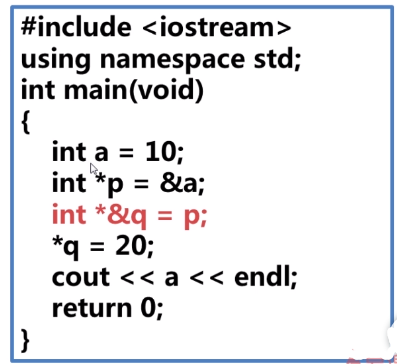
i = 256;//将256赋给了h

同时，引用必须被初始化，下面的操作是非法的：

int &j;//非法，没有初始化

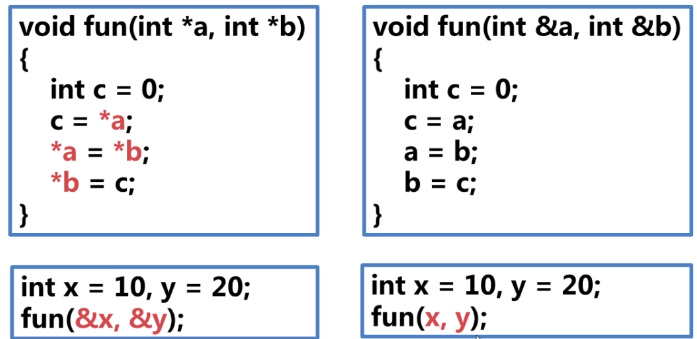
### 指针类型的引用

格式：类型 \*&指针引用名 = 指针;



注：指针引用就是表示指向的是指针类型的引用，其实与普通数据的引用没有差别。比如int\* &q表示的是int\*类型的引用q。此时输出的是a=20

### 引用作为函数参数



### const与引用

int x = 3;

const int &y = x;

//x=10;正确

//y=20;错误

## 指针

学习过C的朋友可能对指针已经比较熟悉了，这里不占用较大篇幅。它存储的是一个地址，地址指向的是一个对象。例如：

int a = 0; //定义变量a

int \*ptr = &a;定义int类型指针ptr，它存放变量a的地址

与引用类似，指针也实现了对其他对象的间接访问。同样的，它也要求指针的类型和指向的类型严格匹配。

## 比较

**引用不可为空，而指针可以**

例如：

int &a; //非法，必须被初始化

int \*p;//合法，但如果是非静态的指针变量，将拥有一个不确定的值

声明一个指针变量而不初始化是合法的，但是不建议这么做，原因可参见C语言入坑指南-被遗忘的初始化。正因如此，使用引用而非指针作为函数的参数是一个不错的选择，因为引用永远不为空，函数入口也就不需要做过多的检查，引用也就更富效率。

**引用不是对象，而指针本身是对象**

引用不是对象，它不能被再次赋值，而指针是可以的。例如：

int a = 1024;

int b = 2048;

int &c = a;

int \*d = &a;

c = b; //其引用不会被再次赋值

d = &b;//合法，指针可以指向另外一个对象

也就是说，一旦引用了引用，就没法让它绑定到其他的对象。看起来引用还挺从一而终啊。

另外，可以有指向指针的指针，而不存在引用的引用。因为引用不是对象。

int a = 1024;

int &b = a;

int &&c = b;//非法

int \*d = &a;

int \*\*e = &d;//合法

**引用指向对象本身，而指针指向的是对象的地址**

定义一个指针的时候，编译器为他分配内存，而引用不会单独分配空间。

引用所代表的就是最初绑定的那个对象，因此使用sizeof分别作用于引用和指针时，前者得到的是引用所绑定对象大小，而后者得到的是指针占用空间大小（4或8字节），例如在64位的程序中：

/\*\*假设有以下结构\*/

typedef struct INFO

{

int a;

int b;

char c;

}INFO;

INFO info = {0};//定义结构info

INFO &ref = info;//定义引用ref

INFO \*ptr = &info;//定义指针ptr

sizeof(ref); //大小为INFO结构体占用空间大小，即12字节

sizeof(ptr); //占用大小为指针占用空间大小，即8字节

**引用使用时无需解引用，而指针需要**

例如：

int a = 1024;

int &ref = a;

int \*p = &a;

ref = 10; //使用引用可直接使用，将所绑定对象的值修改

\*p = 11;//使用指针需要解引用

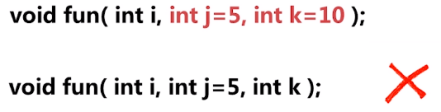
**作为参数时的差别**

指针作为参数时，看起来是地址传参，实际上仍然传值，即将指针的一个拷贝作为实参，而由于指针指向的是一个对象，因此在函数内可以实现对指针所指向对象的内容进行改变。而引用作为参数时，实际上传递的对象本身，但又不需要拷贝，因为引用绑定的就是对象。

# 函数

## 参数默认值

有默认参数值的参数必须在参数表的最右端：



无实参则用默认值，否则实参覆盖默认值。

## 内联函数



既然内联函数有这样的特点，为什么不是所有函数都用内联函数？

1. 内联编译时建议性的，由编译器决定；
2. 逻辑简单，调用频繁的函数建议使用内联；
3. 递归函数无法使用内联方式

# 异常

C语言处理异常的缺陷在于，返回的值没有统一（比如返回值-1一般表示错误，但是也可以表示其他情况），返回的值可以是异常的结果，也可以是正常的结果，C++中使用try-catch-throw。

栈解旋：从try代码块开始起，到throw抛出异常前，所有栈上的对象都被释放掉，释放的顺序和构造的顺序是相反的，这个过程称为栈解旋。

# 封装性

## 类和对象

C++将对象抽象成类进行处理，由于目的的不同抽象出的信息也可能不同。

根据目的不同，决定哪些信息需要暴露出去，哪些信息需要隐藏起来，这些通过访问限定符（public、protected、private）。如果需要暴露信息使用public访问控制符，如果需要隐藏信息则使用private访问控制符。

## 对象实例化

类就是一个模板，所谓的对象实例化就是根据一个类的设定，制造多个对象的过程。

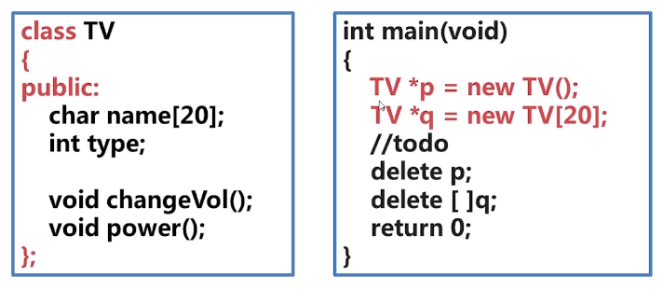
有两种实例化对象的方式：

1. 从栈中实例化：



从栈中实例化对象不需要手动释放内存，系统完成这部分工作，但是在堆中实例化则需要手动释放内存。

1. 从堆中实例化：



在堆中实例化对象需要手动释放内存，如果实例化的是一个对象，则采用delete p的形式，如果是实例化对象数组，那么采用delete []p的形式。

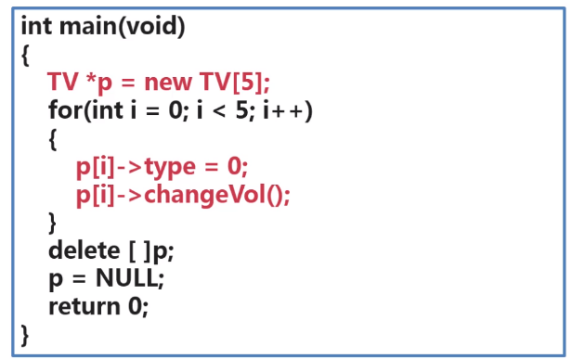
## 对象成员访问

实例化对象后需要访问对象的成员，根据栈和堆两种实例化方式的不同，访问成员的方式也有所不同。

针对单一对象而言访问方式：



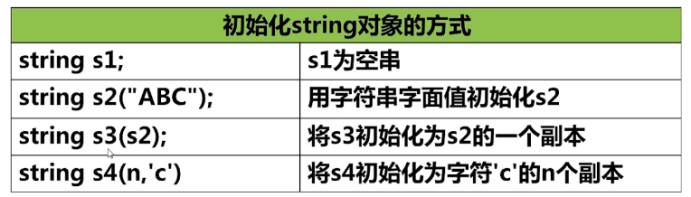
针对对象数组而言访问方式：



## String

C语言中字符处理使用char，C++中可以使用字符串string，这是封装的对象。

### 初始化



### 常用操作



有一种连接方式需要格外注意：



只有string变量与双引号“”的字符串结合的时候才是合法的，两个“”字符串结合是非法的，比如这里的s6。

**拓展：为什么Java的String设置为不可变的呢？**

Java中将String设计成不可变的是综合考虑到各种因素的结果，想要理解这个问题，需要综合内存，同步，数据结构以及安全等方面的考虑：

1、字符串常量池的需要

字符串常量池(String pool, String intern pool, String保留池) 是Java堆内存中一个特殊的存储区域，当创建一个String对象时，假如此字符串值已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象。

假若字符串对象允许改变，那么将会导致各种逻辑错误，比如改变一个对象会影响到另一个独立对象。严格来说，这种常量池的思想，是一种优化手段。

2、允许String对象缓存HashCode

Java中String对象的哈希码被频繁地使用，比如在hashMap 等容器中。

字符串不变性保证了hash码的唯一性，因此可以放心地进行缓存。这也是一种性能优化手段，意味着不必每次都去计算新的哈希码。

3、安全性

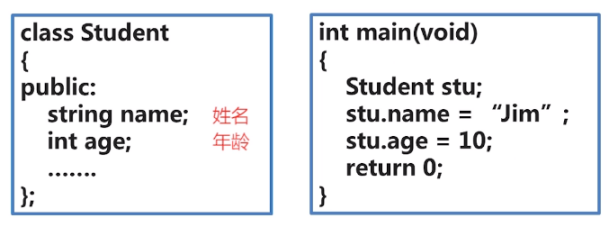
String被许多的Java类(库)用来当做参数。例如：网络连接地址URL，文件路径path，还有反射机制所需要的String参数等，假若String不是固定不变的，将会引起各种安全隐患。

## 数据的封装

所谓数据的封装就是将类中成员变量的操作转换为成员函数的操作，这样才是封装性的体现，即不直接暴露成员变量本身，而是暴露出成员变量操作的函数。

面向对象的思想就是以谁做什么为基础，反映到代码层面上就是类的成员函数。

如下例子就是违背了这种思想：



应该采用如下的方式：



这样具备两种好处：

1、更加符合面向对象的思想；

2、可以根据需要对参数进行校验：



3、可以设置只读属性：

有些时候我们可能需要设置某些成员变量只能被读或者写操作，这样就单独设置仅有getter或者setter中的某一个操作函数即可。



## inline/内联函数

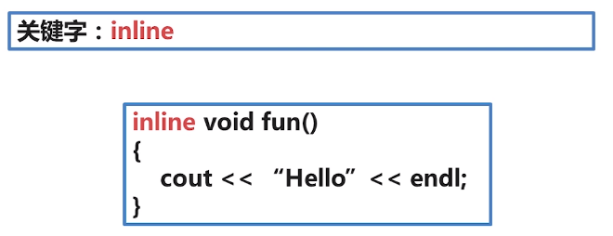
**C++中推荐使用内联函数替代宏代码片段；**

宏代码块看上去像函数但实际不是函数，因此其使用常带有副作用；

消除这样的副作用需要函数，但是函数的调用有入栈返回等开销，这一点宏代码块没有；

综合两者有点，避免两者缺点，内联函数被提出来。

### 语法



### 普通函数

内联函数与普通函数的区别：



内联函数相当于省去了2、4调用和返回的过程，只有一个3函数本身执行的过程（即**内联函数没有普通函数调用时的额外开销（压栈，跳转，返回）**）。这样**效率更高，速度更快**，但是仅仅适用于结构比较简单的函数，如果函数逻辑复杂，即使设置内联函数，编译器也会拒绝按照内联函数的方式处理。

### 虚函数

虚函数可以被声明为inline，这是毫无疑问的，因为inline同register一样，只是对编译器的建议。

**那么inline virtual函数是否会被内联展开呢？答案是可以！**

有的人说不能，理由是virtual的意思是"等到运行时再决定调用哪个函数"，inline的意思是"在编译期间将调用之处用被调函数来代替"，如果编译器甚至还不知道哪个函数将被调用，当然就不能责怪它拒绝生成内联调用了。

其实上面的观点是不正确的，如果virtual可以在编译期决定调用什么函数，那么就可以被inline。

**那么什么时候可以决定调用何种函数呢？**

用一个类对象通过成员选择符.调用虚函数，如obj.vf()。这时虚函数vf()就可以被vc优化内联展开。这样调用等于告诉编译器你要调用的具体函数，在函数有inline修饰或是体内定义的情况下就会被内联展开。

当然还有其他情况可以被inline，总的意思是虚函数可以申明为inline，具体到编译器解释时候，如果编译器在编译的时候就可以确定该虚函数的决议，则编译器以inline方式静态决议该虚函数。如果编译器在编译的时候不能决定，则必须在运行时决议虚函数，此时虚函数不能以inline函数的方式调用。

### 特征

1、内联函数具有普通函数的特征（参数检查，返回类型等）；

类型更加安全；

C++ 提供各种方式让大家摒弃 C 中不好的一些特性，内联函数就是其中之一；

2、函数的内联请求可能被编译器拒绝；

可以通过配置的方式让编译器支持内联函数的请求；

3、函数被内联编译后，函数体直接扩展到调用的地方；

如果内联成功，其效率上可以和宏代码块媲美，且其在类型上比宏代码块安全很多，因此在 C++ 编程中，首选内联函数；

宏代码片段由预处理器处理，进行简单的文本替换，没有任何编译过程，因此可能出现副作用；

### 限制

C++中inline内联编译的限制：

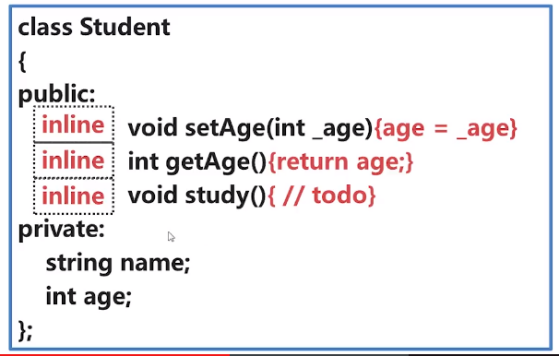
1. 不能存在任何形式的循环语句；
2. 不能存在过多的条件判断语句；
3. 函数体不能过于庞大；
4. 不能对函数进行取址操作；
5. 函数内联声明必须在调用语句之前；

总的来说就是函数体不能过于复杂，这个复杂没有界定，对于现在C++编译器，只要函数不是太夸张，几乎都可以满足请求。

## 类内/类外定义

### 类内定义

如果将成员函数和函数体写在类的内部，这就是类内定义。类内定义的成员函数，编译器会优先将其编译为inline函数，它不会显式地标识inline，而是优先按照inline编译。对于复杂的函数，无法编译为内联函数的，则编译为普通函数。

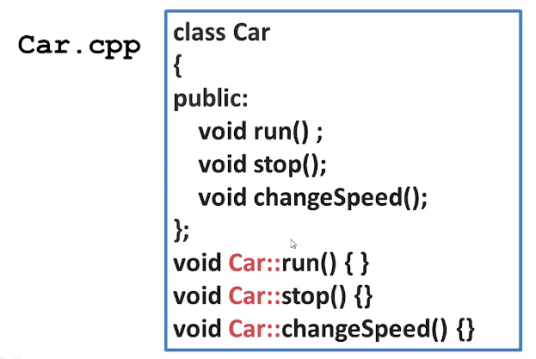


既然将这个函数实现定义在类内部，系统就默认为这个函数比较短小，可以采用inline方式编译，如果是分开的那么就不会默认为该函数非常短小了。

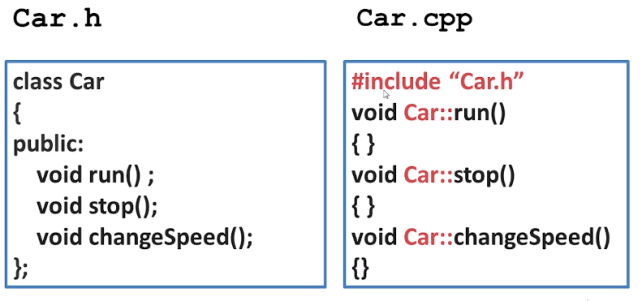
### 类外定义

类外定义是成员函数和函数体写在类的外面。包括两种：同文件类外定义和分文件类外定义。

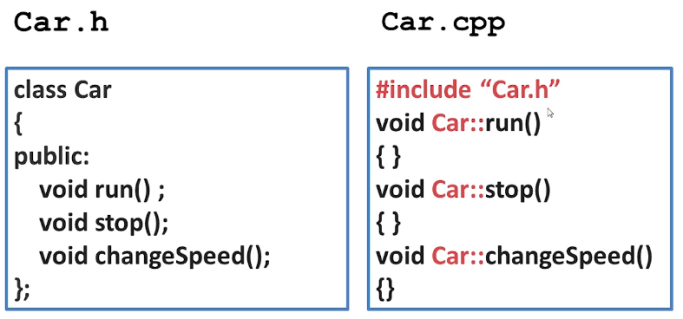
同文件类外定义：



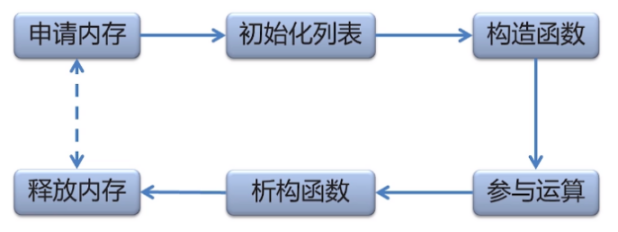
分文件类外定义（常用）：



专业的C++项目都是按照分文件定义的方式来组织代码的。首先，要定义一个头文件.h，这里声明成员变量和成员函数；然后，在.cpp文件中导入.h文件（即声明），实现具体函数。



## 构造函数



### 内存分布/内存模型

要想知道对象是怎么存储的，需要知道内存按照用途是怎么划分的：



注：在构造函数的时候，其实就是构造内存模型。

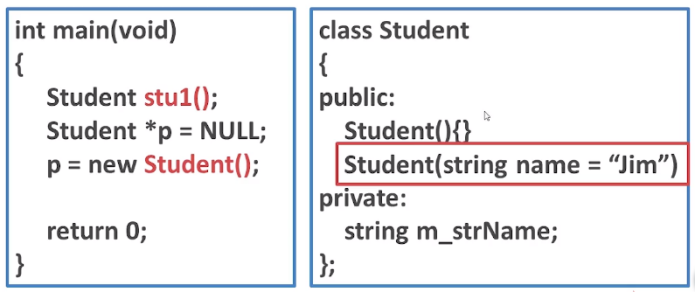
### 对象初始化

一个对象如果new之后没有进行设置，那么就相当于我们知道了地址但是没有内容，所以需要先进行对象的初始化操作。

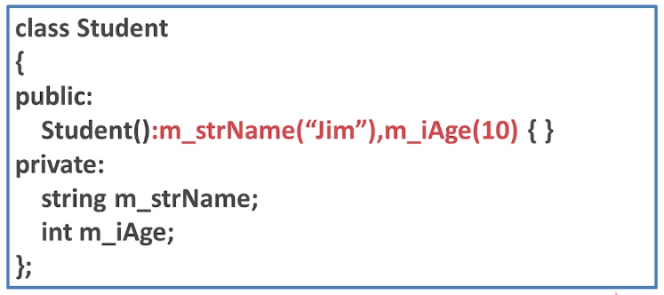
对象的初始化包括：有且仅有一次，根据条件初始化。

有且仅有一次的初始化就是构造函数，构造函数没有返回值，可以重载，但是实例化对象时只用到其中一个构造函数。

在实例化对象的时候，如果不传入参数，则成为默认的构造函数，如下：



### 初始化列表



注：使用初始化列表进行初始化的时候，需要在构造函数后加“：”，赋值操作不使用“=”，而是使用“()”，多个类成员变量之间使用“,”。

初始化列表的一些基本特性：

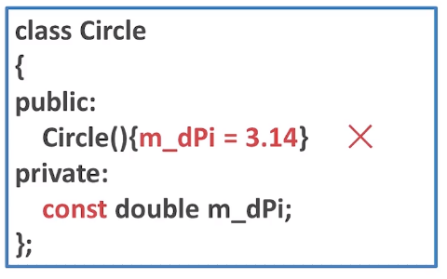
1、初始化列表先于构造函数执行；

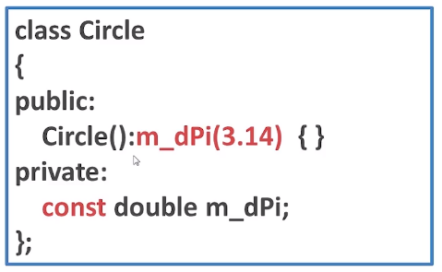
2、初始化列表只能用于构造函数；

3、初始化列表可以同时初始化多个数据成员。

注：初始化列表赋值方式和构造函数在顺序上有区别。

既然初始化列表的工作可以由构造函数完成，那么为什么还需要初始化列表呢？





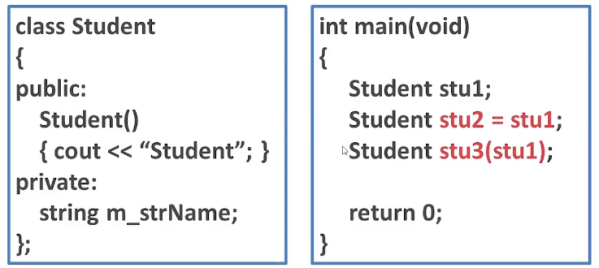
注：对于const类型的成员变量，如果在构造函数中进行初始化，则会报错，只能在构造函数之前（即真正创造对象前）进行初始化赋值。

**实例：**





### 拷贝构造函数



注：这里只输出一个Student（即实际上只调用了1次构造函数，stu1调用的），并不是我们预期的3个（按照预期应该stu1,stu2,stu3调用3次构造函数），这是因为这里没有调用我们预期的Student()构造函数，而是调用了默认的拷贝构造函数。

拷贝构造函数定义：



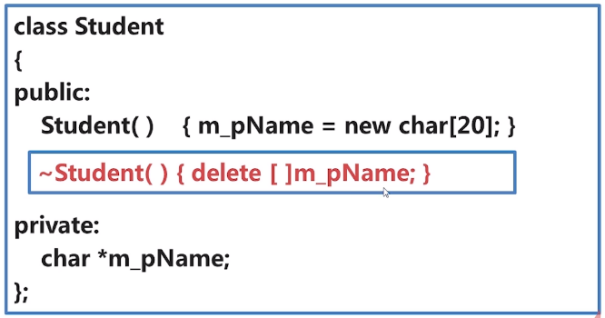
注：拷贝构造函数参数是const+对象的引用

说明：

1、如果没有自定义的拷贝构造函数则系统默认自动生成一个默认的拷贝构造函数；

2、当采用直接初始化或者复制初始化实例化对象时系统自动调用拷贝构造函数。

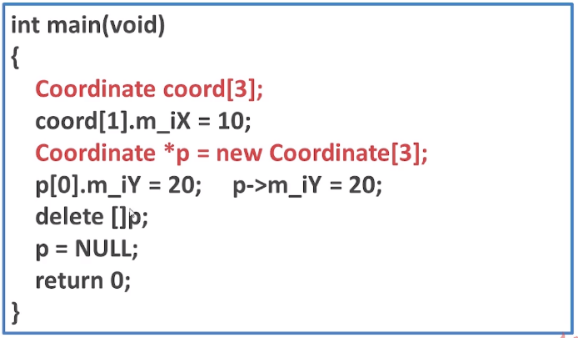
## 析构函数

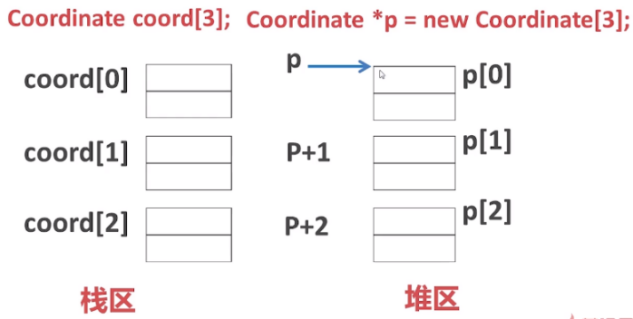


说明：

**析构函数没有参数不允许重载！**

## 对象数组

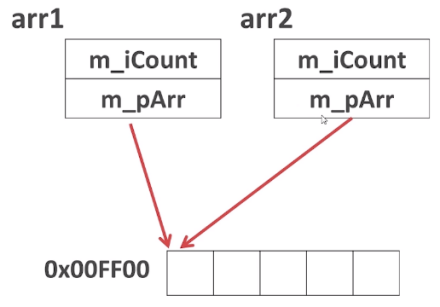




## 深浅拷贝

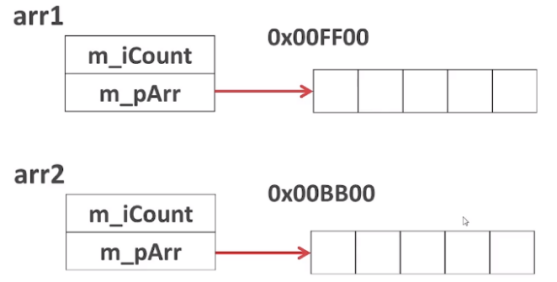
### 浅拷贝





浅拷贝只是将数据成员的值进行简单的拷贝（不管是不是指针），上述arr1和arr2会同时指向一个内存地址，如果往arr1写入一个值，然后再往arr2写入另一个值，那么这个地址对应的值就会被重写覆盖原来的值，这显然不是我们期望的。还有更严重的，在销毁arr1的时候，为了避免内存泄露，会释放m\_pArr的内存，同理arr2在销毁的时候也会释放这个内存，这样同一个内存地址释放了两次，计算机会core掉。

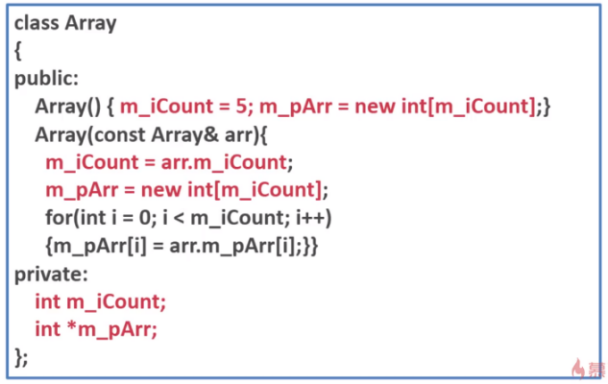
我们期望的是内存地址指向的数据依次执行拷贝，即如下：



即在拷贝构造函数中不是简单的赋值操作，而是重新分配内存。

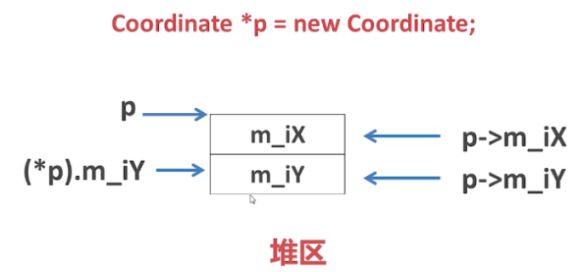
### 深拷贝

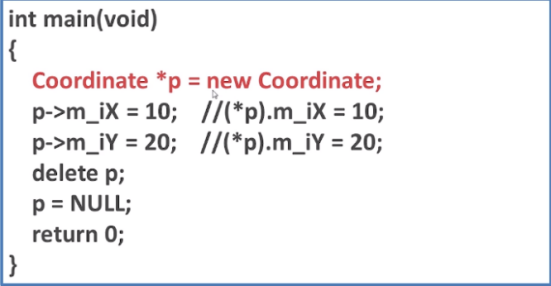
深拷贝不是简单的执行拷贝，而是将堆中的内存数据执行拷贝。



注：与浅拷贝不同的是，在自定义拷贝构造函数中使用new生成一个新的对象。

## 对象指针

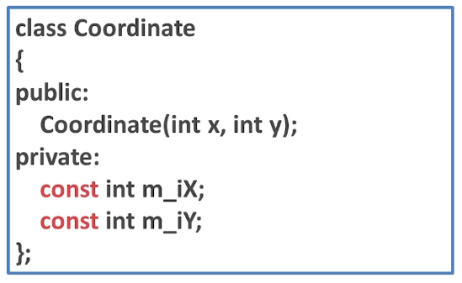


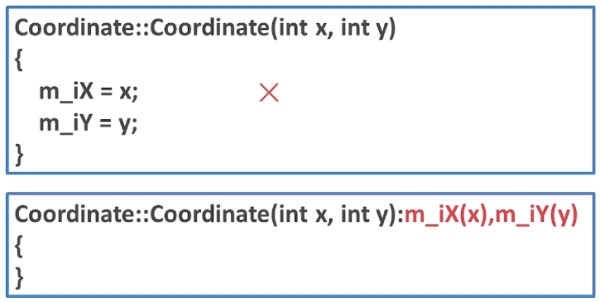


注：(\*p).x与p->x一样

## const

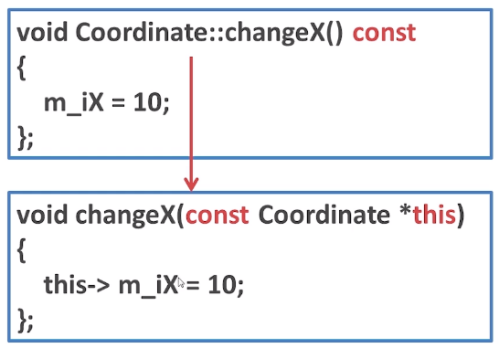
### 常对象成员



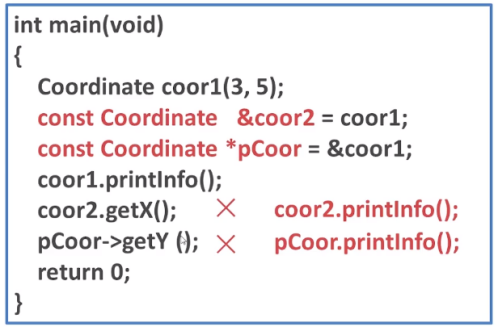


注：常对象成员即一旦被初始化就不允许改变，此时使用初始化列表进行赋值，在有些应用场景是期望这种对象的。

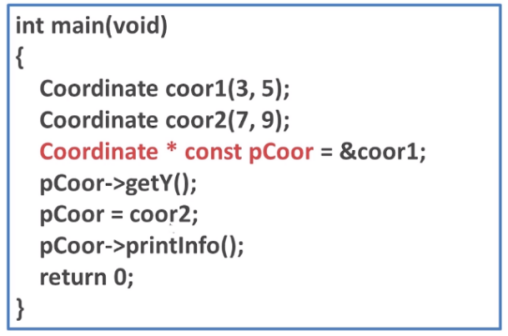
### 常成员函数



### 常指针与常引用



可以采用如下：



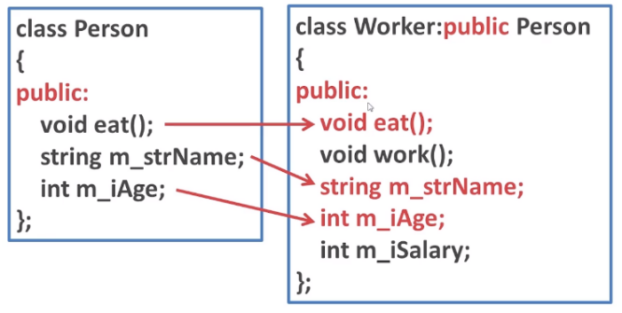
# 继承性

## 公有继承

class A : public B

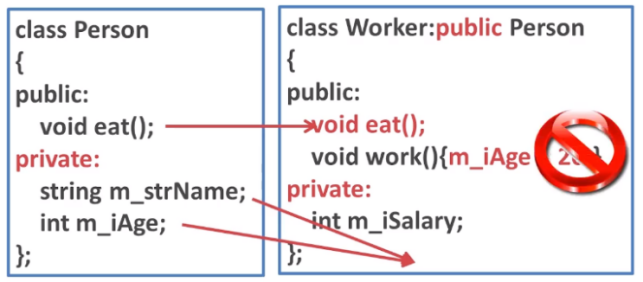


1、public成员变量公有继承



2、private成员变量公有继承

在继承中，private成员变量被继承在子类不可见的位置，而不是private的位置，因此不能在子类中访问父类的private变量。

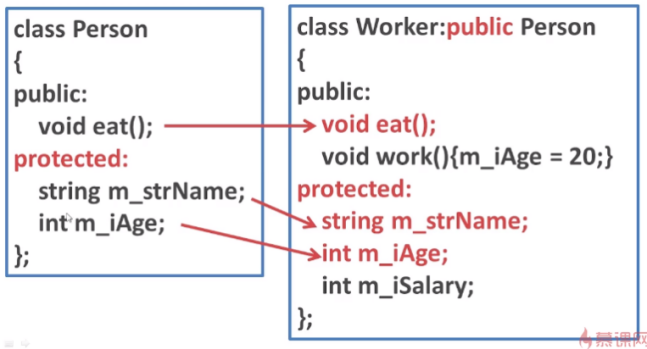


3、protected成员变量继承



说明：当在子类中访问父类protected成员变量时会报错。

在公有继承情况下protected成员变量的访问：



可以通过公有继承的子类的成员函数访问继承来的protected类型的成员变量。

## 私有继承

class A : private B



私有继承是has a的关系，即包含关系，子类对象包含一个父类对象，比如线段和坐标。

## 保护继承

class A : protected B

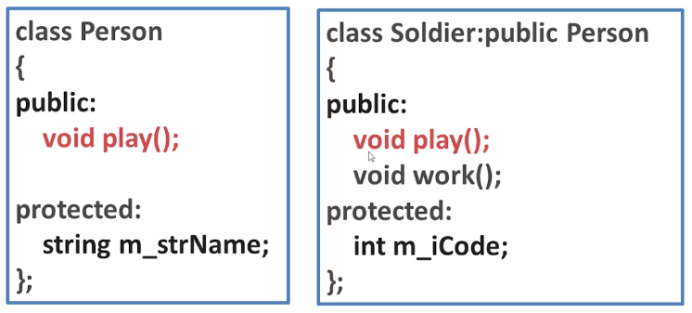


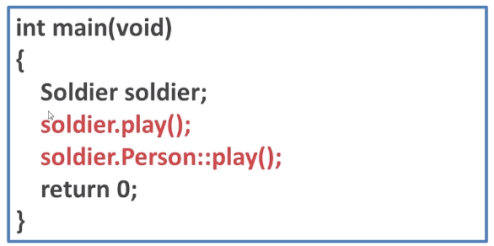
与公有继承不同的是，父类的public成员变量被继承在子类的protected控制符下面。

## 隐藏

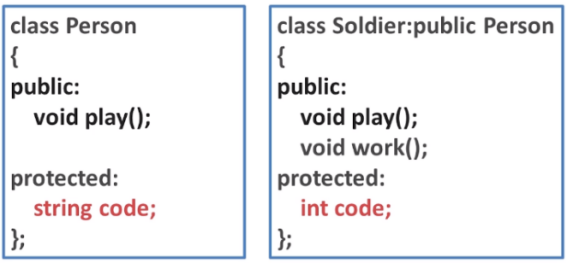
### 同名

如果子类和父类同时定义同一个方法，则其访问方式为：





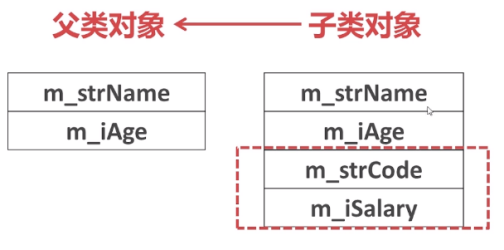
如果子类和父类同时定义了同一个成员变量，则其访问方式为：



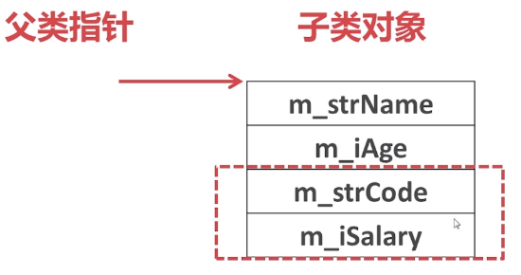


其实针对上述这样的成员变量“重名”的情况，尽量避免使用。也可以采用数据类型不同确定不同名称，比如父类中string类型成员变量命名为m\_scode，子类中int类型成员变量命名为m\_icode。

### is a



子类对象赋值给父类（或者子类初始化父类），其含义就是用子类继承自父类的成员变量给父类中的对应成员变量赋值，父类中不具备子类的成员变量丢失。



如果父类指针访问子类对象，那么它也只能访问到子类中继承自父类的成员变量，而不能访问子类特有的成员变量。

## 多继承/多重继承

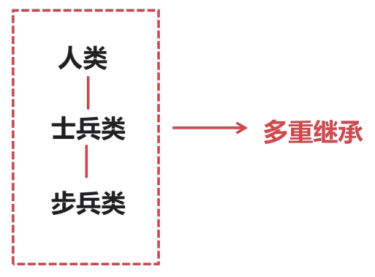
**多重继承：**

多重继承特点总结如下：

1、多重继承与多继承不同，当B类从A类派生，C类从B类派生，此时称为多重继承；

2、当实例化子类时，会首先依次调用所有基类的构造函数，最后调用该子类的构造函数；销毁该子类时，则相反，先调用该子类的析构函数，再依次调用所有基类的析构函数；

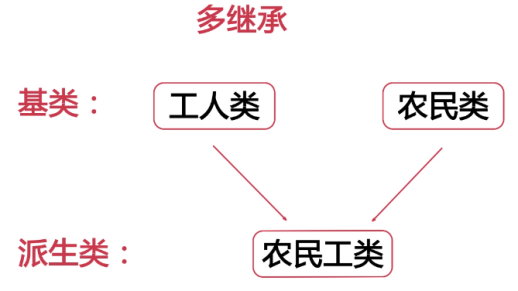
3、无论继承的层级有多少层，只要它们保持着直接或间接的继承关系，那么子类都可以与其直接父类或间接父类构成 is a的关系，并且能够通过父类的指针对直接子类或间接子类进行相应的操作，子类对象可以给直接父类或间接父类的对象或引用赋值或初始化。





**多继承：**

多继承是指一个子类继承多个父类。多继承对父类的个数没有限制，继承方式可以是公共继承、保护继承和私有继承，不写继承方式，默认是private继承。



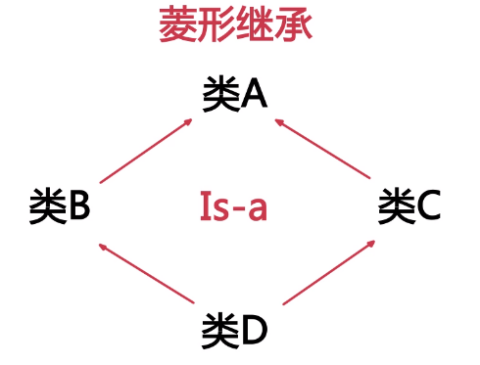


注：**多重继承会使得继承显得混乱**，所以Java中没有引入。

## 虚继承

### 背景

**虚继承是为了解决多继承和多重继承带来的问题**，比如菱形继承中存在的多个父类的问题：



在类D中存在两个完全一样的类A的副本，这种冗余数据我们不能承担带来的额外的开销，显然不是我们希望的。

类 A 派生出类 B 和类 C，类 D 继承自类 B 和类 C，这个时候类 A 中的成员变量和成员函数继承到类 D 中变成了两份，一份来自 A-->B-->D 这条路径，另一份来自 A-->C-->D 这条路径。

在一个派生类中保留间接基类的多份同名成员，虽然可以在不同的成员变量中分别存放不同的数据，但大多数情况下这是多余的：因为保留多份成员变量不仅占用较多的存储空间，还容易产生命名冲突。假如类 A 有一个成员变量 a，那么在类 D 中直接访问 a 就会产生歧义，编译器不知道它究竟来自 A -->B-->D 这条路径，还是来自 A-->C-->D 这条路径。下面是菱形继承的具体实现：

//间接基类A

class A{

protected:

int m\_a;

};

//直接基类B

class B: public A{

protected:

int m\_b;

};

//直接基类C

class C: public A{

protected:

int m\_c;

};

//派生类D

class D: public B, public C{

public:

void seta(int a){ m\_a = a; } //命名冲突

void setb(int b){ m\_b = b; } //正确

void setc(int c){ m\_c = c; } //正确

void setd(int d){ m\_d = d; } //正确

private:

int m\_d;

};

int main(){

D d;

return 0;

}

这段代码实现了上图所示的菱形继承，第 25 行代码试图直接访问成员变量 m\_a，结果发生了错误，因为类 B 和类 C 中都有成员变量 m\_a（从 A 类继承而来），编译器不知道选用哪一个，所以产生了歧义。

为了消除歧义，我们可以在 m\_a 的前面指明它具体来自哪个类：

void seta(int a){ B::m\_a = a; }

这样表示使用 B 类的 m\_a。当然也可以使用 C 类的：

void seta(int a){ C::m\_a = a; }

### 原理

为了解决多继承时的命名冲突和冗余数据问题，C++提出了虚继承，使得在派生类中只保留一份间接基类的成员。

在继承方式前面加上virtual关键字就是虚继承，请看下面的例子：

//间接基类A

class A{

protected:

int m\_a;

};

//直接基类B

class B: virtual public A{ //虚继承

protected:

int m\_b;

};

//直接基类C

class C: virtual public A{ //虚继承

protected:

int m\_c;

};

//派生类D

class D: public B, public C{

public:

void seta(int a){ m\_a = a; } //正确

void setb(int b){ m\_b = b; } //正确

void setc(int c){ m\_c = c; } //正确

void setd(int d){ m\_d = d; } //正确

private:

int m\_d;

};

int main(){

D d;

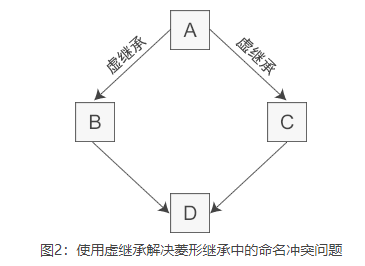
return 0;

}

这段代码使用虚继承重新实现了上图所示的菱形继承，这样在派生类D中就只保留了一份成员变量m\_a，直接访问就不会再有歧义了。

虚继承的目的是让某个类做出声明，承诺愿意共享它的基类。其中，这个被共享的基类就称为虚基类（Virtual Base Class），本例中的A就是一个虚基类。在这种机制下，不论虚基类在继承体系中出现了多少次，在派生类中都只包含一份虚基类的成员。

现在让我们重新梳理一下本例的继承关系，如下图所示：

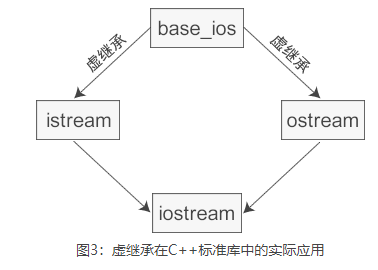


观察这个新的继承体系，我们会发现虚继承的一个不太直观的特征：**必须在虚派生的真实需求出现前就已经完成虚派生的操作。**在上图中，当定义D类时才出现了对虚派生的需求，但是如果B类和C类不是从A类虚派生得到的，那么D类还是会保留A类的两份成员。

换个角度讲，**虚派生只影响从指定了虚基类的派生类中进一步派生出来的类，它不会影响派生类本身。**

在实际开发中，位于中间层次的基类将其继承声明为虚继承一般不会带来什么问题。通常情况下，使用虚继承的类层次是由一个人或者一个项目组一次性设计完成的。对于一个独立开发的类来说，很少需要基类中的某一个类是虚基类，况且新类的开发者也无法改变已经存在的类体系。

C++标准库中的iostream类就是一个虚继承的实际应用案例。iostream从istream和ostream直接继承而来，而istream和ostream又都继承自一个共同的名为base\_ios的类，是典型的菱形继承。此时istream和ostream必须采用虚继承，否则将导致iostream类中保留两份base\_ios类的成员。



虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题，通常可以将派生类对象的地址赋值给基类对象，实现的具体方式是，将基类指针指向继承类（继承类有基类的拷贝）中的基类对象的地址，但是多重继承可能存在一个基类的多份拷贝，这就出现了二义性。

虚继承可以解决多种继承前面提到的两个问题：

虚继承底层实现原理与编译器相关，一般通过虚基类指针和虚基类表实现，每个虚继承的子类都有一个虚基类指针（占用一个指针的存储空间，4字节）和虚基类表（不占用类对象的存储空间）（需要强调的是，虚基类依旧会在子类里面存在拷贝，只是仅仅最多存在一份而已，并不是不在子类里面了）；当虚继承的子类被当做父类继承时，虚基类指针也会被继承。

实际上，vbptr指的是虚基类表指针（virtual base table pointer），该指针指向了一个虚基类表（virtual table），虚表中记录了虚基类与本类的偏移地址；通过偏移地址，这样就找到了虚基类成员，而虚继承也不用像普通多继承那样维持着公共基类（虚基类）的两份同样的拷贝，节省了存储空间。

在这里我们可以对比虚函数的实现原理：他们有相似之处，都利用了虚指针（均占用类的存储空间）和虚表（均不占用类的存储空间）。

虚基类依旧存在继承类中，只占用存储空间；虚函数不占用存储空间。

虚基类表存储的是虚基类相对直接继承类的偏移；而虚函数表存储的是虚函数地址。

### 虚基类成员的可见性

因为在虚继承的最终派生类中只保留了一份虚基类的成员，所以该成员可以被直接访问，不会产生二义性。此外，如果虚基类的成员只被一条派生路径覆盖，那么仍然可以直接访问这个被覆盖的成员。但是如果该成员被两条或多条路径覆盖了，那就不能直接访问了，此时必须指明该成员属于哪个类。

以图2中的菱形继承为例，假设 A 定义了一个名为 x 的成员变量，当我们在 D 中直接访问 x 时，会有三种可能性：

如果 B 和 C 中都没有 x 的定义，那么 x 将被解析为 A 的成员，此时不存在二义性。

如果 B 或 C 其中的一个类定义了 x，也不会有二义性，派生类的 x 比虚基类的 x 优先级更高。

如果 B 和 C 中都定义了 x，那么直接访问 x 将产生二义性问题。

### 方式

虚继承方式：class A : virtual public B

# 多态性

多态的作用：

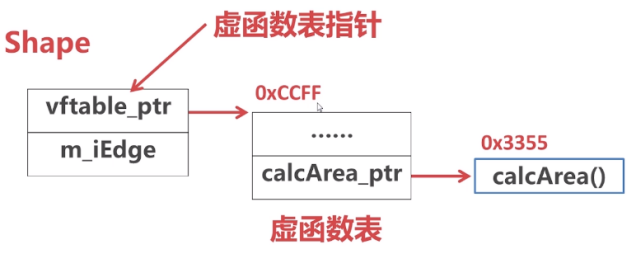
1. 隐藏细节：使得代码能够模块化，拓展代码模块，实现代码重用
2. 接口重用：为了类在继承和派生的时候，保证使用家族中任一类的实例的某一属性时的正确调用。

## 虚函数

### 原理

对象指针：指向对象的指针；

函数指针：指向函数的指针。

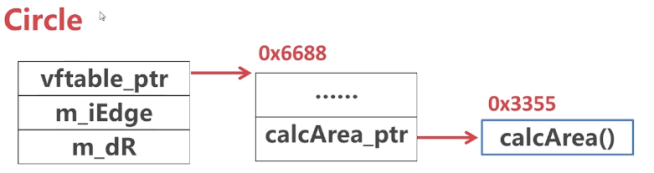


**虚函数表是编译时确定的，运行时会查找该表**。各个虚拟函数这时被组织成了一个虚拟函数的入口地址的数组，而对象的隐藏成员—虚拟函数表指针是在运行期，也就是构造函数被调用时进行初始化的，这就是实现多态的关键。

注：当实例化对象的时候，除了必要的成员变量，还会有一个虚函数表指针的成员变量vftable\_ptr（指针，4字节），它指向虚函数表，**父类的虚函数表只有一个**，**通过父类实例化的子类虚函数表指针vftable\_ptr都是一样的**，以确保每一个子类都指向自己的虚函数表（通过偏移量），在虚函数表中找到对应函数的入口地址。

**注：如何找到虚函数表？即通过虚函数表指针的成员变量vftable\_ptr。**

当实例化Circle的时候，它本身没有虚函数，但是它继承自Shape，故子类Circle也会产生一个虚函数表，但是这个虚函数表中继承自父类的函数的函数入口地址也被继承来了，所以指向的是父类的那个函数：



如果子类覆盖父类的虚函数：



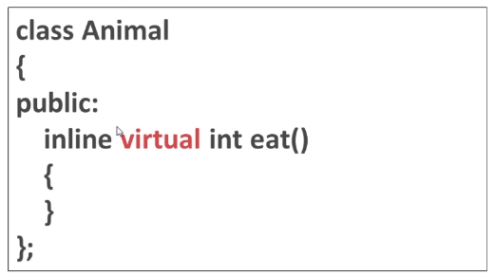
### 限制

1. 不能修饰普通函数（必须是某一类的成员函数，不能是全局函数）；
2. 不能修饰静态成员函数



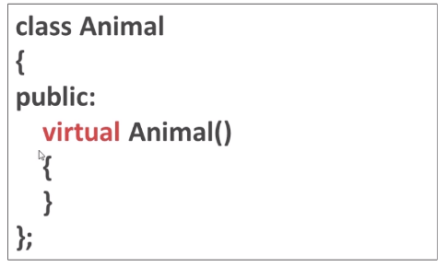
因为static的成员函数不属于任何一个对象，它与类共生存的，所以使用virtual修饰会造成编译报错。

1. 不能修饰内联函数：



如果修饰内联函数，计算机会忽略inline关键字，而使其变为虚函数。

1. 不能修饰构造函数：



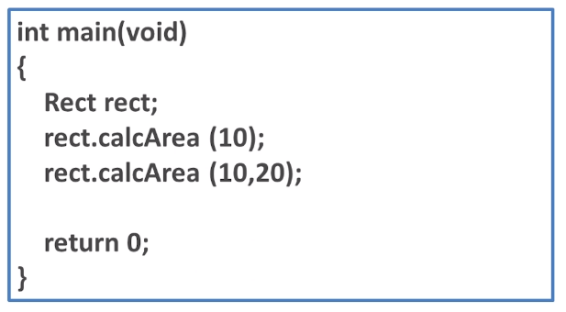
因为构造函数专属于本类，负责本类对象的构造，所以一个类不可能用另一个类的构造函数来构造本类的对象，也就不可能去重定义一个他类的构造函数，也就没有必要使之成为virtual函数。

并且子类对象中父类成分的构造并不与继承机制相同。无论是动态绑定还是静态绑定父类的构造函数总是先被调用，因而可以说构造函数的调用次序是与绑定方式无关的，不存在次序的变更。也可以说它是自动的。

### 静态多态

**静态多态**又叫**早绑定**，在编译阶段就确定使用哪个函数了，比如重载函数，只要在定义的时候设置了参数就可以知道执行哪一个函数了。





#### 为什么要重载

函数重载在一定程序上可以减轻程序员**起名字的负担**。最常见的一个例子就是构造函数的重载。

class Test

{

public:

Test(void); // 无参构造函数

Test(int a);//构造函数

Test(int a,int b);//两个整型参数的构造函数

};

可以看到，类Test的三个构造函数名都为Test。如果没有重载，要实现三个构造函数就可能需要三个不同的构造函数名区分，这也就增加了类的使用者的负担，使用者需要传入不同参数构造对象时，就需要使用不同的构造函数名称。而有函数重载之后，类的使用者可以使用同一个函数名传入不同的参数即可。

当然了，如果单纯地为了减轻起名字的负担而去使用函数重载，而使得函数失去了本来的信息，则是一个不明智的选择。我们可以为那些操作确实极其相似的函数进行重载。

#### 不能重载的情况

以下几种情况下，是不能重载或者说是非法的。

**main函数不能重载**

这是在C++ 11标准中说明的：

A program shall contain a global function called main, which is the designated start of the program....

This function shall not be overloaded.

试想如果作为用户程序入口函数的main函数被重载了，那么加载的时候该以哪个为入口呢？

**只有返回值不同**

例如下面两个声明只有返回值不同，函数名和形参都相同：

double calcArea(const Square&);

int calcArea(const Square&); //非法，仅有返回值不同，不可重载

/\*以上声明同时出现会报错\*/

试想一下，当你传入Square类型参数，而不去使用返回值时，应该调用上面的哪个函数呢？

**形参列表看似不同，实则相同**

例如使用typedef给Triangle起了一个“别名”：

typedef Triangle MyTri;

double calcArea(const Triangle&);

double calcArea(const MyTri&);

/\*以上声明同时出现会报错\*/

上面这种情况的形参看似不一样，本质上来说它们并没有什么不同。

**形参名不同**

例如：

double calcArea(const Circle &circle );//形参名为circle

double calcArea(const Circle& cir);//形参名为cir

double calcArea(const Circle& );//省略形参名

/\*以上声明同时出现会报错\*/

这里形参的名字仅仅是起到说明或者记忆的作用，因此对于上面三个声明，它们的形参名可以随意起，但不会影响形参列表的内容。

**仅有顶层const的差异**

例如：

double calcArea(const Circle);//函数1

double calcArea(Circle);//重复声明了函数1

/\*以上声明同时出现会报错\*/

double calcArea(Circle\* const);//函数2

double calcArea(Circle\*);//重复声明了函数2

/\*以上声明同时出现会报错\*/

但需要特别注意的是，**如果形参是指针或引用，是可以通过区分指向到底是常量对象还是非常量对象来实现函数重载**。例如下面的情况是可以实现函数重载的：

double calcArea(const Circle&);//作用于常量引用

double calcArea(Circle&);

/\*以上声明同时出现不会报错\*/

double calcArea(const Circle\*);//作用于常量指针

double calcArea(Circle\*);

/\*以上声明同时出现不会报错\*/

#### 总结

在定义了重载函数后，我们需要以合理的实参进行调用。大多数情况下，我们很容易判断传入的对应实参需要调用哪个函数，但是有些时候却并不那么容易。我们将会在后面的文章中看到如何进行函数匹配。

函数重载能够减轻程序员命名的负担，但这不应该以丢失可读性为代价。

main函数不能重载。

重载函数的形参在数量或者类型上要有不同。

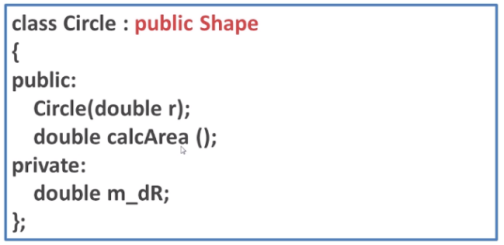
不能以返回值作为函数重载要素。

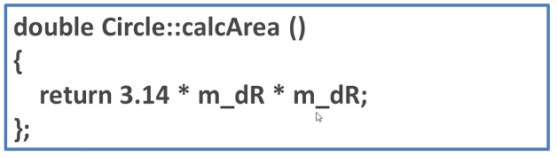
C语言没有函数重载。

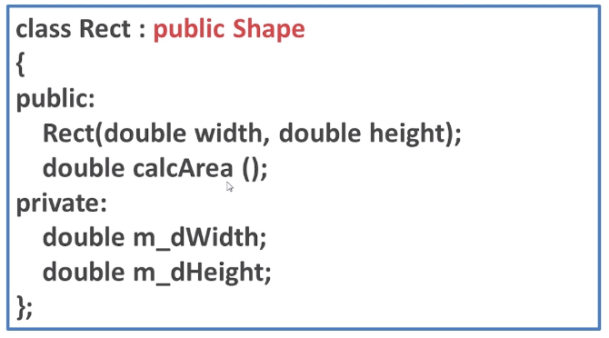
### 动态多态

**动态多态**又叫**晚绑定**，可以理解为对不同的对象下发相同的指令，但是却执行不同的操作，即在真正运行的时候确定执行的函数。动态多态的前提是必须以封装和继承为前提才可以。









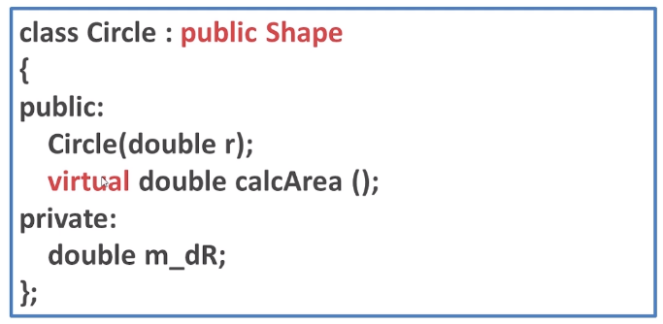


如果采用如下的调用方式，此时调用的都是父类的计算面积，即打印两行“cal area”，而不是调用子类具体的计算函数，这个结果不是我们想要的：

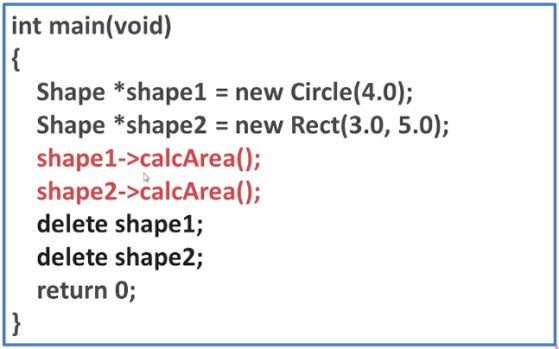


如果想要实现动态多态，需要使用virtual虚函数，对函数做如下修改：

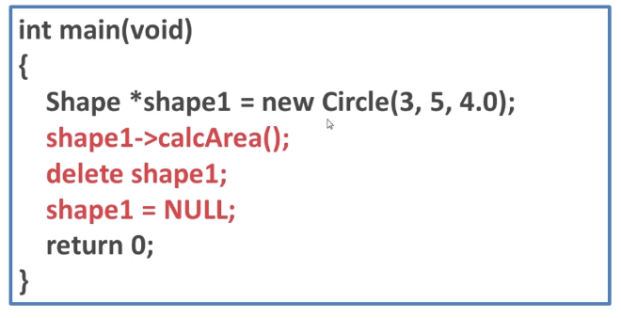




在定义子类的时候继承自父类的virtual函数加一个virtual，如果不加默认会添加virtual，但是还是推荐显式增加virtual关键字。

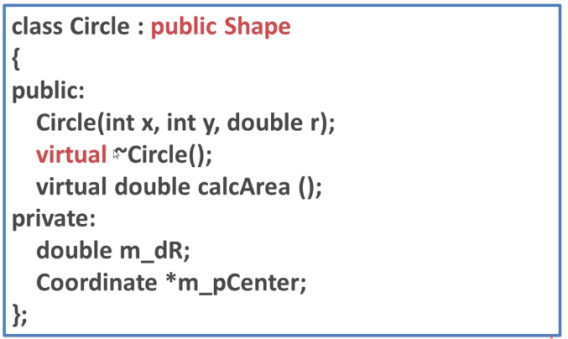


### 虚析构函数

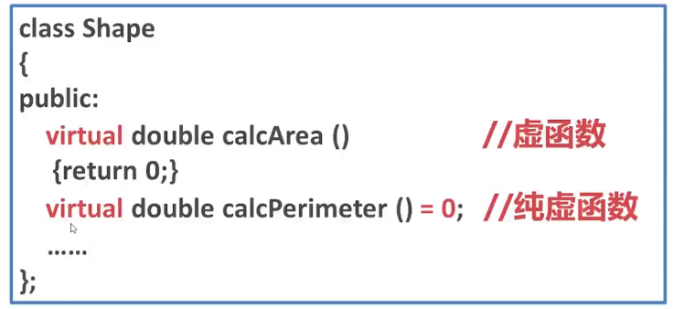


在上面的例子中，调用虚函数是正确的，执行结果与预期一致，但是在delete内存释放的时候就会有问题，这里释放的是父类的指针，则调用的是父类的析构函数，并不会调用子类Circle的析构函数，那么就造成内存泄露了。因此引入了虚析构函数，即用virtual修饰析构函数，此时就不会造成内存泄露。





## 纯虚函数





注：如果是虚函数，则在父类初始化后的虚函数表中存储的是一个有意义的地址，这个可以供子类使用，但是如果是纯虚函数，则这个地址是0，无意义，只能等到子类实例化之后赋值才有意义。

## 抽象类

包含纯虚函数的类叫做抽象类，抽象类的子类也可能是抽象类。

抽象类无法实例化对象。

抽象类的子类也可能是抽象类。

## 接口类

仅含有纯虚函数的类叫做接口类。

## RTTI

RTTI（Run-Time Type Identification），即运行时类型识别。

Typeid注意事项：

1. type\_id返回一个type\_info对象的引用；
2. 如果想通过基类的指针获得派生类的数据类型，基类必须带有虚函数；
3. 只能获取对象的实际类型。

dynamic\_cast注意事项：

1. 只能应用于指针和引用的转换；
2. 要转换的类型中必须包含虚函数；
3. 转换成功返回子类的地址，失败返回NULL。

## 异常处理

# 模板编程

C++的模板编程体现在函数上就是模板函数，体现在类上就是模板类，这些都可以用标准模板类表示。

## 友元函数

### 友元全局函数

函数定义在全局的，且是友元函数。

函数声明如下：



友元函数的参数传入一个对象的引用或者指针，目的得能够通过这个参数访问到类中的私有或者保护的成员变量/函数，这样才能体现出友元函数的优势。

使用方法如下：



传入引用或指针执行效率更高，因此推荐这种入参，如果直接传入对象（传值）则效率较低。这里是使用类的对象直接访问私有数据，因为这个方法已经声明为friend类型的了。

### 友元成员函数

函数定义在类中，并且声明为另外一个类的友元函数，叫做友元成员函数。

函数声明如下：



这里的友元函数printXY不再是类Coordinate的成员函数，而是Circle类的成员函数。

使用方法如下：



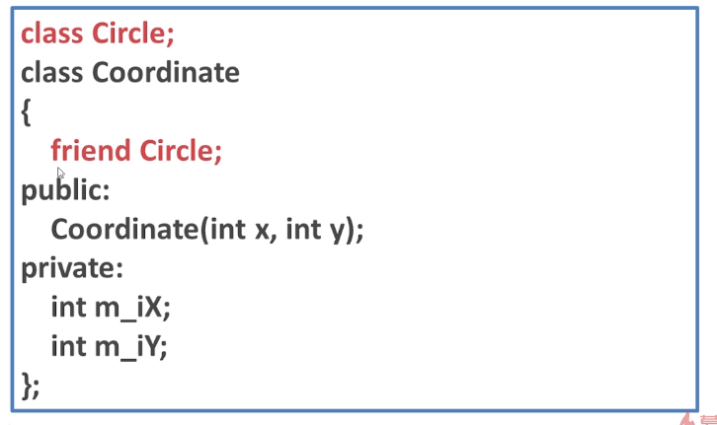
友元函数可以方便地访问另外一个类的私有数据，这是友元函数的优势；但是，友元函数破坏了封装性，出问题时比较难排查。一般情况下，少使用友元函数。

注：友元函数位置没有特殊要求，可以放在public、private、protected访问控制符下面，但是一般情况下我们放置在最前面，这样更加明晰。

## 友元类

友元只是封装的补充，并不是一种很好地语法，是不得已而为之。如果前期设计的好，完全可以避免使用友元。

友元类的声明如下：



在声明友元类之前，先使用class声明一下类，这里是告诉计算机存在Circle类，这里只是使用一下，后面会实现的，放置编译报错。

使用方法如下：



注意：

1、友元关系不可传递；

2、单向性；

3、友元声明的形式和数量不受限制。

## static

使用关键字static声明静态成员和函数：



### 静态数据成员

静态数据成员不依赖于对象存在，它不会在对象的构造函数中初始化，而是在构造函数前就存在，其声明周期比对象的久，存在于程序运行中。

静态成员函数依赖于类存在，不依赖于对象，而普通成员函数依赖于对象存在。

初始化方法：

类型 类名::变量名 = 变量值

访问方法：

一种是直接通过类名::变量名，一种是通过类对象访问（普通数据成员也采用该方法）。



注意：

1、静态数据成员必须单独的初始化，因为它不随着对象的产生而产生（因为构造函数析构函数不适用于静态数据成员）；

2、静态成员函数不能调用非静态成员函数和非静态数据成员；

3、静态数据成员只有一份，且不依赖对象而存在（如果用sizeof求对象大小，一定是不包含静态数据成员的）。

### 静态成员函数

## 运算符重载

C++中字符拼接“+”以及cout都是运算符重载：



运算符重载的本质就是函数重载。

关键字：operator

### 一元运算符

比如：-（负号）的重载（友元函数重载/成员函数重载），++运算符重载

成员函数重载：





使用：



友元函数重载：





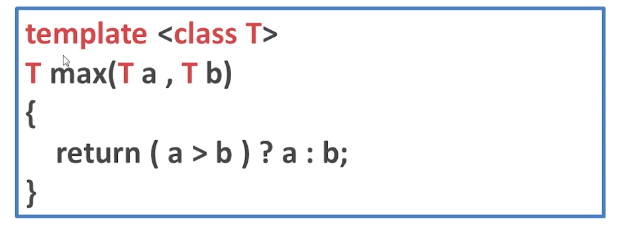
### 二元运算符

## 函数模板

模板技术就是类型参数化，编写代码可以忽略类型。为了让编译器区分是普通函数还是模板函数，采用关键字template区分。

函数模板实现原理：函数模板-->模板函数-->被调用

关键字：template typemane class



注：typename<class T>和template <typename T>可以混用。

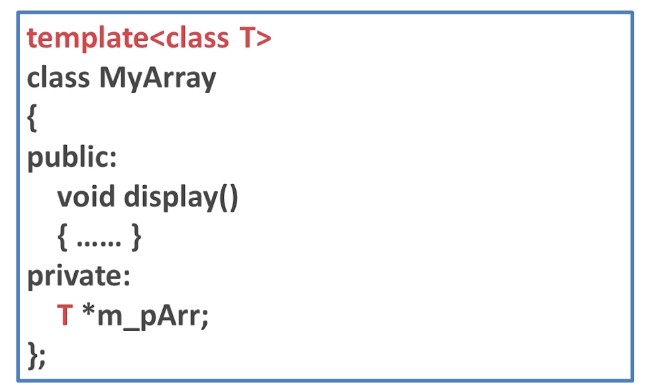
函数模板与普通函数的区别：

1. 函数模板不允许自动类型转换
2. 普通函数能够自动进行类型转化

函数模板和普通函数在一起调用规则：

1. 函数模板可以像普通函数那样被重载
2. C++编译器优先考虑普通函数
3. 如果函数模板可以产生一个更好的匹配，那么选择模板
4. 可以通过空模板实参列表的语法限定编译器只能通过模板匹配

## 类模板



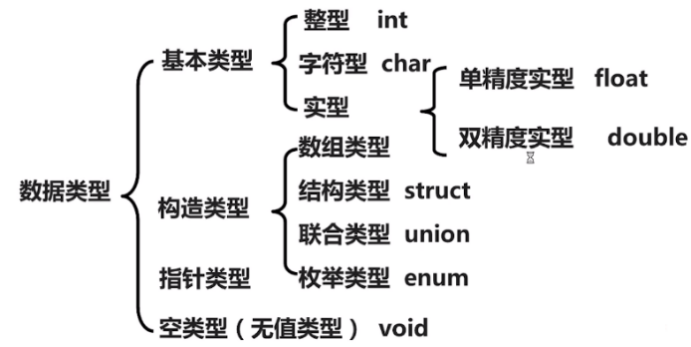
注：模板代码不能分离编译。

## 标准模板类

# C++与C

## 数据类型

C语言中基本数据类型：



C/C++中的结构体struct：

C语言struct仅仅是对数据类型的封装，不能封装函数指针，C++可以。

封装性：C语言struct默认是public访问，不具备保护权限，C++中默认是pricate。

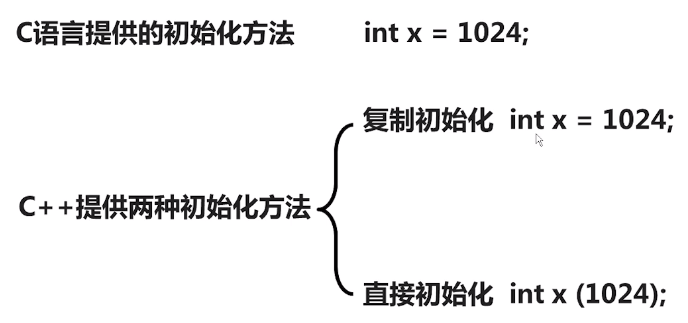
C++新增bool类型变量：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 逻辑类型 | 真 | 假 |
| C | 没提供 | 非0 | 0 |
| C++ | bool | true | false |

C语言的缺点：

1. C风格字符串（C语言没有字符串类型，使用\0作为结束符的字符数组表示字符串）：容易导致程序崩溃或执行恶意代码（很容易导致越界读写非法内存，如果写内存无效，则导致崩溃，如果非法读是有效的，则会出现不可预料的后果，如果非法写有效，则会植入恶意代码）
2. 动态内存管理：容易导致内存溢出，程序和系统僵死
3. C语言语法限制不严格，各种类型可以强制转换

## 初始化方法



## 随用随定义

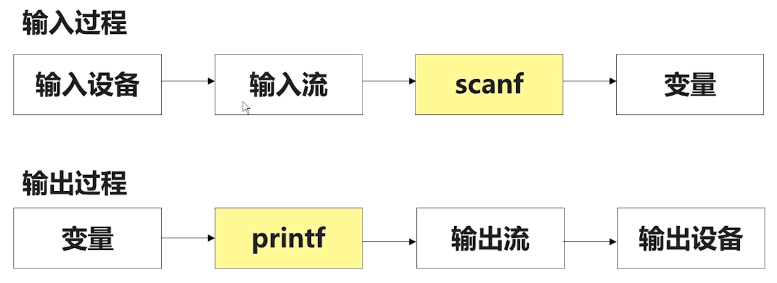
C 语言：所有变量定义必须位于函数体的最前面

C++：所有变量随用随定义

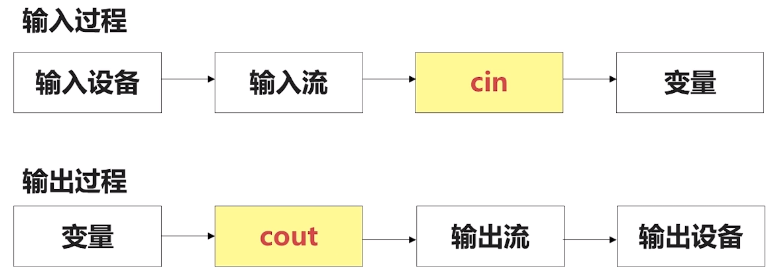


## 输入输出

C语言I/O方式：



C++语言I/O方式：



注：C++这种输入输出方式的好处：

1. 不用关注占位符
2. 不用关注数据类型
3. 不易出现问题

## namespace



# C++11

## 演进

**C++演进：**

C++98（1.0）

C++03（TR1，TTechnical Report 1）

C++11（2.0）

C++14

**新特性：**

C++2.0新特性包括语言和标准库两个层面，标准库以header files头文件形式呈现。

语言：

Variadic Templates

Move Semantics

Auto

Range-base for loop

Initializer list

Lambda

标准库：

Type\_traits

Unordered容器

Forward\_list

Array

Tuple

Con-currency

RegEx

在测试C++11新特性前，需要确认支持C++11：macro \_\_cplusplus

如果是支持C++11，则\_\_cplusplus定义为：

#define \_\_cplusplus 2011031

否则如果是C++98或C++03：

#define \_\_cplusplus 1997111

注：可以直接cout<<\_\_cplusplus根据输出结果，查看是否支持新特性。

## 兼容性

## 常量

### nullptr/nullptr\_t

C++11允许使用nullptr代替0或者NULL。

C++11标准不仅定义了指针空值常量nullptr，也定义了其指针空值类型nullptr\_t，也就表示了指针空值类型并非仅有nullptr一个实例。通常情况下，也可以通过nullptr\_t来声明一个指针空值类型的变量。

大致规则如下：

1. 所有定义为nullptr\_t类型的数据都是等价的，行为也是完全一致；
2. nullptr\_t类型数据可以隐式转换成任意一个指针类型；
3. nullptr\_t类型数据不能转换为非指针类型，即使使用reinterpret\_cast<nullptr\_t>()的方式也是不可以的；
4. nullptr\_t类型数据不适用于算术运算表达式；
5. nullptr\_t类型数据可以用于关系运算表达式，但仅能与nullptr\_t类型数据或者指针类型数据进行比较，当且仅当关系运算符为==、<=、>=等时返回true。

### constexpr

## 变量

### 强枚举类型

### if/switch

在传统C++中，变量的声明虽然能够位于任何位置，甚至于for语句内能够声明一个临时变量int，但始终没有办法在if和switch语句中声明一个临时的变量。

### 一致化初始化uniform initialization

在C++11之前，初始化可以使用{}，()或者=符号，在C++11中引入一致化初始化，你可以只使用()就完成初始化。

### 初始化列表initialize lists

初始化是一个非常重要的语言特性，最常见的就是在对象进行初始化时进行使用。在传统C++中，不同的对象有着不同的初始化方法，例如普通数组、POD（Plain Old Data，即没有构造、析构和虚函数的类或结构体）类型都可以使用{}进行初始化，也就是我们所说的初始化列表。而对于类对象的初始化，要么需要通过拷贝构造、要么就需要使用()进行。这些不同方法都针对各自对象，不能通用。

为了解决这个问题，C++11首先把初始化列表的概念绑定到了类型上，并将其称之为 std::initializer\_list，允许构造函数或其他函数像参数一样使用初始化列表，这就为类对象的初始化与普通数组和POD的初始化方法提供了统一的桥梁。

初始化列表的优势：

1. 丰富了自动变量和全局变量的初始化；
2. 使用初始化列表最大的优势是可以防止类型收窄（narrowing）。

### 结构化绑定

## 类型推导

C++11引入了auto和decltype这两个关键字实现了类型推导，让编译器来操心变量的类型。这使得C++也具有了和其他现代编程语言一样，某种意义上提供了无需操心变量类型的使用习惯。

### auto

在传统C++中，如果一个变量没有声明为register变量，将自动被视为一个auto变量。而随着register被弃用（在C++17中作为保留关键字，以后使用，目前不具备实际意义），对auto的语义变更也就非常自然了。

注：auto在类型很长或者类型很复杂的场景常用，其余场景不要滥用。

使用auto进行类型推导的一个最为常见而且显著的例子就是迭代器。



auto的优势：

1. auto推导最大的优势就是在拥有初始化表达式的复杂类型变量声明时简化代码；
2. 可以免除程序员在一些类型声明时的麻烦，或者避免一些在类型声明时的错误；
3. 其“自适应”性能够在一定程度上支持泛型的编程。

### decltype

decltype关键字是为了解决auto关键字只能对变量进行类型推导的缺陷而出现的，可以让编译器获取表达式的类型。

注：这类似于typeof的功能，但是typeof功能不是非常完善。

在C++11之前写法：

map<string,float>::value\_type elem; //不能自动由对象推导出类型

在C++11之后：

map<string,float> coll;

decltype(coll)::value\_type elem;

应用：

1. 声明return type；

template<typename T1,typename T2>

decltype(x+y) add(T1 x,T2 y);

这种写法在C++11前是不允许的，因为前面的返回表达式中x+y在编译的时候还不可见。在C++11中，你可以在函数后声明返回值参数：

template<typename T1,typename T2>

auto add(T1 x,T2 y)-> decltype(x+y);

1. 元编程（metaprogramming）
2. 传递Lambda表达式的类型

### 尾返回类型推导

### decltype(auto)

decltype(auto)主要用于对转发函数或封装的返回类型进行推导，它使我们无需显式的指定decltype的参数表达式。

## 控制流

### if constexpr

C++17将constexpr这个关键字引入到if语句中，允许在代码中声明常量表达式的判断条件。

### 区间for迭代range-based for statement

## 模板

### 外部模板

### 尖括号’>’

### 模板表达式空格space intemplate expressions

vector<list<int> >; //C++所有版本

vector<list<int>>; //C++11

### 类型别名模板alias template

别名就是模板的另一个名字，与#define和typedef效果不一样（#define只是简单替换，typedef不接收参数）。

定义：

template <typename T>

using Vec = std::vector<T,MyAlloc<T>>;

使用：

Vec<int> coll;

### 默认模板参数

### 变长参数模板variadic templates

数量不定的模板参数，可以传入任意个数和任意类型的参数，可以帮助我们做递归（不断将包中的参数拆分）。例如：

void print()//最后一次调用，什么都不做

{

}

template <typename T,typename… Types>

void print(const T& firstArg,const Types&…args)

{

cout<<firstArg<<endl;

print(args…);

}

可以下面的并存：

template < typename… Types>

void print(const Types&…args)

{

}

注：二者哪个泛化，哪个特化？调用的时候是否存在歧义？

参数列表中的省略号“...”表示多个参数，就是所谓的包（pack）：

1、用于template parameters，就是模板参数包（template parameters pack）

2、用于function parameter types，就是函数参数类型包（function parameter types pack）

3、用于function parameters，就是函数参数包（function parameters pack）

注：hash function和tuple实现中会用到

### 折叠表达式

### 非类型模板参数推导

## 面向对象

### 委托构造

C++11引入了委托构造的概念，这使得构造函数可以在同一个类中一个构造函数调用另一个构造函数，从而达到简化代码的目的。

### 继承构造

在传统C++中，构造函数如果需要继承是需要将参数一一传递的，这将导致效率低下。C++11利用关键字using引入了继承构造函数的概念。

### 异常noexception

### 显式虚函数重载override/final

override当重载虚函数时，引入override关键字将显式的告知编译器进行重载，编译器将检查基函数是否存在这样的虚函数，否则将无法通过编译。

final则是为了防止类被继续继承以及终止虚函数继续重载引入的。

### 默认函数控制

#### 类与默认函数

#### =default/=deleted

C++标准称“=default”修饰的函数为显式缺省（explicit defaulted）函数，而称“=delete”修饰的函数为删除（deleted）函数（可以应用在任何函数）。

一个类只要存在指针成员变量，几乎可以断定需要自己实现构造函数/拷贝函数/运算符重载（Big-Five），否则直接用默认的即可。

如果不允许拷贝，可以将类中拷贝相关的拷贝函数设置为delete（联系单例模式和深拷贝）。

注：拷贝构造函数不能有两个，因此不能给将拷贝构造函数设置=default，也不能设置为=delete（已经声明了又设置delete，这样编译器会报错）。运算符重载也不可以使用=default/=delete。

一般的函数没有默认版本，不可以使用=default，但是可以使用=delete（但是很少用）。

析构函数需要使用=default，如果设置=delete会造成使用已释放的对象。

为什么编译器需要给类默认构造函数？

构造函数除了执行初始化工作，主要的任务是执行一些幕后的工作，比如虚函数表，明确父子调用关系。

### 强类型枚举enum class

在传统C++中，枚举类型并非类型安全，枚举类型会被视作整数，则会让两种完全不同的枚举类型可以进行直接的比较（虽然编译器给出了检查，但并非所有），甚至同一个命名空间中的不同枚举类型的枚举值名字不能相同，这通常不是我们希望看到的结果。

C++11引入了枚举类（enumeration class），并使用enum class的语法进行声明。

## Lambda表达式

### 概述

C++11中的Lambda表达式用于定义并创建匿名的函数对象，以简化编程工作。

对于函数调用C++11之前操作：



C++11引入Lambda表达式之后：



### 参数

Lambda表达式的基本构成（匿名函数声明）：

[capture](parameters)mutable->return-type

{

statement

}

匿名函数调用：

[函数对象参数](操作符重载函数参数)mutable->返回值(函数体)

注：匿名函数声明和匿名函数调用的区别在于最后是否有()。

1. 函数对象参数：

[]，标识一个lambda表达式开始，这部分是必须存在的，不能省略。

函数对象参数是传递给编译器自动生成的函数对象类的构造函数的。函数对象只能使用那些到定义Lambda为止时Lambda所在作用范围内可见的局部变量（包括Lambda所在类的this）。函数对象参数有以下形式：

（1）空。没有使用任何函数对象参数。

（2）=。函数体内可以使用Lambda所在作用范围内所有可见的局部变量（包括Lambda所在类的this），并且是值传递方式（相当于编译器自动为我们的值传递了所有局部变量）。

（3）&。函数体内可以使用Lambda所在作用范围内所有可见的局部变量（包括Lambda所在类的this），并且是引用传递方式（相当于编译器自动为我们按引用传递了所有局部变量）。

（4）this。函数体内可以使用Lambda所在类中的成员变量。

（5）a。将a按值进行传递。按值传递时，函数体内不能修改传递进来的a的拷贝，因为默认情况下函数是const的。要修改传递进来的a的拷贝，可以添加mutable修饰符。

（6）&a。将a按引用进行传递。

（7）a,&b。将a按值进行传递，b按引用进行传递。

（8）=,&a,&b。除a和b按引用进行传递外，其他参数都按值进行传递。

注：推荐使用=，不推荐使用&。在Qt中，做信号-槽连接的时候，默认内部变量会进行锁状态，只读状态，如果进行写操作，就会core。

1. 操作符重载函数参数：

标识重载的()操作符的参数，没有参数时。这部分可以省略。参数可以通过按值（如：(a,b)）和按引用(如:(&a,&b))两种方式进行传递。

1. 可修改标识符：

mutable声明，这部分可以省略。按值传递函数对象参数时，加上mutable修饰符后，可以修改按值传递进来的拷贝（注意是能修改拷贝，而不是值本身）。

1. 函数返回值：

->返回值类型。标识函数返回值的类型，当返回值为void，或者函数体只有一处return的地方（此时编译器可以自动推断出返回值类型）时，这部分可以省略。

1. 函数体：

{}，标识函数的实现，这部分不能省略，但函数体可以为空。

### 使用

### Lambda与仿函数

### Lambda与STL

## 函数对象包装器

### function

### bind

## 右值引用rvalue references

### 拷贝构造函数

### 左值、右值和右值引用

右值引用是一种新的引用，可以解决不必要的拷贝（当赋值语句右边是右值rvalue时，左边的对象可以不用重新分配内存去steal右边的内存，避免重新分配内存）。

注：右值引用主要是为了move的实现服务，因为左值需要指向右值的地址（原来的指针失效，只有新的指针有效，有点类似深拷贝和浅拷贝结合，C++11之前容器insert原型为insert(…,&x)，C++11后为insert(…,&&x)）。

左值：可以出现在operator=左侧者（变量就是左值，函数是左值）

右值：只能出现在operator=右侧者（临时对象就是右值）

C++11之前不支持右值引用：



### std::move

std::move实际上并不能移动任何东西，它唯一的功能就是将一个左值强制转化为右值引用，继而我们可以通过右值引用使用该值，以用于移动语义。

从实现上将，std::move基本等同于一个类型转换：

static\_cast<T&&>(lvaue);

注：被转化的左值，其生命期并没有随着左右值的转化而改变，如果期望move转化的左值变量lvalue能够立即被析构，那么肯定会失望了。

### std::forward

多个嵌套调用的时候，需要做到完美转交（perfect forwarding）。

## 容器

### 头文件

头文件新特性：

1. C++标准库的header files不带副档名（.h），例如#include<vector>
2. 新式C header files不带副名称.h，例如#include<cstdio>
3. 旧式C header files（带有副名称.h）仍可用，例如#include<stdio.h>

### 线性容器

#### array

与std::vector不同，std::array对象的大小是固定的，如果容器大小是固定的，那么可以优先考虑使用std::array容器。

另外由于std::vector是自动扩容的，当存入大量的数据后，并且对容器进行了删除操作，容器并不会自动归还被删除元素相应的内存，这时候就需要手动运行shrink\_to\_fit()释放这部分内存。

使用std::array能够让代码变得更加“现代化”，而且封装了一些操作函数，比如获取数组大小以及检查是否非空，同时还能够友好的使用标准库中的容器算法，比如std::sort。

#### forward\_list

std::forward\_list是一个列表容器，使用方法和 std::list基本类似。

和std::list的双向链表的实现不同，std::forward\_list使用单向链表进行实现，提供了O(1) 复杂度的元素插入，不支持快速随机访问（这也是链表的特点），也是标准库容器中唯一一个不提供 size()方法的容器。当不需要双向迭代时，具有比std::list更高的空间利用率。

### 无序容器

#### hash function

哈希函数用来计算hash code，用于unordered\_set和unordered\_map等无序容器。

例如：

hash<int>()(123);

hash<long>()(123L);

hash<string>()(string(“abc”));

hash<const char\*>()(“abc”);

#### unordered containers

无序容器中的元素是不进行排序的，内部通过Hash表实现，插入和搜索元素的平均复杂度为 O(constant)，在不关心容器内部元素顺序时，能够获得显著的性能提升。

注：STL中的set和map是基于红黑树的，hash\_set和hash\_map是基于哈希表，unordered\_set和unordered\_map是基于哈希表。

C++11引入了两组无序容器：

std::unordered\_map/std::unordered\_multimap，std::unordered\_set/std::unordered\_multiset。

### tuple

纵观传统C++中的容器，除了std::pair外，似乎没有现成的结构能够用来存放不同类型的数据（通常我们会自己定义结构）。但std::pair的缺陷是显而易见的，只能保存两个元素。

基本操作：

1. std::make\_tuple: 构造元组

2. std::get: 获得元组某个位置的值

3. std::tie: 元组拆包

## 智能指针

### RAII

RAII是Resource Acquisition Is Initialization（“资源获取就是初始化”）的简称，是C++语言的一种管理资源、避免泄漏的惯用法。利用的就是C++构造的对象最终会被销毁的原则。RAII的做法是使用一个对象，在其构造时获取对应的资源，在对象生命期内控制对资源的访问，使之始终保持有效，最后在对象析构的时候，释放构造时获取的资源。

### shared\_ptr

std::shared\_ptr是一种智能指针，它能够记录多少个shared\_ptr共同指向一个对象，从而消除显示的调用delete，当引用计数变为零的时候就会将对象自动删除。

但还不够，因为使用std::shared\_ptr仍然需要使用new来调用，这使得代码出现了某种程度上的不对称。std::make\_shared 就能够用来消除显式的使用 new，所以std::make\_shared 会分配创建传入参数中的对象，并返回这个对象类型的std::shared\_ptr指针。

std::shared\_ptr可以通过get()方法来获取原始指针，通过reset()来减少一个引用计数，并通过use\_count()来查看一个对象的引用计数。

### unique\_ptr

std::unique\_ptr是一种独占的智能指针，它禁止其他智能指针与其共享同一个对象，从而保证代码的安全。

make\_unique并不复杂，C++11没有提供std::make\_unique。

既然是独占，换句话说就是不可复制。但是，我们可以利用std::move将其转移给其他的unique\_ptr。

### weak\_ptr

## 垃圾回收

### 分类

### C++垃圾回收

### 最小垃圾回收

## 正则表达式

### 概述

### regex

## 并发

std::thread用于创建一个执行的线程实例，所以它是一切并发编程的基础，使用时需要包含<thread>头文件，它提供了很多基本的线程操作，例如get\_id()来获取所创建线程的线程ID，使用join()来加入一个线程等等。

### 互斥量与临界区

std::mutex是C++11中最基本的mutex类，通过实例化std::mutex可以创建互斥量，而通过其成员函数lock()可以进行上锁，unlock()可以进行解锁。但是在在实际编写代码的过程中，最好不去直接调用成员函数，因为调用成员函数就需要在每个临界区的出口处调用 unlock()，当然，还包括异常。这时候C++11还为互斥量提供了一个RAII语法的模板类 std::lock\_gurad。RAII在不失代码简洁性的同时，很好的保证了代码的异常安全性。。

### 期物

期物（Future）表现为std::future，它提供了一个访问异步操作结果的途径。

### 条件变量

条件变量std::condition\_variable是为了解决死锁而生，当互斥操作不够用而引入的。比如，线程可能需要等待某个条件为真才能继续执行，而一个忙等待循环中可能会导致所有其他线程都无法进入临界区使得条件为真时，就会发生死锁。所以，condition\_variable实例被创建出现主要就是用于唤醒等待线程从而避免死锁。std::condition\_variabl的notify\_one()用于唤醒一个线程；notify\_all()则是通知所有线程。

### 内存模型

### 原子操作

### 总结