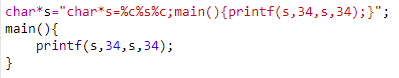
**---------\*小知识点\*----------**

**输出代码本身**

**法一：**



将换行符和引号等用对应的字符编码替换，巧妙地避免使用“\”引起死循环。用char 34来代替'\"'，endl代替\n,规避了转义字符的绕来绕去，为了打印出\，则需要\\，如此死循环肯定行不通，其他处理方式又需要额外逻辑判断。将程序分为2部分，一部分负责记录代码，另一部分运行时打印。

回车，ASCII码13  
换行，ASCII码10  
空格，ASCII码32

双引号，ASCII码34

**C版本：**



**C++版本：**

#include <iostream>

#include <string>

using namespace std;

int main()

{

char aDoubleQuot = 34;

string str[] = {

"#include <iostream>",

"#include <string>",

"using namespace std;",

"int main()",

"{",

" char a = 34;",

" string str[] = {",

" ",

" };",

" for (int i = 0; i <= 6;i++) cout << str[i] << endl;",

" for (int i = 0; i <= 13; i++) cout << str[7] + aDoubleQuot + str[i] + aDoubleQuot + ',' << endl;",

" for (int i = 8; i <= 13; i++) cout << str[i] << endl;",

" return 0",

"}"

};

for (int i = 0; i <= 6;i++) cout << str[i] << endl;

for (int i = 0; i <= 13; i++) cout << str[7] + aDoubleQuot + str[i] + aDoubleQuot + ',' << endl;

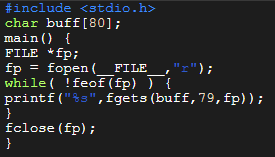
for (int i = 8; i <= 13; i++) cout << str[i] << endl;

return 0;

}

**法二：**

我们也可以换一种思路写Quine，我们可以利用编译器中的预定义宏“\_\_FILE\_\_”



**设计模式**



**创建型模式**

[单例模式](http://blog.csdn.net/liang19890820/article/details/61615495)（Singleton Pattern）   
保证一个类仅有一个实例，并提供一个访问它的全局访问点。

[抽象工厂模式](http://blog.csdn.net/liang19890820/article/details/70653800) （Abstract Factory Pattern）   
提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

[建造者模式](http://blog.csdn.net/liang19890820/article/details/66968761)（Builder Pattern）   
将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

[工厂方法模式](http://blog.csdn.net/liang19890820/article/details/70652858) （Factory Method Pattern）   
定义一个用于创建对象的接口，让子类决定将哪一个类实例化。Factory Method 使一个类的实例化延迟到其子类。

[原型模式](http://blog.csdn.net/liang19890820/article/details/66969965)（Prototype Pattern）   
用原型实例指定创建对象的种类，并且通过拷贝这个原型来创建新的对象。

## [单例模式](http://www.zkt.name/dan-li-mo-shi-singleton-ji-c-shi-xian/)

**作用：**

保证一个类只有一个实例，并提供一个访问它的全局访问点，使得系统中只有唯一的一个对象实例。单例类拥有一个私有构造函数，确保用户无法通过new关键字直接实例化它。除此之外，该模式中包含一个静态私有成员变量与静态公有的工厂方法，该工厂方法负责检验实例的存在性并实例化自己，然后存储在静态成员变量中，以确保只有一个实例被创建。

在单例模式的实现过程中，需要注意如下三点：

单例类的构造函数为私有；

提供一个自身的静态私有成员变量；

提供一个公有的静态工厂方法。

**优缺点：**

优点在于提供了对唯一实例的受控访问并可以节约系统资源；

其主要缺点在于因为缺少抽象层而难以扩展，且单例类职责过重。

**应用：**

常用于管理资源，如日志、线程池

**实现要点：**

在类中，要构造一个实例，就必须调用类的构造函数，并且为了保证全局只有一个实例，需防止在外部调用类的构造函数而构造实例，需要将构造函数的访问权限标记为private。构造函数声明成private的目的是只允许内部调用，getInstance()中的静态局部变量创建时调用，但不允许外部调用构造创建第二个实例；

同时阻止拷贝创建对象时赋值的拷贝对象，因此也将它们声明并权限标记为private。拷贝构造和拷贝赋值符是声明成了private而不给出定义，其目的是阻止拷贝，如果企图通过拷贝构造来创建第二个实例，编译器会报错。阻止拷贝的另一种写法是声明后接一个"=delete",也能起到相同的作用（C++11）；

另外，需要提供一个全局访问点，就需要在类中定义一个static函数，返回在类内部唯一构造的实例。



## 模板模式（方法）

**一、模板方法定义**

定义一个操作中的某种算法的框架，而将一些步骤延迟到子类中。模板方法模式使得子类在不改变一个算法结构的前提下，对这些步骤进行个性化定义。

**二、模板方法程序构成**

在模板方法设计模式中，存在一个父类，其中包括两类方法：模板方法和步骤方法。

**模板方法**，即实现某种算法的方法步骤。而这些步骤都是调用的步骤方法完成的。

**步骤方法**，即完成模板方法的每个阶段性方法。每个步骤方法完成某一特定的、完成总算法的一部分功能。

步骤方法有三种类型：抽象方法、最终方法和钩子方法。

**抽象方法**：要求子类必须实现的方法，是完成模板方法的算法步骤中必须由子类完成的个性化定义。（一个抽象方法由抽象类声明、由其具体子类实现。）

**最终方法**：子类不能重写的方法，是若要完成模板方法的算法步骤，对于所有子类执行都一样的步骤。

**钩子方法**，是父类给出默认实现，但子类也可以重写的方法。（个钩子方法由一个抽象类或具体类声明并实现，而其子类可能会加以扩展。通常在父类中给出的实现是一个空实现（可使用virtual关键字将其定义为虚函数），并以该空实现作为方法的默认实现，当然钩子方法也可以提供一个非空的默认实现。）

**优点**

1、封装不变部分/逻辑框架，扩展可变部分/具体细节

2、提取公共部分代码，便于维护

3、行为由父类控制，子类实现-->基本方法子类实现，因此子类可以通过扩展的方式增加相应的功能，符合开闭原则

**缺点**

1、因为模板方法将依赖点到，抽象中的框架实现依赖具体类的细节实现。所以是子类行为影响了父类。在复杂项目中，会造成代码阅读困难。

**使用场景**

多个子类有共有的方法，并且逻辑相同/逻辑相同，细节有差异

对重要，复杂的算法，可以讲核心算法设计为模板方法，周边细节由子类实现

重构时，经常使用的方案，将相同的代码抽象到父类，通过钩子函数约束行为。

**扩展**

①对于原定框架中的模板方法，如果存在子类中对某个接口的具有选择性（执行或者不执行），那么在调用时就需要具体判别这种差异性调用。

②对这种选择性，用过钩子，策略模式等方案。判别，调用。然后在子类中实现具体的钩子定义。

**狡辩**

父类调用子类的方法：

①把子类传递到父类的有残构造函数，然后调用

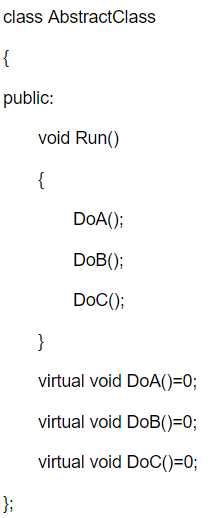
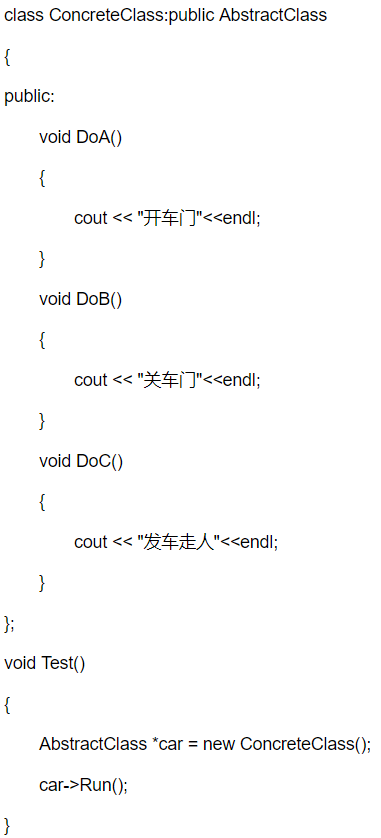
②使用反射的方式调用

③父类调用子类的静态方法

虽然这几种方式也实现了父类调用子类方法，但是在项目中并不能运用。子类简历框架，子类在重写了父类部分方法后，再调用从父类集成的方法，产生不同的结果。这正是模板方法--曲线救国。

**应用**

在开元框架中，模板方法提供了一个抽象类，然后开元框架写一堆子类。如果需要扩展功能，可以继承这个抽象类，然后覆写protected方法，然后调用一个execute（run）方法，就完成功能扩展。

**sizeof(i++)**

如果sizeof的操作数是一个表达式的话，这个表达式时不会被计算的。   
　　sizeof当预处理看就行了,它后面括号里的东西,根本不求值,只根据C的一堆规则判断结果类型,然后返回结果类型的大小。另外一个操作符typeid也是如此。

**Sizeof（构造/析构函数）=1**

调用构造函数和析构函数只需要知道函数的地址即可，而这些函数的地址只与类型有关，而与类型的实例无关，编译器也不会因为这两个函数而在实例中添加任何额外的信息。

**那如果把析构函数标记为虚函数呢？**  
C++的编译器一旦发现一个类型中有虚函数，就会为该类型生成虚函数表，并在该类型的每一个

实例中添加一个指向虚函数表的指针。在32位机器上，指针占用4个字节空间，所以sizeof得到4；如果在64位机器上，则一个指针占8字节的空间，因此求sizeof得带8。

**Git和SVN区别**

其实Git和SVN还是挺像的，都有提交，合并等操作，看来这是源码管理工具的基本操作。

1. Git是分布式的，SVN是集中式的，好处是跟其他同事不会有太多的冲突，自己写的代码放在自己电脑上，一段时间后再提交、合并，也可以不用联网在本地提交；

2. Git下载下来后，在本地不必联网就可以看到所有的log，很方便学习，SVN却需要联网；

3. Git鼓励分Branch，而SVN，说实话，我用Branch的次数还挺少的，SVN自带的Branch merge我还真没用过，有merge时用的是Beyond Compare工具合并后再Commit的；

4. Tortoise也有出Git版本，真是好东西；

5. SVN在Commit前，我们都建议是先Update一下，跟本地的代码编译没问题，并确保开发的功能正常后再提交，这样其实挺麻烦的，有好几次同事没有先Updata，就Commit了，发生了一些错误，耽误了大家时间，Git可能这种情况会少些。

**SVN优缺点**  
优点：   
1、 管理方便，逻辑明确，符合一般人思维习惯。   
2、 易于管理，集中式服务器更能保证安全性。   
3、 代码一致性非常高。   
4、 适合开发人数不多的项目开发。   
缺点：   
1、 服务器压力太大，数据库容量暴增。   
2、 如果不能连接到服务器上，基本上不可以工作，看上面第二步，如果服务器不能连接上，就不能提交，还原，对比等等。   
3、 不适合开源开发（开发人数非常非常多，但是Google app engine就是用svn的）。但是一般集中式管理的有非常明确的权限管理机制（例如分支访问限制），可以实现分层管理，从而很好的解决开发人数众多的问题。

**Git优缺点**  
优点：   
1、适合分布式开发，强调个体。   
2、公共服务器压力和数据量都不会太大。   
3、速度快、灵活。   
4、任意两个开发者之间可以很容易的解决冲突。   
5、离线工作。   
缺点：   
1、学习周期相对而言比较长。   
2、不符合常规思维。   
3、代码保密性差，一旦开发者把整个库克隆下来就可以完全公开所有代码和版本信息。

**大小端**

现在有一个数字 65430，这个数字在二进制的内存中一个字节无法完全存下，存储这个数字需要1个字节以上的空间。这样的值被称为多字节量（multi-byte quantity）。65430在内存中由两个字节表示：0xFF 和 0x96 ，其中：0xFF被称为最高有效字节（most significant byte, MSB）；0x96 被称为最低有效字节（last significant byte, LSB）

在内存中存储这样的多字节整数有两种方式

**大端模式**：  数据的高字节存在低地址  数据的低字节存在高地址

**小端模式**：  数据的高字节存在高地址  数据的低字节存在低地址

**大端**：最高有效字节存储在较低的内存位置

such as：

0x0->0xFF

0x1->0x96

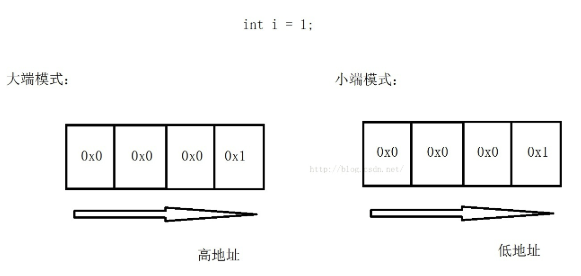
**小端**：最低有效字节存储在较低的内存位置

such as：

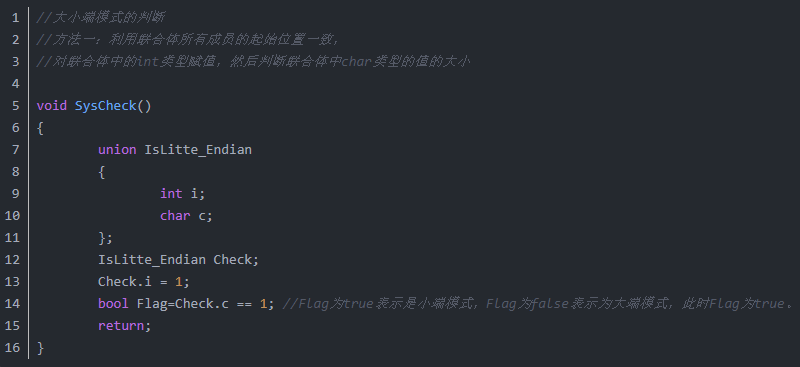
0x0->0x96

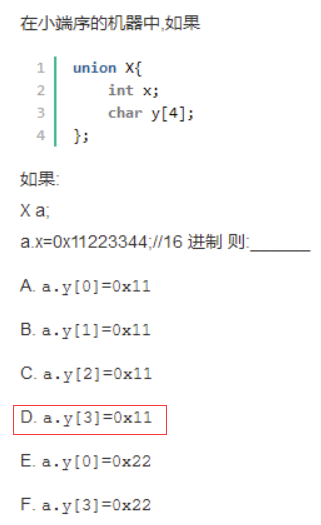
0x1->0xFF

**大小端的判断**



**如图，i为int类型占4个字节，但只有1个字节的值为1，另外3个字节值为0；取出低地址上的值，当其为1时则为小端模式，为0时为大端模式。**





我们都知道小端存储是数据低位存储在内存低位，大端存储是数据低位存储在内存高位。

0x11223344在内存中占据4byte，因此我们选择存储在y[4]中。

**按照大端存储，y[0]=11; y[1]=22; y[2]=33; y[3]=44;**

**按照小端序存储，y[0]=44; y[1]=33; y[2]=22; y[3]=11**



通过强制类型转换。程序中通过取Pvar变量的地址，获得起始空间的存储内容。如果起始空间存储的是数据的低位内容，则为little-endian，否则为big-endian

**左值右值**

a左：int a;  
b右：int &b = a;

在C++11中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在C++11中可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。

举个例子，int a = b+c, a 就是左值，其有变量名为a，通过&a可以获取该变量的地址；表达式b+c、函数int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，＆(b+c)这样的操作则不会通过编译。

**右值、将亡值**

在理解C++11的右值前，先看看C++98中右值的概念：C++98中右值是纯右值，纯右值指的是临时变量值、不跟对象关联的字面量值。临时变量指的是非引用返回的函数返回值、表达式等，例如函数int func()的返回值，表达式a+b；不跟对象关联的字面量值，例如true，2，”C”等。

C++11对C++98中的右值进行了扩充。在C++11中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。其中纯右值的概念等同于我们在C++98标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用T&&的函数返回值、std::move的返回值，或者转换为T&&的类型转换函数的返回值。

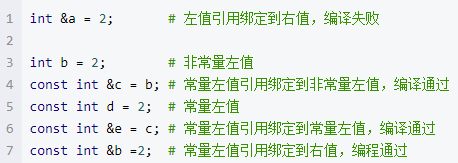
将亡值可以理解为通过“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量值的生命期。

**左值引用、右值引用**

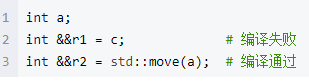
左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。

右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

左值引用通常也不能绑定到右值，但常量左值引用是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在它的“余生”中只能是只读的。相对地，非常量左值只能接受非常量左值对其进行初始化。



右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要std::move()将左值强制转换为右值，例如：



下表列出了在C++11中各种引用类型可以引用的值的类型。值得注意的是，只要能够绑定右值的引用类型，都能够延长右值的生命期。



**long和double不是原子操作**

long和double占用的字节数都是8，也就是64bits。在32位操作系统上对64位的数据的读写要分两步完成，每一步取32位数据。这样对double和long的赋值操作就会有问题：如果有两个线程同时写一个变量内存，一个进程写低32位，而另一个写高32位，这样将导致获取的64位数据是失效的数据。因此需要使用volatile关键字来防止此类现象。[volatile](#_volatile关键字)本身不保证获取和设置操作的原子性，仅仅保持修改的可见性。

但是java的内存模型保证声明为volatile的long和double变量的get和set操作是原子的。

**将一个整型的二进制最后三位清零。（x&=~7））**

**浮点数的大小比较为什么不能用等号？**

1、浮点数精度不同。

2、寄存器与内存表示浮点数精度不同。即使在精度相同的情况下，比较也可能会出问题。因为在运算过程中会将内存（或高速缓存）中的值加载到CPU浮点寄存器（80 bit扩展精度）中，然后再进入CPU浮点计算单元进行计算，计算结果写回浮点寄存器，然后写回内存（或高速缓存）。

从内存到浮点寄存器，浮点数的精度会扩展，从浮点寄存器到内存，浮点数的精度会降低精度扩展通常没问题，但如果精度降低了，很可能值会发生变化，出现截断，而浮点运算的结果由于下面还要使用所以暂时保存在浮点寄存器中留待下次使用（没有及时写回内存，这是一种优化策略），从而导致数据 并不是 内存中 和 内存中 的数据 比较 而是 浮点寄存器中的 值 和 内存中的值 进行比较，而无论内存中是float类型还是double类型，其精度和浮点寄存器精度都不相同，从而导致比较结果是不相等。

**C++崩溃原因**

1）内存越界

具体的代码形式可为数组下标越界和指针移动越界，在操作一个数组或是指针之前，要明确其边界范围，不应访问不属于系统分配的内存区域。

2）指针为空

即此时通过指针访问的内存地址为0，这个地址系统是不允许访问的，系统也不会分配地址为0的内存给程序，这属于一种特殊的内存越界行为。空指针常见于未初始化或初始化(赋值)失败的指针变量。

编译器一般会对未初始化的指针变量进行warning警告，所以在编译程序后不要忽略警告，这些警告有可能就是以后错误的根源；

比较隐蔽的一类是利用函数对指针变量进行初始化，但初始化失败了，如malloc函数，fopen函数，对于失败的情况这些函数都会返回0，若不加以判断，很可能在后续的使用中会出问题，当程序比较复杂时，这种错误很难排查，特别是出现了程序有时依然会正常运行偶尔会崩掉的怪现象的时候，所以在对指针进行赋值后，在不确定是否赋值成功时就应该及时判断。

|  |  |  |
| --- | --- | --- |
| **错误类型** | **具体表现** | **备注（案例）** |
| 声明错误 | 变量未声明 | 编译时错误 |
| 初始化错误 | 未初始化或初始化错误 | 运行不正确 |
| 访问错误 | 1、  数组索引访问越界  2、  指针对象访问越界  3、  访问空指针对象  4、  访问无效指针对象  5、  迭代器访问越界 |  |
| 内存泄漏 | 1、  内存未释放  2、  内存局部释放 |  |
| 参数错误 | 本地代理、空指针、强制转换 |  |
| 堆栈溢出 | 调用堆栈溢出：  1、递归调用  2、循环调用  3、消息循环  4、大对象参数  5、大对象变量 | 参数、局部变量都在栈(Stack)上分配 |
| 转换错误 | 有符号类型和无符号类型转换 |  |
| 内存碎片 | 小内存块重复分配释放导致的内存碎片，最后出现内存不足 | 数据对齐，机器字整数倍分配 |

**Java和C++的区别**

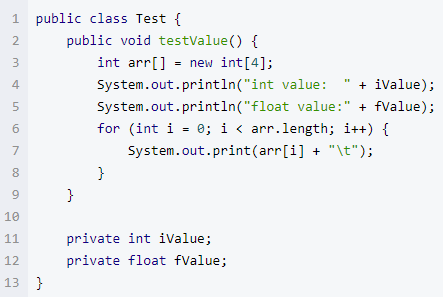
Java为解释性语言，其运行过程为：程序源代码经过Java编译器编译成字节码，然后由JVM解释执行。而C/C++为编译型语言，源代码经过编译和链接后生成可执行的二进制代码，可直接执行。因此Java的执行速度比C/C++慢。

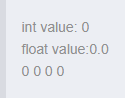
Java能够跨平台执行，C/C++不能。java 是运行在JVM上的，之所以说它的可移植性强，是因为jvm可以安装到任何的系统。c++不是不能在其他系统运行，而是c++在不同的系统上运行，需要不同的编码（这一点不如java，只编写一次代码，到处运行）。java程序一般都是生成字节码，在JVM里面运行得到结果。

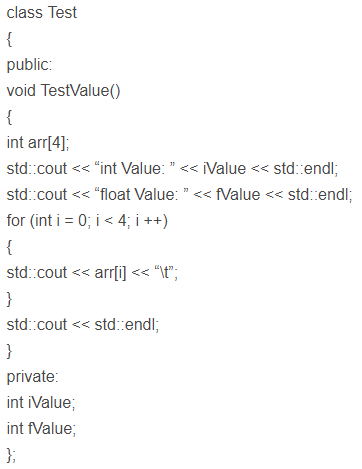
Java是纯面向对象语言，所有代码（包括函数、变量）必须在类中实现，除基本数据类型（包括int、float等）外，所有类型都是类。此外，Java语言中不存在全局变量或者全局函数，而C++兼具面向过程和面向对象编程的特点，可以定义全局变量和全局函数。

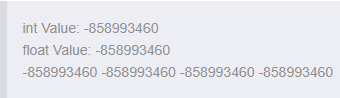
**1.定义变量和数组时，Java默认初始化，C++不初始化；**

在类中，方法中定义变量、动态数组时，Java默认初始化，C++不初始化。**因为变量未进行初始化，结果是一个任意的值，这将是非常危险的。因此C++的代码在定义变量后必须给其进行初始化。**





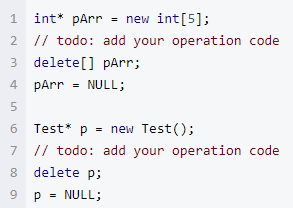




**2.Java中有垃圾回收机制，C++没有。**

在C++语言中，需要开发人员去管理内存的分配（包括申请和释放），而Java语言提供了垃圾回收器来实现垃圾的自动回收，不需要程序显示地管理内存的分配。在C++语言中，通常会把释放资源的代码放到析构函数中，Java语言中虽然没有析构函数，但却引入了一个finalize()方法，当垃圾回收器要释放无用对象的内存时，会首先调用该对象的finalize()方法，因此，开发人员不需要关心也不需要知道对象所占的内存空间何时被释放。

“垃圾收集”意味着在Java中出现内存漏洞的情况会少得多，但也并非完全不可能（若调用一个用于分配存储空间的固有方法，垃圾收集器就不能对其进行跟踪监视）。然而，内存漏洞和资源漏洞多是由于编写不当的finalize()造成的，或是由于在已分配的一个块尾释放一种资源造成的（“破坏器”在此时显得特别方便）。垃圾收集器是在C++基础上的一种极大进步，使许多编程问题消弥于无形之中。但对少数几个垃圾收集器力有不逮的问题，它却是不大适合的。但垃圾收集器的大量优点也使这一处缺点显得微不足道。



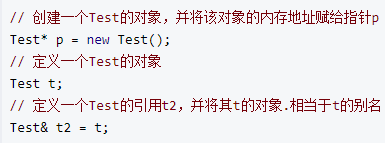
**3.Java中只有引用，没有指针；C++中有引用，也有指针。**

Java语言中没有指针的概念，引入了真正的数组。不同于C++中利用指针实现的“伪数组”，Java引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在c++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题。



new Test()创建了一个Test类的对象，t是指向这一对象的引用。

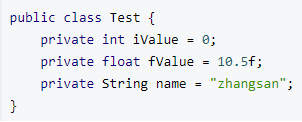
C++中有指针，也有引用：



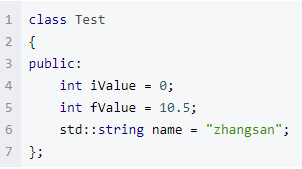
因为没有指针，Java代码的可读性好，更也安全；相较而言，C++就更难懂，但更灵活(特别是算法的一些操作时，指针的存在使数据和内存的操作变得极大的方便和灵活)。

**4.在定义类的成员时，Java可以赋初值，C++不可以**

如Java中，可以这样定义：



而C++中则不可以这样定义：



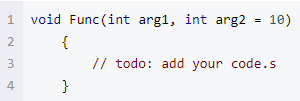
C++11之前都不能有上面这样的定义。编译时将报类似如下的错误：



但VS2015支持上面的定义，但VS2015好像已经支持C++14、C++17标准了。

**5.函数的参数，C++可以有默认形参，Java不可以**

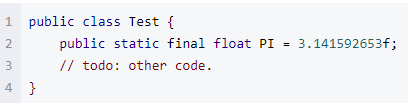
C++的函数可以有这样的定义：



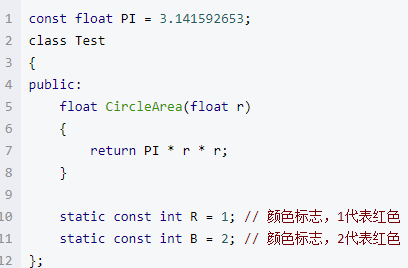
Java不支持这样定义

**6.Java中用final修饰不改变的成员，C++用const修饰不可改变的变量**

Java中可以有这样的定义：



C++中可以有这样的定义：



**C与C++的区别**

1、C是面向过程的语言，而C++是面向对象的语言。**请问什么是面向对象？**（封装继承多态）

2、C和C++动态管理内存的方法不一样，C是使用malloc/free函数，而C++除此之外还有new/delete关键字。**关于malooc/free与new/delete的不同？**

3、C中的struct和C++的类，C++的类是C所没有的，但是C中的struct是可以在C++中正常使用的，并且C++对struct进行了进一步的扩展，使struct在C++中可以和class一样当做类使用，而唯一和class不同的地方在于struct的成员默认访问修饰符是public,而class默认的是private;

4、C++支持函数重载，而C不支持函数重载，而C++支持重载的依仗就在于C++的名字修饰与C不同，例如在C++中函数int fun(int ,int)经过名字修饰之后变为 \_fun\_int\_int ,而C是\_fun，一般是这样的，所以C++才会支持不同的参数调用不同的函数；

5、C++中有引用，而C没有。[**引用和指针的区别**](#_------*指针引用/数组链表队列*------)**？**

6、C++全部变量的默认链接属性是外链接，而C是内连接；

7、C 中用const修饰的变量不可以用在定义数组时的大小，但是C++用const修饰的变量可以（如果不进行&解引用的操作的话，是存放在符号表的，不开辟内存）；

8、局部变量的声明规则不同

9、多态

10、C++特有输入输出流。

**面向过程与面向对象的区别**

面向过程编程的设计思路：步骤化分析问题

面向对象编程的设计思路：功能化分析问题

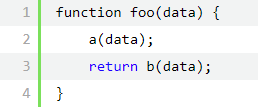
1.数据和方法（方法可以理解为函数）被封装在一起，这样做改动的时候对整个程序的影响不大。   
2.通过继承减少代码的冗余。   
3.易扩展

**\*大规模对象分配，生命周期控制的难点（并发情况下）?**

智能指针控制生命周期，小对象可以参考java的String直接搞成只读的，配合copy on write 和原子操作尽量把临界区弄小。在读的情况下把锁优化掉以及减小临界区

**递归和尾递归（+尾调用）**

**尾调用：**尾调用是指一个函数里的最后一个动作是一个函数调用的情形：尾调用由于是函数的最后一步操作，所以不需要保留外层函数的调用记录，因为调用位置、内部变量等信息都不会再用到了。即这个调用的返回值直接被当前函数返回的情形。这种情形下该调用位置为尾位置。尾调用很重要的特性就是它可以不在调用栈上面添加一个新的堆栈帧，而是更新它。



求递归中，也有一定的缺点，假如说求1000!的阶乘，会出现栈溢出的问题，因为在函数执行中，每调用一个函数都会把当前函数的调用位置和内部变量保存在栈里面，由于栈的空间不是无限大，假如说调用层数过多，就是出现栈溢出的情况。这个时候就可以用尾递归优化来解决。

尾调用的概念非常简单，一句话就能说清楚，就是指某个函数的最后一步是调用另一个函数。所以尾递归优化可以有效的**防止栈溢出**，但是尾递归优化需要编译器或者解释器的支持，遗憾的是，大多数编程语言没有针对尾递归做优化，Python解释器也没有做优化，所以，即使把上面的fact(n)函数改成尾递归方式，也会导致栈溢出。

**尾递归：**若一个函数在尾位置调用本身（或是一个尾调用本身的其他函数等），则称这种情况为尾递归，是递归的一种特殊情形。而形式上只要是最后一个return语句返回的是一个完整函数，它就是尾递归。这里注意：尾调用不一定是递归调用，但是尾递归一定是尾调用。

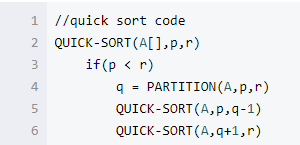
**为什么尾递归快？**

1、尾递归就是从最后开始计算, 每递归一次就算出相应的结果, 也就是说, 函数调用出现在调用者函数的尾部, 因为是尾部, 所以根本没有必要去保存任何局部变量. 直接让被调用的函数返回时越过调用者, 返回到调用者的调用者去.

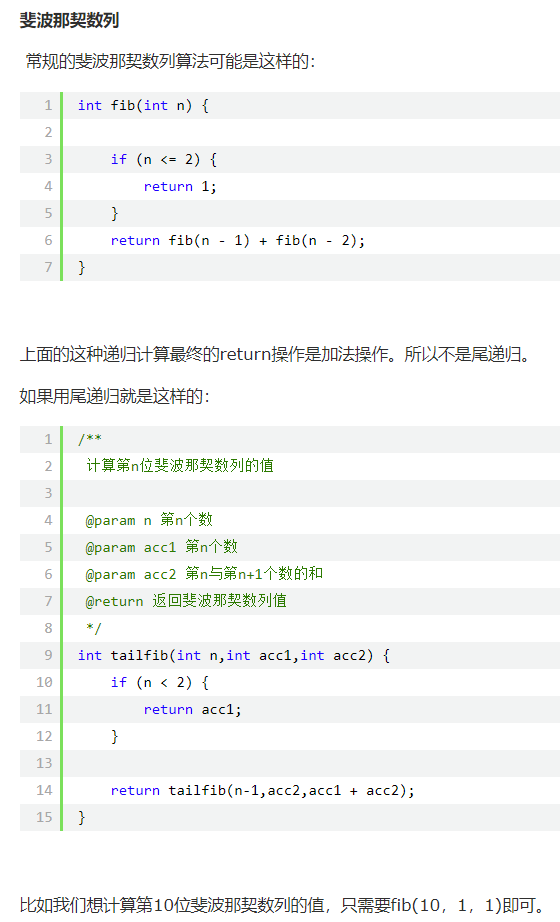
2、递归解题相对常用的算法如普通循环等，运行效率较低。因此，应该尽量避免使用递归，除非没有更好的算法或者某种特定情况，递归更为适合的时候。在递归调用的过程当中系统为每一层的返回点、局部量等开辟了栈来存储，因此递归次数过多容易造成栈溢出。

3、函数调用出现在调用者函数的尾部, 因为是尾部, 所以根本没有必要去保存任何局部变量. 直接让被调用的函数返回时越过调用者, 返回到调用者的调用者去。尾递归就是把当前的运算结果（或路径）放在参数里传给下层函数，深层函数所面对的不是越来越简单的问题，而是越来越复杂的问题，因为参数里带有前面若干步的运算路径。

快排的尾递归优化：



由上面可看，quick-sort在 在编译时，能够进行 尾递归 优化，减少运行栈深度，提升效率和速度。



**64位x86操作系统和32位x86操作系统函数调用的差别？**

64位前6个参数放寄存器，32位是直接压栈

**32bit与64bit有什么区别**

所谓32位和64位，指的是CPU的字长，其实主要是 GPRS(General Purpose Regisers,通用寄存器)的数据宽度。电脑内部都是实行2进制运算，处理器（CPU）一次处理数据的能力也是2的倍数。8位处理器、16位处理器、32位处理器和64位处理器，其计数都是2的倍数。一次处理的数据越大，该电脑处理信息的能力越来越大。**所谓32位处理器就是一次只能处理32位，也就是4个字节的数据，而64位处理器一次就能处理64位，即8个字节的数据。**

CPU从32位提升到64位，主要的变化如下：  
1）是处理能力的提升。比如原来要两个周期做的事，可以在一个周期搞定。  
2）64bit的CPU有更大的寻址能力。原本32位CPU32根地址线只可以寻址2^32（4GB）的内存（所以，如果我们装32位系统，搞8G的内存实际上是没有用的）。而64位CPU理论上的寻址能力是2^64，这个相当相当的大，目前的硬件还达不到这个水平，当然我们更用不到这么大的内存。  
3）数据的精度提升。一些数据类型可以用更大的字长表示，可以表示更大范围或者更精确的数。CPU分为32位和64位。在这基础上就有了32位操作系统和64位操作系统。进而还有32位程序和64位程序。

**32位编译器：**

char ：1个字节  
char\*（即指针变量）: 4个字节（32位的寻址空间是2^32, 即32个bit，也就是4个字节。同理64位编译器）  
short int : 2个字节  
int：  4个字节  
unsigned int : 4个字节  
float:  4个字节  
double:   8个字节  
long:   4个字节  
long long:  8个字节  
unsigned long:  4个字节

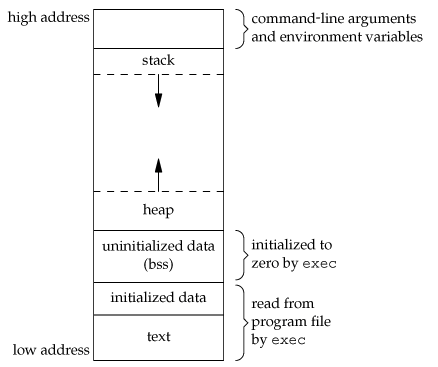
**64位编译器：**

    char ：1个字节  
      char\*(即指针变量): 8个字节  
      short int : 2个字节  
      int：  4个字节  
      unsigned int : 4个字节  
      float:  4个字节  
      double:   8个字节  
      long:   8个字节  
      long long:  8个字节  
      unsigned long:  8个字节

**---------\*内存/编译/运行\*----------**

**C++内存布局**

在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区



1.栈，就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的存储区。里面的变量通常是**局部变量、函数参数**等。存放在栈中的数据只在当前函数及下一层函数中有效，一旦函数返回了，这些数据也就自动释放了。

2.堆，就是那些由**new**分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个new就要对应一个delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。涉及的问题：“缓冲区溢出”、“内存泄露”

3.自由存储区，就是那些由malloc等分配的内存块，他和堆是十分相似的，不过它是用free来结束自己的生命的。

4.全局/静态存储区（.bss段和.data段），**全局变量和静态变量**被分配到同一块内存中，在以前的C语言中，未初始化的放在.bss段中，初始化的放在.data段，在C++里面没有这个区分了，他们共同占用同一块内存区。

5.常量存储区（.rodata段），这是一块比较特殊的存储区，他们里面存放的是**常量**，常量字符串就是放在这里的，不允许修改(当然，你要通过非正当手段也可以修改)

**C的存储区**

1.栈 - 由编译器自动分配释放，在函数体中定义的变量通常是在栈上

2.堆 - 一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。用malloc， calloc， realloc等分配内存的函数分配得到的就是在堆上。

3.全局区(静态区)，在所有函数体外定义的是全局量，加了static修饰符后不管在哪里都存放在全局区(静态区)，在所有函数体外定义的static变量表示在该文件中有效，不能extern到别的文件用，在函数体内定义的static表示只在该函数体内有效。。全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。- 程序结束释放

4.另外还有一个专门放常量的地方。函数中的”adgfdf”这样的字符串存放在常量区- 程序结束释放

5 程序代码区，存放2进制代码。

另外，根据数据在内存中存在的时间（生存周期）不同，将内存空间分为三个区：

1）程序区：用于存储程序的代码，即程序的二进制代码。

2）静态存储区：用于存储全局变量和静态变量，这些变量的空间在程序编译时就已经分配好了。

3）动态存储区：用于在程序执行时分配的内存，又分为：堆区（heap）和栈区(stack)。堆区：用于动态内存分配，程序运行时由内存分配函数在堆上分配内存。在C语言中，只能使用指针才能动态的分配内存。栈区:在函数执行时，函数内部的局部变量和函数参数的存储单元的内存区域，函数运行结束时，这些内存区域会自动释放。

内存分配方式有三种：

[1]从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。

[2]在栈上创建。在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

[3]从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。动态内存的生存期由程序员决定，使用非常灵活，但如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，频繁地分配和释放不同大小的堆空间将会产生堆内碎块。

**堆和栈的区别**

栈区（stack）由编译器自动分配释放，存放为运行函数而分配的局部变量、函数参数、返回数据、返回地址等。其操作方式类似于数据结构中的栈。

具体地说，现代计算机(串行执行机制)，都直接在代码底层支持栈的数据结构。这体现在，有专门的寄存器指向栈所在的地址，有专门的机器指令完成数据入栈出栈的操作。这种机制的特点是效率高，支持的数据有限，一般是整数，指针，浮点数等系统直接支持的数据类型，并不直接支持其他的数据结构。因为栈的这种特点，对栈的使用在程序中是非常频繁的。对子程序的调用就是直接利用栈完成的。机器的call指令里隐含了把返回地址推入栈，然后跳转至子程序地址的操作，而子程序中的ret指令则隐含从堆栈中弹出返回地址并跳转之的操作。C/C++中的自动变量是直接利用栈的例子，这也就是为什么当函数返回时，该函数的自动变量自动失效的原因。

堆区（heap）一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。分配方式类似于链表。

和栈不同，堆的数据结构并不是由系统(无论是机器系统还是操作系统)支持的，而是由函数库提供的。基本的malloc/realloc/free 函数维护了一套内部的堆数据结构。当程序使用这些函数去获得新的内存空间时，这套函数首先试图从内部堆中寻找可用的内存空间，如果没有可以使用的内存空间，则试图利用系统调用来动态增加程序数据段的内存大小，新分配得到的空间首先被组织进内部堆中去，然后再以适当的形式返回给调用者。当程序释放分配的内存空间时，这片内存空间被返回内部堆结构中，可能会被适当的处理(比如和其他空闲空间合并成更大的空闲空间)，以更适合下一次内存分配申请。这套复杂的分配机制实际上相当于一个内存分配的缓冲池(Cache)，使用这套机制有如下若干原因：

1、系统调用可能不支持任意大小的内存分配。有些系统的系统调用只支持固定大小及其倍数的内存请求(按页分配);这样的话对于大量的小内存分类来说会造成浪费。

2、系统调用申请内存可能是代价昂贵的。系统调用可能涉及用户态和核心态的转换。

3、没有管理的内存分配在大量复杂内存的分配释放操作下很容易造成内存碎片。

**0申请方式和回收方式不同**

栈（英文名称是stack）是系统自动分配空间的，例如我们定义一个 char a；系统会自动在栈上为其开辟空间。由于栈上的空间是自动分配自动回收的，所以栈上的数据的生存周期只是在函数的运行过程中，运行后就释放掉，不可以再访问。

对于栈来讲，则不会存在碎片问题，因为栈是先进后出的队列，他们是如此的一一对应，以至于永远都不可能有一个内存块从栈中间弹出，在他弹出之前，在他上面的后进的栈内容已经被弹出。

堆（英文名称是heap）则是程序员根据需要自己申请的空间，例如malloc（10）；开辟十个字节的空间。堆上的数据只要程序员不释放空间，就一直可以访问到，不过缺点是一旦忘记释放会造成内存泄露。

由于大量new/delete的使用，容易造成大量的内存碎片;由于没有专门的系统支持，效率很低;由于可能引发用户态和核心态的切换，内存的申请，代价变得更加昂贵。

**1.申请后系统的响应**

栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

堆：操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。也就是说堆会在申请后还要做一些后续的工作这就会引出申请效率的问题。

**2.申请效率的比较**

栈：由系统自动分配，速度较快。但程序员是无法控制的。栈是机器系统提供的数据结构，计算机会在底层对栈提供支持：分配专门的寄存器存放栈的地址，压栈出栈都有专门的指令执行，这就决定了栈的效率比较高。

堆：是由new或malloc分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便。堆则是C/C++函数库提供的，它的机制是很复杂的，例如为了分配一块内存，库函数会按照一定的算法(具体的算法可以参考数据结构/操作系统)在堆内存中搜索可用的足够大小的空间，如果没有足够大小的空间(可能是由于内存碎片太多)，就有可能调用系统功能去增加程序数据段的内存空间，这样就有机会分到足够大小的内存，然后进行返回。显然，堆的效率比栈要低得多。

**3.申请大小的限制**

栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在 WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

**4.堆和栈中的存储内容**

栈： 在函数调用时，第一个进栈的是主函数中函数调用后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈的，然后是函数中的局部变量。注意静态变量是不入栈的。 当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。   
 堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容有程序员安排。

**5.存取效率的比较**

char s1[] = "aaaaaaaaaaaaaaa";

char \*s2 = "bbbbbbbbbbbbbbbbb";

aaaaaaaaaaa是在运行时刻赋值的；放在栈中。

而bbbbbbbbbbb是在编译时就确定的；放在堆中。

但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。

**常量存放位置**

对于局部对象，常量存放在栈区；

对于全局对象，常量存放在全局/静态存储区；

对于字面值常量，存放在常量存储区。

**字面值常量和字面值类型**

一个形如42的值被称为字面值常量。字面值常量的形式和值决定了常量的类型。

例如0x42是16进制表示的整型常量。‘a'是char型字面值。字面值常量顾名思义由字面意思表示，是常量。字面值常量在程序中是直接表示的，整型直接写出大小，字符直接写出字符。一个字面值常量在编译时被直接解析为立即数，编译器内部维护字面值常量的类型。

字面值的类型根据字面值表达形式不同而不同，编译器根据字面值形式推断字面值的类型。字面值多数为算术类型。自定义类、IO类不属于该类型。

对于指针字面值，其只有nullptr和NULL。

**常量表达式**

是指在编译和执行过程中，该表达式的值不会改变，且编译过程中可以立即得到其值的表达式。一部分const对象是常量表达式，由常量表达式初始化的const对象也是常量表达式。常量表达式在程序运行时不会改变，即使一个程序多次启动或外部参数发生变化，该值也不会改变。编译器在编译优化时可能把常量表达式直接替换为立即数，具体要看编译环境。一般来讲，字面值常量属于常量表达式。

并非所有const对象都是常量表达式，const仅标记对象为只读属性，该对象在初始化后无法再改变。如果const对象所赋初值在编译阶段就可确定，那么此const对象才是常量表达式。const对象和存储位置也没有必然联系，常量可以分布在栈、堆、静态存储区中。对于声明在函数体内的const常量，如果没有被编译优化掉，该常量存储在栈中。全局的const常量存储在全局存储区。

C++中允许将变量声明为constexpr类型以使编译器在编译时检查该变量是否是常量表达式。声明为constexpr的对象一定是常量表达式。且初始化必须用常量表达式。

例如：

const int a = 12;    //a是常量表达式

const int b = a+1;  //b也是常量表达式

constexpr int c = a+b; //编译器可以在编译时期可以确定其值

const int d = getsize();  //c不是常量表达式，编译器编译时无法确知getsize()的执行结果。

constexpr int e = getsize();  //error! 将会报错

指针也可以是constexpr的常量表达式，表示该指针为const且指向在编译时确定。不是所有的顶层const指针都是constexpr的。只有那些在编译时就确定地址指向的const指针才是constexpr。因此nullptr、NULL、指向固定地址的指针是constexpr的。

对于显式指出的字符串常量，其存储在常量存储区中，例如：

const char \*const str1 = "hello"; //

const char \*const str2 = "hello"; //“hello"存储在常量存储区

constexpr const char \*str = str1; //constexpr,编译时确定的指针str

assert(str1 == str2); //str1和str2指向相同地址

此时的str1和str2指针都是constexpr的。那么，如何判断指针是否指向固定地址呢？

程序在内存中的组织形式是段，有堆栈段、数据段和代码段。对于数据指针指向数据，函数指针则指向某个代码。



对于定义在函数体外的变量，其指针是constexpr的；此类变量要占用数据段，而程序运行时，代码段和数据段大小位置均不会改变，因此编译器可以确定地址指向，是constexpr的。此外，函数内部定义的static静态变量，也会在数据区使用固定地址，因此指针也是constexpr的。对于定义在函数内部的变量，由于要在栈中开辟内存空间，而栈的情况要看程序运行状态，因此这类变量没有确定的地址，其指针不是constexpr的。代码段不会改变，函数指针也是constexpr的。

const/constexpr常量表达式可能在编译时直接替换为立即数；但也可能被编译为const对象，程序运行时占用内存空间。const/constexpr是否被替换为立即数或生成对象取决于编译环境和语义解析，语义不允许替换的地方便生成const对象，就如inline函数一样，不是所有inline函数都被替换。

**内存泄漏**

通常我们所说的内存泄漏，是指分配出去的内存在使用之后没有释放掉，没有回收，长此以往，会造成没有足够的内存可以分配。一般表现为运行时间越长，占用的内存越多，最终导致系统奔溃。

一般的内存泄漏是指**堆**内存的泄漏。堆内存是指程序从堆中分配的，大小任意的（内存块的大小可以在程序运行期决定），使用完后必须显式释放的内存。应用程序一般使用malloc，realloc，new等函数从堆中分配到一块内存，使用完后，程序必须负责相应的调用free或delete释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了。

一般来说内存泄漏有两种情况。一种情况如在C/C++ 语言中的，在堆中的分配的内存，在没有将其释放掉的时候，就将所有能访问这块内存的方式都删掉（如指针重新赋值）；另一种情况则是在内存对象明明已经不需要的时候，还仍然保留着这块内存和它的访问方式（引用）。

**如何避免：**

1. 是在声明对象引用之前，明确内存对象的有效作用域。在一个函数内有效的内存对象，

应该声明为 local 变量，与类实例生命周期相同的要声明为实例变量……以此类推。

1. 在内存对象不再需要时，记得手动将其引用置空。

**内存泄漏和内存溢出的区别和联系**

1、**内存泄漏**memory leak :是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄漏似乎不会有大的影响，但内存泄漏堆积后的后果就是内存溢出。   
2、**内存溢出** out of memory :指程序申请内存时，没有足够的内存供申请者使用，或者说，给了你一块存储int类型数据的存储空间，但是你却存储long类型的数据，那么结果就是内存不够用，此时就会报错OOM,即所谓的内存溢出。

3、**二者的关系**

内存泄漏的堆积最终会导致内存溢出；内存溢出就是你要的内存空间超过了系统实际分配给你的空间，此时系统相当于没法满足你的需求，就会报内存溢出的错误；内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就是分配的内存不足以放下数据项序列,称为内存溢出。

4、**内存泄漏的分类**（按发生方式来分类）

常发性内存泄漏。发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。

偶发性内存泄漏。发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。

一次性内存泄漏。发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。

隐式内存泄漏。程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

5、**内存溢出的原因及解决方法**：

内存溢出原因：   
1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；   
2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；   
3.代码中存在死循环或循环产生过多重复的对象实体；   
4.使用的第三方软件中的BUG；   
5.启动参数内存值设定的过小

内存溢出的解决方案：   
第一步，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)

第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

第三步，对代码进行走查和分析，找出可能发生内存溢出的位置。

重点排查以下几点：   
1.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

2.检查代码中是否有死循环或递归调用。

3.检查是否有大循环重复产生新对象实体。

4.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

5.检查**List**、**MAP**等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

第四步，使用内存查看工具动态查看内存使用情况

**避免内存碎片：开辟内存池**

频繁的申请和释放小块内存时，如何才能高效一点呢？所以就有了内存池的概念

从内存分配的概念上看，内存申请无非就是向内存分配方索要一个指针，当向操作系统申请内存时，操作系统需要进行复杂的内存管理调度之后，才能正确的分配出一个相应的指针。而这个分配的过程中，我们还面临着分配失败的风险。

所以，每一次进行内存分配，就会消耗一次分配内存的时间，设这个时间为 T，那么进行 n 次分配总共消耗的时间就是 nT；如果我们一开始就确定好我们可能需要多少内存，那么在最初的时候就分配好这样的一块内存区域，当我们需要内存的时候，直接从这块已经分配好的内存中使用即可，那么总共需要的分配时间仅仅只有 T。当 n 越大时，节约的时间就越多。

内存池的优点：

（1）针对特殊情况，例如需要频繁分配释放固定大小的内存对象时，不需要复杂的分配算法和多线程保护。也不需要维护内存空闲表的额外开销，从而获得较高的性能。  
（2）由于开辟一定数量的连续内存空间作为内存池块，因而一定程度上提高了程序局部性，提升了程序性能。  
（3）比较容易控制页边界对齐和内存字节对齐，没有内存碎片的问题。  
（4）当需要分配管理的内存在100M一下的时候，采用内存池会节省大量的时间，否则会耗费更多的时间。  
（5）内存池可以防止更多的内存碎片的产生

（6）更方便于管理内存

1.内存的分配：

（1）如果分配大小超过1024，直接采用malloc分配，分配的时候多分配sizeof(size\_t)字节，用于保存该块的大小；

（2）否则根据分配大小，查找到容纳该大小的最小size的MemoryChunk；

（3）查找MemoryChunk的链表指针pList，找到空闲的MemoryBlock返回；

（4）如果pList为NULL，临时创建MemoryBlock返回；

（5）MemoryBlock头部包含两个成员，pChunk指向的所属的MemoryChunk对象，size表明大小，其后才是给用户使用的空间；

2.内存的释放：

（1）根据释放的指针，查找器size头部，即减去sizeof(size\_t)字节，判断该块的大小；

（2）如果大小超过1024，直接free;

（3）否则交给MemoryChunk处理，而块的头部保存了该指针，因此直接利用该指针就可以收回该内存。

注意的问题：

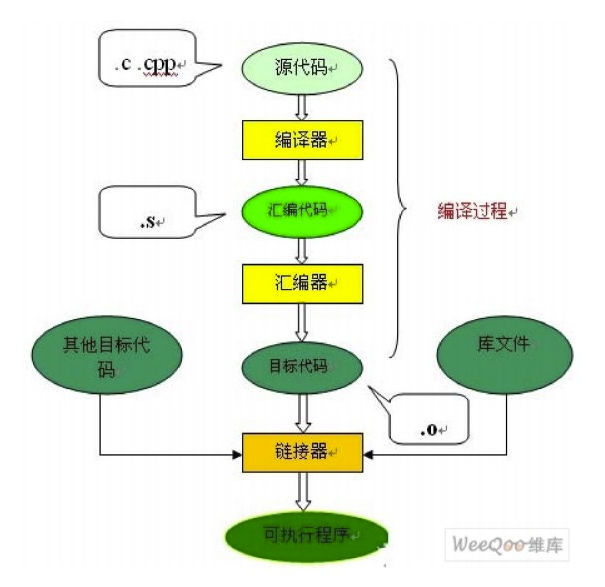
上述设计的内存池通过冗余的头部来实现内存块的分配与释放，减少了内存池的操作时间，速度上要优于原始的malloc和free操作，同时减少了内存碎片的增加。但是该设计中没有去验证释放的块冗余头部的正确性，因此故意释放不属于内存池中的块或者修改头部信息都会导致内存池操作失败，当然这些可以由程序员来控制。

此外，内存池中分配出去的内存块如果不主动释放，内存池没有保留信息，不会自动释放，但是在退出的时候会验证验证是否完全释放，其实这个在系统测试时候就可以检测出来，我想这个缺陷也是可以弥补的，在此提出，希望使用者注意。

**C++编译过程**[**（预处理/编译/汇编/链接）**](https://www.cnblogs.com/Lynn-Zhang/p/5377024.html)

## 类的静态数据成员不能就地初始化

因为类体一般是放在头文件中的，如果允许其静态成员就地初始化，那就相当于允许在头文件中定义变量了。



**编译期**是指把你的源程序交给编译器**编译的过程**，最终目的是得到obj文件，链接后生成可执行文件（预处理、编译、汇编和连接）。

2.**编译期分配内存，就是用静态或全局数组。**这是在编译的时候确定的。

**1.预处理 .c -> .i**

预处理相当于根据预处理指令组装新的C/C++程序。经过预处理，会产生一个没有宏定义，没有条件编译指令，没有特殊符号的输出文件，这个文件的含义同原本的文件无异，只是内容上有所不同。

**读取C/C++源程序，对其中的伪指令（以#开头的指令）进行处理**

①将所有的“#define”删除，并且展开所有的宏定义

②处理所有的条件编译指令，如：“#if”、“#ifdef”、“#elif”、“#else”、“endif”等。这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。预编译程序将根据有关的文件，将那些不必要的代码过滤掉。

③处理“#include”预编译指令，将被包含的文件插入到该预编译指令的位置。这个过程可能是递归进行的，也就是说被包含的文件可能还包含其他文件。

**删除所有的注释**

**添加行号和文件名标识。**

以便于编译时编译器产生调试用的行号信息及用于编译时产生的编译错误或警告时能够显示行号

**保留所有的#pragma编译器指令**

**2.编译 .c/.h -> .s**

通过预编译输出的文件中，只有常量：数字、字符串、变量的定义，以及c语言的关键字：main、if、else、for、while等。这阶段要做的工作主要是，通过语法分析，词法分析，语义分析及优化后，确定所有指令是否符合规则，之后翻译成汇编代码。

**3.汇编 .s -> .o**

将编译完的汇编代码文件翻译成机器指令，并生成可重定位目标程序的.o文件，该文件为二进制文件，字节编码是机器指令。目标文件中存放的也就是与源程序等效的目标的机器语言代码。

汇编器是将汇编代码转变成机器可以执行的指令，每一个汇编语句几乎都对应一条机器指令。所以汇编器的汇编过程相对于编译器来讲比较简单，它没有复杂的语法，也没有语义，也不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译即可。

目标文件由段组成，通常至少又两个段：

1、代码段：包换主要程序的指令。该段是可读和可执行的，一般不可写

2、数据段：存放程序用到的全局变量或静态数据。可读、可写、可执行。

**.o文件至少要提供3张表**

1、导出符号表：即该目标文件可以提供的符号及地址

2、未解决符号表：即找不到地址的符号的列表，告诉链接器这些符号没找到地址

3、地址重定向表：

链接的时候，链接器会为目标文件的“未解决符号表”里的符号在其他目标文件中寻找地址，但是每个目标文件的地址都是从0x0000开始的，这样直接将对方文件中符号的地址拿过来用显然会是不正确的，为了区分不同的文件，链接器在链接时就会对每个目标文件的地址进行调整：假如B.obj的0x0000被定位到可执行文件的0x00001000上，而A.obj的0x0000被定位到可执行文件的0x00002000上，那么实现上对链接器来说，A.obj的导出符号地地址都会加上0x00002000，B.obj所有的符号地址也会加上0x00001000。这样就可以保证地址不会重复。

因为被加上了起始地址，所以符号在自身文件中的实际地址就不对了，需要再用一张地址重定向表记录符号相对自身文件的地址

**4.链接**

通过链接器将一个个目标文件（或许还会有库文件）链接在一起生成一个完整的可执行程序。

由汇编程序生成的目标文件并不能立即就执行，还要通过链接过程。

原因：

1、某个源文件调用了另一个源文件中的函数或常量

2、在程序中调用了某个库文件中的函数

例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。

**链接过程**

链接程序的主要工作就是将有关的目标文件（库文件、.o文件）彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够被操作系统装入执行的统一整体。

**具体工作**：

当链接器进行链接的时候，首先决定各个目标文件在最终可执行文件里的位置。然后访问所有目标文件的地址重定义表，对其中记录的地址进行重定向（加上一个偏移量，即该编译单元在可执行文件上的起始地址）。然后遍历所有目标文件的未解决符号表，并且在所有的导出符号表里查找匹配的符号，并在未解决符号表中所记录的位置上填写实现地址。最后把所有的目标文件的内容写在各自的位置上，再作一些另的工作，就生成一个可执行文件。

## 外部链接和内部链接

**对于函数和变量，默认链接是外部链接**

如果函数默认是内部链接，那么大家会倾向于把函数连同其定义都放入头文件中。然而，函数是多变的，可能会经常修改，这样一来，所以包含它的模块都需要被重新编译，很麻烦。另外一方面，如果函数中定义了静态变量，这样每一个包含该函数的模块都会有一个静态变量（因为假设是默认内部链接），导致不一致。

**extern：**

这就是告诉编译器，这个变量或函数在别的编译单元里定义了，也就是要把这个符号放到未解决符号表里面去（**外部链接**）。

**static：**

如果该关键字位于全局函数或者变量的声明前面，表明该编译单元不导出这个函数或变量，因些这个符号不能在别的编译单元中使用（**内部链接**）。如果是static局部变量，则该变量的存储方式和全局变量一样，但是仍然不导出符号。

**const默认内部链接**

因为它是常量，初始化后就不能改变，这样即使每一个包含它的模块都有一份它的复制，那也不会导致不一致。如果变量默认是内部链接，它是可变的量，所以在每个包含它的模块中，它的值可能会被改变，从而导致不一致的状况出现。

**外部链接的利弊**：

外部链接的符号在整个程序范围内都是可以使用的，这就要求其他编译单元不能导出相同的符号（不然就会报 duplicated external symbols）。

## 静态链接和动态链接

**静态链接：**

函数的代码将从其所在的静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。

静态链接方法：#pragma comment(lib, "test.lib") ，静态链接的时候，载入代码就会把程序

会用到的动态代码或动态代码的地址确定下来。

静态库的链接可以使用静态链接，动态链接库也可以使用这种方法链接导入库。

**动态链接：**

函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

动态链接方法：LoadLibrary()/GetProcessAddress()和FreeLibrary()，使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码时，载入程序才计算被调用的那部分动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序又去计算这部分代码的逻辑地址，所以，这种方式使程序初始化时间较短，但运行期间的性能比不上静态链接的程序。

## 静态库和动态库

在linux中，静态库为lib\*.a，动态库为lib\*.so。

g++(gcc)编译选项

 -shared ：指定生成动态链接库。

 -static ：指定生成静态链接库。

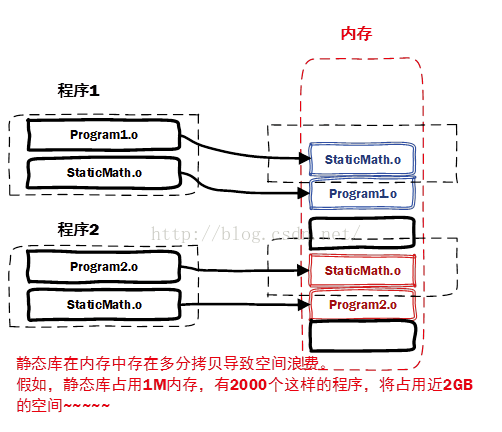
**静态链接库：**

在链接阶段，将源文件中用到的库函数与汇编生成的目标文件.o合并生成可执行文件。该可执行文件可能会比较大。

函数和数据被编译进一个二进制文件(通常扩展名为.LIB)。在使用静态库的情况下，在编译链接可执行文件时，链接器从库中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件(.EXE文件)。

**优点：**方便程序移植，因为可执行程序与库函数再无关系，静态库和应用程序编译在一起，在任何情况下都能运行。代码的装载速度快，执行速度也较快，因为编译时它只会把你需要的那部分链接进去。

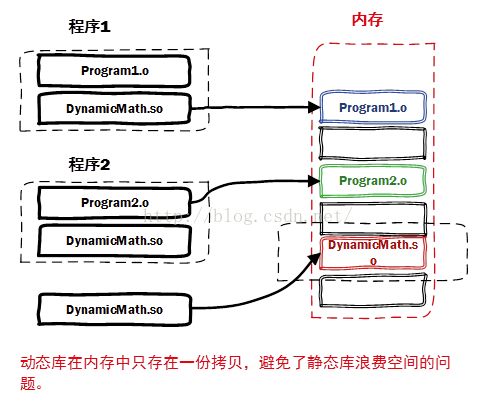
**缺点：**文件太大，往往实现很小的一个功能就需要占用很大的空间，一个全静态方式生成的简单print文件都有857K。而动态链接生成的一样的可执行文件却只要8.４Ｋ。而且每次库文件升级的话，都要重新编译源文件，很不方便。应用程序相对比较大。但是如果多个应用程序使用的话，会被装载多次，浪费内存。

  
对于静态编译的程序１和程序２，都应用库staticMath。在内存中就又两份相同的staticＭath目标文件，很浪费空间，一旦程序数量过多就很可能会内存不足。

**动态链接库：**

**优点：**1.共享：多个应用程序可以使用同一个动态库，启动多个应用程序的时候，只需要将动态库加载到内存一次即可；2.开发模块好：要求设计者对功能划分的比较好。

**缺点：**动态库是动态链接，顾名思义就是在应用程序启动的时候才会链接，所以，当用户的系统上没有该动态库时，应用程序就会运行失败。

  
我们看到在这种模型中，两个程序只应用一个库，这个目标文件在内存中只有一份，供所有程序使用。并且在程序运行过程中动态调用库文件，很方便，又不占空间，但是动态链接有一个缺点就是可移植性太差，如果两台电脑运行环境不同，动态库存放的位置不一样，很可能导致程序运行失败。

Windows为应用程序提供了丰富的函数调用，这些函数调用都包含在动态链接库中。其中有3个最重要的DLL，Kernel32.dll，它包含用于管理内存、进程和线程的各个函数；User32.dll，它包含用于执行用户界面任务(如窗口的创建和消息的传送)的各个函数；GDI32.dll，它包含用于画图和显示文本的各个函数。

### 导入库(Import Library)

在使用动态链接库的时候，往往提供两个文件：一个引入库和一个DLL。

引入库包含被DLL导出的函数和变量的符号名，DLL包含实际的函数和数据。

在编译链接可执行文件时，只需要链接引入库，DLL中的函数代码和数据并不复制到可执行文件中，在运行的时候，再去加载DLL，访问DLL中导出的函数。

在运行Windows程序时，它通过一个被称作“动态链接”的进程与Windows相接。一个Windows的.EXE文件拥有它使用不同动态链接库的引用，所使用的函数即在那里。当Windows程序被加载到内存中时，程序中的调用被指向DLL函数的入口，如果DLL不在内存中，系统就将其加载到内存中。当链接Windows程序以产生一个可执行文件时，你必须链接由编程环境提供的专门的“导入库(import library)库”。这些导入库包含了动态链接库名称和所有Windows函数调用的引用信息。链接程序使用该信息在.EXE文件中构造一个表，当加载程序时，Windows使用它将调用转换为Windows函数。

### 静态库与导入库的区别：

导入库和静态库的区别很大，他们实质是不一样的东西。静态库本身就包含了实际执行代码、符号表等等，而对于导入库而言，其实际的执行代码位于动态库中，导入库只包含了地址符号表等，确保程序找到对应函数的一些基本地址信息。

**C++运行过程**

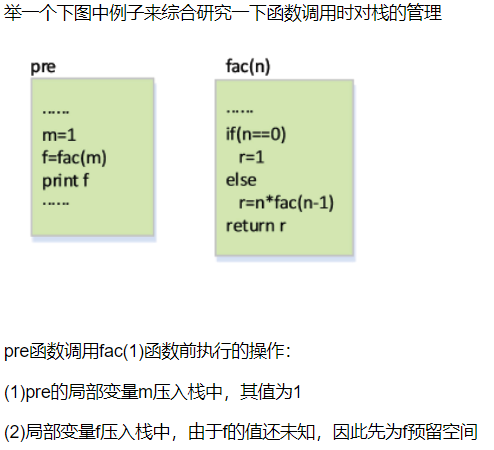
运行期指的是你将可执行文件交给操作系统（输入文件名，回车）执行、直到程序执行结束。执行的目的是为了实现程序的功能。

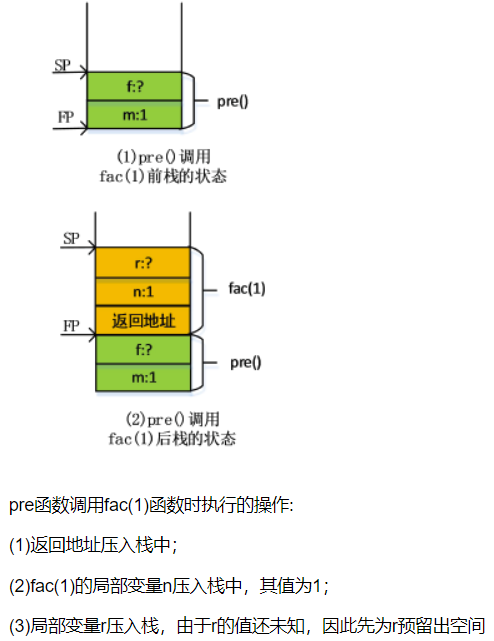
运行期分配内存，就是用malloc()之类的函数，在堆上分配内存。

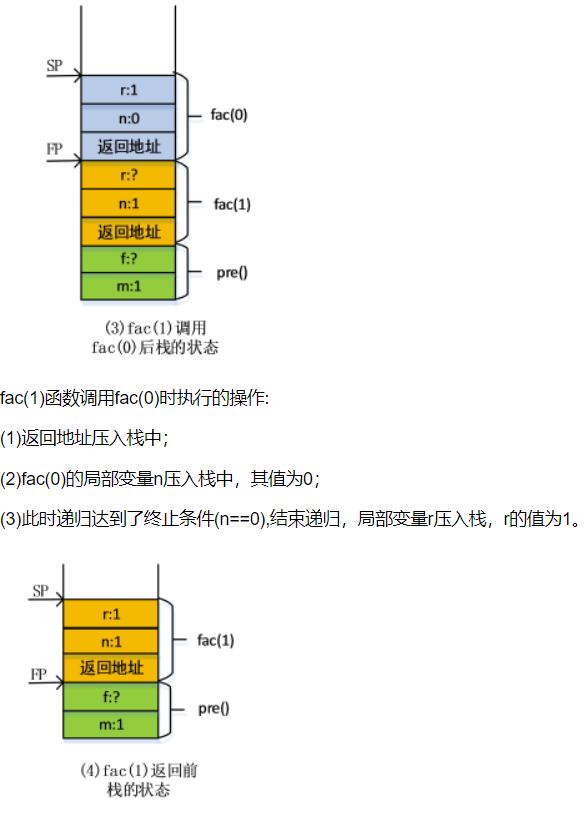
运行期内存错误，就是运行的时候发生的，比如申请不到内存，内存越界访问，等等。

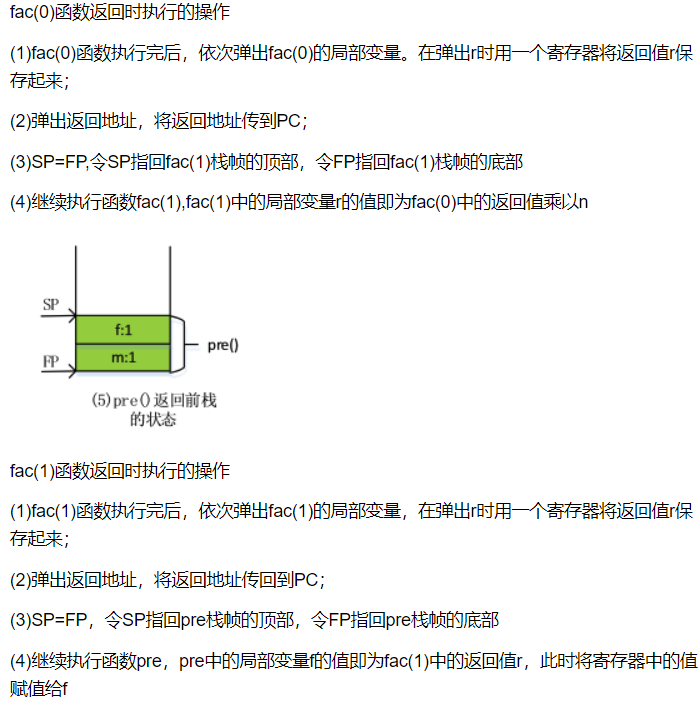
首先，执行一些命令，使**可执行文件中的代码和数据从磁盘复制到主存**，即程序的**加载**过程。注意：**利用直接存储器存取（DMA）技术，数据可以不通过处理器而直接从磁盘到达主存**。可执行文件的代码和数据被加载到主存后，处理器就开始执行程序的main中的机器指令。

加载可执行的hello文件，将hello目标文件的代码和数据从磁盘复制到主存，数据包含输出的字符串"HELLO,WORLD\n"。一旦目标文件hello中的代码和数据被加载到主存，处理器开始执行main的机器语言指令，将字符串从主存拷贝到寄存器，并输出到屏幕上。由于涉及大量的主存，磁盘，寄存器通信，故产生了cache等缓冲提高速度的设备，减少通信阻塞。









**输入输出流C/C++**

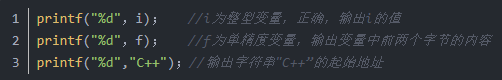
C++ 的 I/O 发生在流中，流是字节序列。如果字节流是从设备（如键盘、磁盘驱动器、网络连接等）流向内存，这叫做**输入**操作。如果字节流是从内存流向设备（如显示屏、打印机、磁盘驱动器、网络连接等），这叫做**输出**操作。

输入和输出是数据传送的过程，数据如流水一样从一处流向另一处。C++形象地将此过程称为流(stream)。C++的输入输出流是指由若干字节组成的字节序列，这些字节中的数据按顺序从一个对象传送到另一对象。流表示了信息从源到目的端的流动。在输入操作时，字节流从输入设备(如键盘、磁盘)流向内存，在输出操作时，字节流从内存流向输出设备(如屏幕、打印机、磁盘等)。流中的内容可以是ASCII字符、二进制形式的数据、图形图像、数字音频视频或其他形式的信息。

## C语言中I/O存在问题

1、在C语言中，用prinff和scanf进行输入输出，往往不能保证所输入输出的数据是可靠安全的。

学过C语言的读者可以分析下面的用法：想用格式符％d输出一个整数，但不小心错用了它输出单精度变量和字符串，会出现什么情况?假定所用的系统int型占两个字节。



编译系统认为以上语句都是合法的，而不对数据类型的合法性进行检查，显然所得到的结果不是人们所期望的。

2、在用scanf输入时，有时出现的问题是很隐蔽的。如



如已有声明语句"int i=1"，定义i为整型变量，其初值为1。编译系统不认为上面的scanf语句出错，而是将输入的值存放到地址为000001的内存单元中，这个错误可能产生严重的后果。  
  
注意：C++为了与C兼容，保留了用printf和scanf进行输出和输入的方法，以便使过去所编写的大量的C程序仍然可以在C++的环境下运行，但是希望读者在编写新的C++程序时不要用C的输入输出机制，而要用C++自己特有的输入输出方法。在C++的输入输出中，编译系统对数据类型进行严格的检查，凡是类型不正确的数据都不可能通过编译。因此C++的I/0操作是类型安全(typesafe)的。

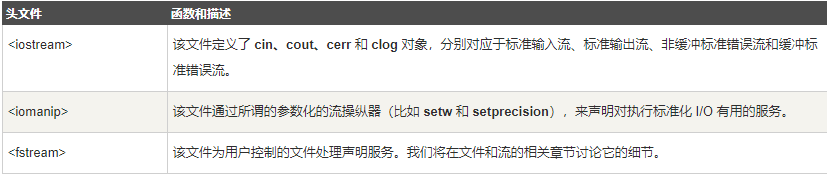
3、用printf和scanf可以输出和输入标准类型(如:int，float，double，char)的数据，但无法输出用户自己声明的类型(如数组、结构体、类)的数据。在C++中，会经常遇到对类对象的输入输出，显然无法使用printf和scanf来处理。C++的I／O操作是可扩展的，不仅可以用来输入输出标准类型的数据，也可以用于用户自定义类型的数据。C++对标准类型的数据和对用户声明类型数据的输入输出，采用同样的方法处理。显然，在用户声明了一个新类后，是无法用printf和scanf函数直接输出和输入这个类的对象的。

**解决办法：**  
   可扩展性是C++输入输出的重要特点之一，它能提高软件的重用性，加快软件的开发过程。  
   C++通过I/O类库来实现丰富的I/0功能。这样使C++的输入输出明显地优于C语言中的pfintf和scanf，但是也为之付出了代价，C++的I/O系统变得比较复杂，要掌握许多细节。

## C++的输入输出流

实际上，在内存中为每一个数据流开辟一个内存缓冲区，用来存放流中的数据。当用cout和插入运算符“<<”向显示器输出数据时，先将这些数据送到程序中的输出缓冲区保存，直到缓冲区满了或遇到endl，就将缓冲区中的全部数据送到显示器显示出来。在输入时，从键盘输入的数据先放在键盘缓冲区中，当按回车键时，键盘缓冲区中的数据输入到程序中的输入缓冲区，形成cin流，然后用提取运算符“>>”从输入缓冲区中提取数据送给程序中的有关变量。总之，流是与内存缓冲区相对应的，或者说，缓冲区中的数据就是流。  
  
   在C++中，输入输出流被定义为类。C++的I/0库中的类称为流类(streamclass)。用流类定义的对象称为流对象。

C++的输入与输出包括以下3方面的内容：  
   1、对系统指定的标准设备的输入和输出。简称标准I/O。（设备）  
   2、以外存磁盘(或光盘)文件为对象进行输入和输出。简称文件I/0。（文件）  
   3、对内存中指定的空间进行输入和输出。简称串I/O。（内存）



**标准输出流（cout）**

预定义的对象 cout 是 iostream 类的一个实例。cout 对象"连接"到标准输出设备，通常是显示屏。cout 是与流插入运算符 << 结合使用的。C++ 编译器根据要输出变量的数据类型，选择合适的流插入运算符来显示值。<< 运算符被重载来输出内置类型（整型、浮点型、double 型、字符串和指针）的数据项。

流插入运算符 << 在一个语句中可以多次使用，endl用于在行末添加一个换行符。

**标准输入流（cin）**

预定义的对象 cin 是 iostream 类的一个实例。cin 对象附属到标准输入设备，通常是键盘。cin 是与流提取运算符 >> 结合使用的。C++ 编译器根据要输入值的数据类型，选择合适的流提取运算符来提取值，并把它存储在给定的变量中。

流提取运算符 >> 在一个语句中可以多次使用

**标准错误流（cerr）**

预定义的对象 cerr 是 iostream 类的一个实例。cerr 对象附属到标准错误设备，通常也是显示屏，但是 cerr 对象是非缓冲的，且每个流插入到 cerr 都会立即输出。

cerr 也是与流插入运算符 << 结合使用的



**标准日志流（clog）**

预定义的对象 clog 是 iostream 类的一个实例。clog 对象附属到标准错误设备，通常也是显示屏，但是 clog 对象是缓冲的。这意味着每个流插入到 clog 都会先存储在缓冲在，直到缓冲填满或者缓冲区刷新时才会输出。

clog 也是与流插入运算符 << 结合使用的



**声明与定义的区别**

函数或变量在声明时，并没有给它实际的物理内存空间，它有时候可以保证你的程序编译通过， 但是当函数或变量定义的时候，它就在内存中有了实际的物理空间，如果你在编译模块中引用的外部变量没有在整个工程中任何一个地方定义的话， 那么即使它在编译时可以通过，在连接时也会报错，因为程序在内存中找不到这个变量！你也可以这样理解， 对同一个变量或函数的声明可以有多次，而定义只能有一次!

**头文件里一般只可以有声明不能有定义：**

头文件可以被多个编译单元包含，如果头文件里面有定义的话，那么每个包含这头文件的编译单元都会对同一个符号进行定义，如果该符号为外部链接，则会导致duplicated external symbols链接错误。

**------\*指针引用/数组链表队列\*------**

**值传递/引用传递/指针传递**

**值传递**

是只把对象的值传入函数，函数中可以使用这个值，但却无法更改该对象的值。这种传递方式中，实参和形参是两个不同的地址空间，参数传递的实质是将原函数中变量的值，复制到被调用函数形参所在的存储空间中，这个形参的地址空间在函数执行完毕后，会被回收掉。整个被调用函数对形参的操作，只影响形参对应的地址空间，不影响原来函数中的变量的值，因为这两个不是同一个存储空间。

　值传递（passl-by-value）过程中，被调函数的形式参数作为被调函数的局部变量处理，**即在堆栈中开辟了内存空间以存放由主调函数放进来的实参的值，从而成为了实参的一个副本。**值传递的特点是被调函数对形式参数的任何操作都是作为局部变量进行，不会影响主调函数的实参变量的值。





值传递swap传不成功，因为在栈区，结束会销毁。

**引用传递**

传递的是内存地址，修改后会改变内存地址对应储存的值。引用传递是将整个对象本身（或地址）传入函数，在函数中既可调用对象的值，也可改变对象的值。这种参数传递方式中，形参是引用类型变量，其实就是实参的一个别名，在被调用函数中，对引用变量的所有操作等价于对实参的操作，这样，整个函数执行完毕后，原先的实参的值将会发生改变。

引用传递(pass-by-reference)过程中，被调函数的形式参数虽然也作为局部变量在堆栈中开辟了内存空间，**但是这时存放的是由主调函数放进来的实参变量的地址。**被调函数对形参的任何操作都被处理成间接寻址，即通过堆栈中存放的地址访问主调函数中的实参变量。正因为如此，被调函数对形参做的任何操作都影响了主调函数中的实参变量。





**指针传递**

这种参数传递方式中，实参是变量的地址，形参是指针类型的变量，在函数中对指针变量的操作，就是对实参（变量地址）所对应的变量的操作，，函数调用结束后，原函数中的变量的值将会发生改变。





**比较：**

1.从功能上。按值传递在传递的时候，实参被复制了一份，然后在函数体内使用，**函数体内修改参数变量时修改的是实参的一份拷贝，**而实参本身是没有改变的，所以如果想在调用的函数中修改实参的值，使用值传递是不能达到目的的，这时只能使用引用或指针传递。例如，要实现两个数值交换。

 void swap(int a  int b)

 void main(){

     int a=1  b=2

     swap(a b)

 }

 这样，在main()函数中的a b值实际上并没有交换，如果想要交换只能使用指针传递或引用传递，如：

 void swap(int pa int pb)

 或

 void swap(int& ra int& rb)

**2.从传递效率上。这里所说传递效率，是说调用被调函数的代码将实参传递到被调函数体内的过程**，正如上面代码中，这个过程就是函数main()中的a b传递到函数swap()中的过程。这个效率不能一概而论。对于内建的int char short long float等4字节或以下的数据类型而言，实际上传递时也只需要传递1－4个字节，而使用指针传递时在32位cpu中传递的是32位的指针，4个字节，都是一条指令，这种情况下值传递和指针传递的效率是一样的，而传递double  long long等8字节的数据时，在32位cpu中，其传值效率比传递指针要慢，因为8个字节需要2次取完。而在64位的cpu上，传值和传址的效率是一样的。

再说引用传递，这个要看编译器具体实现，引用传递最显然的实现方式是使用指针，这种情况下与指针的效率是一样的，而有些情况下编译器是可以优化的，采用直接寻址的方式，这种情况下，效率比传值调用和传址调用都要快，与采用全局变量方式传递的效率相当。

再说自定义的数据类型，class  struct定义的数据类型。这些数据类型在进行传值调用时生成临时对象会执行构造函数，而且当临时对象销毁时会执行析构函数，如果构造函数和析构函数执行的任务比较多，或者传递的对象尺寸比较大，那么传值调用的消耗就比较大。这种情况下，采用传址调用和采用传引用调用的效率大多数下相当，正如上面所说，某些情况下引用传递可能被优化，总体效率稍高于传址调用。

**3.从执行效率上讲。这里所说的执行效率，是指在被调用的函数体内执行时的效率。**因为传值调用时，当值被传到函数体内，临时对象生成以后，所有的执行任务都是通过直接寻址的方式执行的，而指针和大多数情况下的引用则是以间接寻址的方式执行的，所以实际的执行效率会比传值调用要低。如果函数体内对参数传过来的变量进行操作比较频繁，执行总次数又多的情况下，传址调用和大多数情况下的引用参数传递会造成比较明显的执行效率损失。

综合2、3两种情况，具体的执行效率要结合实际情况，通过比较传递过程的资源消耗和执行函数体消耗之和来选择哪种情况比较合适。**而就引用传递和指针传递的效率上比，引用传递的效率始终不低于指针传递，所以从这种意义上讲，在c＋＋中进行参数传递时优先使用引用传递而不是指针。**

4.从类型安全上讲。值传递与引用传递在参数传递过程中都执行强类型检查，而指针传递的类型检查较弱，特别地，如果参数被声明为 void ，那么它基本上没有类型检查，只要是指针，编译器就认为是合法的，所以这给bug的产生制造了机会，使程序的健壮性稍差，如果没有必要，就使用值传递和引用传递，最好不用指针传递，更好地利用编译器的类型检查，使得我们有更少的出错机会，以增加代码的健壮性。

**这里有个特殊情况，就是对于多态的情况，**如果形参是父类，而实参是子类，在进行值传递的时候，临时对象构造时只会构造父类的部分，是一个纯粹的父类对象，而不会构造子类的任何特有的部分，因为办有虚的析构函数，而没有虚的构造函数，这一点是要注意的。如果想在被调函数中通过调用虚函数获得一些子类特有的行为，这是不能实现的。

 5. 从参数检查上讲。一个健壮的函数，总会对传递来的参数进行参数检查，保证输入数据的合法性，以防止对数据的破坏并且更好地控制程序按期望的方向运行，在这种情况下使用值传递比使用指针传递要安全得多，因为你不可能传一个不存在的值给值参数或引用参数，而使用指针就可能，很可能传来的是一个非法的地址（没有初始化，指向已经delete掉的对象的指针等）。所以使用值传递和引用传递会使你的代码更健壮，具体是使用引用还是使用，**最简单的一个原则就是看传递的是不是内建的数据类型，对内建的数据类型优先使用值传递，而对于自定义的数据类型，特别是传递较大的对象，那么请使用引用传递。**

 6. 从灵活性上。无疑，指针是最灵活的，因为指针除了可以像值传递和引用传递那样传递一个特定类型的对象外，还可以传递空指针，不传递任何对象。指针的这种优点使它大有用武之地，比如标准库里的time( )函数，你可以传递一个指针给它，把时间值填到指定的地址，你也可以传递一个空指针而只要返回值。

**智能指针**

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

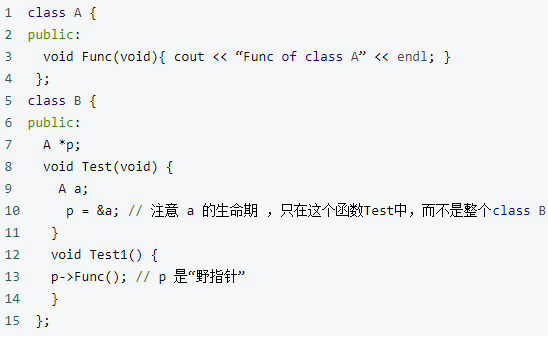
**空指针和野指针**

**“野指针”产生的原因：**

指针定义时未被初始化：指针在被定义的时候，如果程序不对其进行初始化的话，它会指向随机区域，因为任何指针变量（除了static修饰的指针变量）在被定义的时候是不会被置空的，它的默认值是随机的。

指针被释放时没有被置空：我们在用malloc开辟内存空间时，要检查返回值是否为空，如果为空，则开辟失败；如果不为空，则指针指向的是开辟的内存空间的首地址。指针指向的内存空间在用free()或者delete（注意delete只是一个操作符，而free()是一个函数）释放后，如果程序员没有对其置空或者其他的赋值操作，就会使其成为一个野指针。

指针操作超越变量作用域：不要返回指向栈内存的指针或引用，因为栈内存在函数结束的时候会被释放



函数 Test1 在执行语句 p->Func()时，p 的值还是 a 的地址，对象 a 的内容已经被清除，所以 p 就成了“野指针” 。

**“野指针”的危害**：

野指针的问题在于，指针指向的内存已经无效了，而指针没有被置空，解引用一个非空的无效指针是一个未被定义的行为，也就是说不一定导致段错误，野指针很难定位到是哪里出现的问题，在哪里这个指针就失效了，不好查找出错的原因。所以调试起来会很麻烦，有时候会需要很长的时间。

**规避方法**

1、初始化指针时将其置为NULL，之后再对其进行操作

2、释放指针时将其置为NULL，最好在编写代码时将free()函数封装一下，在调用free()后就将指针置为NULL。为了避免给野指针发送消息会报错，一般情况下，当一个对象被释放后我们会将这个对象的指针设置为空指针。

**空指针：**

一般声明一个指针变量赋值为NULL，这就是空指针，各个类型的空指针都存在确确实实的内存地址，但是不会指向任何有效的值的内存地址，对空指针操作，例如访问属性和方法，会抛出空指针异常，因为空指针指向的内存地址没有对应的物理地址。

**野指针：**

指那些释放内存，但是指针赋值为空，这时候的指针指向任意地址，好可怕，例如指向内核地址或不属于本程序的内存地址，程序会被kill，即奔溃。

**内存泄漏：**

分为堆泄露和资源泄露 两种，内存分配失败或者内存分配成功却没有指针指向它（即无法操作该内存），会导致内存分配的越来越多，导致系统内存不够而终止程序

**指针常量和常量指针**

**const int \*p**

“指向常量的指针”的最佳理解应为：我们不能通过该指针修改它所指向的东西（常量或者变量）。注意，const int \*p=&a;只是说不能通过p来修改a，如果a本身不是const的，通过其它方式修改a自然是可以的（例如直接++a）。

另外一点，由于p本身只是一个普通的指针，所以允许在声明时不初始化。

但需要注意的是，我们只是说可以，但并不提倡这样做。在任何时候都不应该让指针无所指，如果在声明一个指针时还不知道让它指向谁，就先将其初始化为nullptr或NULL（nullptr是C++11新特性，用它比用NULL更安全些）。

\*的右边没有限定成分，表明p就是我们熟悉的普通指针，p的内容（即值，也就是它指向的东西的地址）是可以改变的；\*的左边是const int，表明p指向的东西是一个const的int，我们不能通过p来修改这个int,因为它是const的。

**int\* const p**

就是所谓的“本身是常量的指针”。关于“p本身不能修改但可以通过p修改其所指”这一点，我们在讲判断方法时已经说过，这里主要再说一下p的初始化。由于p本身是const的，所以在编译的时候必须知道p的值（即p所指向的东西的地址），所以在声明p的同时必须初始化p。但要注意，对于 int\* const p=&a，我们只要求a的地址是确定的，但a的值可以不确定。

\*的右边是const，表明p本身是const的，我们不能对p的内容进行修改（例如++p;是不可以的），\*的左边是int，即p指向的东西是普通的int，我们可以通过p来修改它（例如 \*p=100;是可以的）。

**const int\* const p**

就是所谓的“所指和本身都是常量的指针”。

\*的右边是const，表明指针p本身是const的，\*的左边是const int，表明p指向的int也是const的。即这种情况下，p本身不能修改，同时也不能通过p修改它所指向的那个int。

**指针和引用**

**相同点：**指针指向一块内存，而引用是一块内存的别名；

**不同点：**

1指针是一个实体，他在栈中有自己使用的空间，但是引用没有；

2引用必须初始化，指针不用但是最好初始化

3指针使用时必须加\*然后解引用，引用不用；

4引用只能初始化一次，指针不是；

5引用不用const去修饰，但是指针可以，const 的指针不可变

6指针和地址运用自增（++）不同，引用是值进行自增，而指针是地址进行自增；

7引用不能为空，指针可以为空

8“sizeof 引用”得到的是所指向的变量(对象)的大小，而“sizeof 指针”得到的是指针本身(所指向的变量或对象的地址)的大小

**联系**

1引用的内部使用指针实现的（如何实现这里不做解释）

2引用是受了限制的指针

**队列和堆栈的区别**

队列和堆栈是一个更高层次的概念, 是一种特殊的数据组织形式，其底层可以是用链表也可以是用数组来实现.

栈的特点是先进后出，队列的特点是先进先出。

队列可以看成是有2个口的集合一个口叫队头一个叫队尾，只能在对头进行删除操作，在队尾做插入。

堆栈可以看成是有1个口的集合，这个口叫栈顶。插入和删除操作只能在栈顶操作。根据这样的操作。

**数组和链表的区别**

链表实际上可以认为是一种数据的物理组织形式,是用指针或对象的引用组织起的一种数据的存储方式。它可以在非连续的内存空间里面存储一个集合的元素。和它对应的是数组，数组要在连续的空间里存储集合的元素。链表就是带着指针的结构体，含数据域和指向下一个节点的地址。头部没有前驱节点，尾部指向NULL，增删快于数组，遍历慢于数组。

**数组的特点：**

随机访问性强；查找速度快。插入和删除效率低；可能浪费内存；内存空间要求高，必须有足够的连续内存空间；数组大小固定，不能动态拓展。

在内存中，数组是一块连续的区域。数组需要预留空间，在使用前要先申请占内存的大小，可能会浪费内存空间。插入数据和删除数据效率低，插入数据时，这个位置后面的数据在内存中都要向后移。删除数据时，这个数据后面的数据都要往前移动。随机读取效率很高。因为数组是连续的，知道每一个数据的内存地址，可以直接找到给地址的数据。不利于扩展，数组定义的空间不够时要重新定义数组。

**链表的特点：**

插入删除速度快；内存利用率高，不会浪费内存；大小没有固定，拓展很灵活；不能随机查找，必须从第一个开始遍历，查找效率低。

在内存中可以存在任何地方，不要求连续。每一个数据都保存了下一个数据的内存地址，通过这个地址找到下一个数据。增加数据和删除数据很容易。查找数据时效率低，因为不具有随机访问性，所以访问某个位置的数据都要从第一个数据开始访问，然后根据第一个数据保存的下一个数据的地址找到第二个数据，以此类推。不指定大小，扩展方便。链表大小不用定义，数据随意增删。

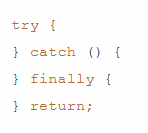
**---------\*函数/类/结构体\*----------**

**try-catch-finally**

异常try-catch-finally中，try和finally中都有return，怎么返回

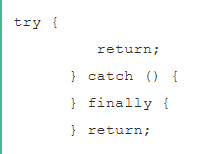
**结论：**  
1、不管有木有出现异常，finally块中代码都会执行；  
2、当try和catch中有return时，finally仍然会执行；  
3、finally是在return后面的表达式运算后执行的（此时并没有返回运算后的值，而是先把要返回的值保存起来，不管finally中的代码怎么样，返回的值都不会改变，仍然是之前保存的值），所以函数返回值是在finally执行前确定的；  
4、finally中最好不要包含return，否则程序会提前退出，返回值不是try或catch中保存的返回值。

**情况1：**



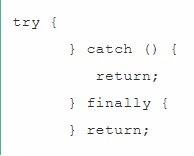
显然程序按顺序执行。

**情况2:**



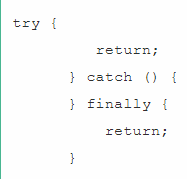
程序执行try块中return之前（包括return语句中的表达式运算）代码；再执行finally块，最后执行try中return;finally块之后的语句return，因为程序在try中已经return所以不再执行。

**情况3:**



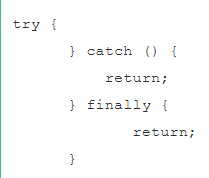
程序先执行try，如果遇到异常执行catch块，有异常：则执行catch中return之前（包括return语句中的表达式运算）代码，再执行finally语句中全部代码，最后执行catch块中return. finally之后的代码不再执行。无异常：执行完try再finally再return.

**情况4:**



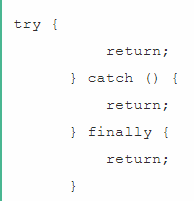
程序执行try块中return之前（包括return语句中的表达式运算）代码；再执行finally块，因为finally块中有return所以提前退出。

**情况5:**



程序执行catch块中return之前（包括return语句中的表达式运算）代码；再执行finally块，因为finally块中有return所以提前退出。

**情况6:**



程序执行try块中return之前（包括return语句中的表达式运算）代码；有异常：执行catch块中return之前（包括return语句中的表达式运算）代码；则再执行finally块，因为finally块中有return所以提前退出。无异常：则再执行finally块，因为finally块中有return所以提前退出。

最终结论：任何执行try 或者catch中的return语句之前，如果finally存在的话,都会先执行finally语句。如果finally中有return语句，那么程序就return了，所以finally中的return是一定会被return的，编译器把finally中的return实现为一个warning。

**模板**

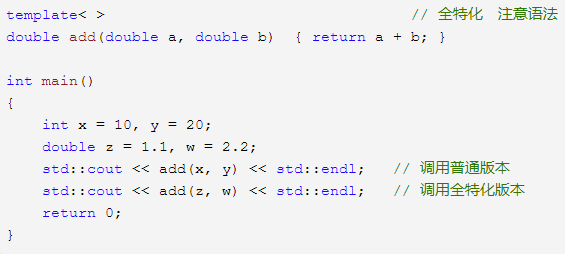
## 模板偏特化

全特化就是限定死模板实现的具体类型，偏特化就是如果这个模板有多个类型，那么只限定其中的一部分。

**函数模板特例化**

函数模板只能全特化，不能偏特化，如果要偏特化的话只能重载

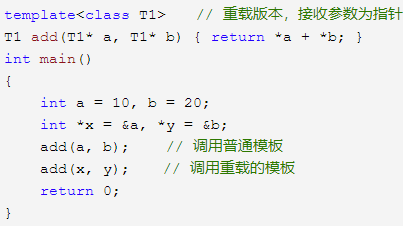
**函数模板全特化**



如果有与实参更加匹配的特例化版本，编译器将会选择特例化版本

**函数模板重载（不存在偏特化）**

因为偏特化版本本质上仍然是模板，所以如果需要的话，可以重载一个函数模板



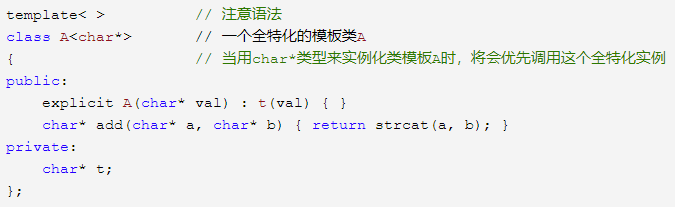
如上，如果需要一个接收指针的偏特化版本，那么可以用重载模板实现  
函数模板不存在偏特化

**类模板特例化**

类模板既有全特化，又有偏特化  
这里重新写一个更一般的模板类来说明类模板的特例化

**类模板全特化**

类模板全特化比较好理解，跟函数模板一样，全特化是一个实例，当编译器匹配时会优先匹配参数一致的实例



**类模板的偏特化**

类模板偏特化本质上都是指定部分类型，让偏特化版本称为普通版本的子集，若实例化时参数类型为指定的类型，则优先调用特例化版本。对主版本模板类、全特化类、偏特化类的调用优先级从高到低进行排序是：**全特化类>偏特化类>主版本模板类**。这样的优先级顺序对性能也是最好的。  


**几个值得注意的地方**  
１．特例化本质上是我们顶替了编译器的工作，我们帮编译器做了类型推导  
２．全特化本质上是一个实例，而偏特化本质上还是一个模板，只是原来模板的一个子集  
３．所以全特化的函数模板，本质上是实例，从而不会与函数模板产生二义性  
４．若想让用户能使用特例化版本，特例化版本必须与模板定义在同一个.h头文件中

## 类模板偏特化与STL

模板特化并不只是为了性能优化，更多是为了让模板函数能够正常工作，最典型的例子就是STL中的iterator\_traits。algorithm中大多数算法通过iterator对象来处理数据，但是同时允许以指针代替iterator对象，这是为了支持C-Style Array。如果直接操作iterator，那么为了支持指针类型，每个函数都需要进行重载，因为指针没有::value\_type类型。为了解决这个问题，STL使用了iterator\_traits，并为指针类型进行转化，算法通过它来操作iterator，不需要知道实际操作的是iterator对象还是指针。

**偏特化在STL中最重要的两个应用**

**１．应用在迭代器设计中，为了使迭代器既可以萃取出值类型，又可以包容原生指针**  
如果要通过一个迭代器就能知道它的值类型，那么一般会使用iterator\_traits  
**迭代器萃取技术的两个核心是：**  
**１）在每个迭代器类中定义value\_type值类型的类型成员，这样直接通过迭代器的value\_type类型成员就可以知道值类型**  
**２）问题就在于，迭代器必须兼容原生指针，而原生指针很难被重新定义，即要在原生指针的类中添加value\_type的值类型的类型成员．这时候，靠的就是类模板的偏特化了．新添加一层iterator\_traits类，专门萃取迭代器的属性，然后再对iterator\_traits类设计原生指针与原生引用的偏特化版本，就解决了这个棘手的问题**

**２．type\_traits类型萃取，对待特殊类型，特殊处理，提高效率**  
对于没有构造函数，析构函数等的内置类型，如果与复杂类型一样，执行同样的操作，显然是效率不高的  
先实现一个对所有类型都设置一个最保守值的type\_traits模板类，然后再对每个内置类型设置偏特化版本，内置类型设置一个更为激进的值，表明可以采取更为高效的操作来提高效率  
比如copy函数，如果传递的对象是一个复杂类型，那么可能只能采取最保守的处理方式，一个一个的构造；如果是内置类型，这样显然太低效，使用memcpy()可能会好一些

其实iterator\_traits也不止是处理兼容原生指针的问题，它也可以提高效率．  
迭代器分为很多种，有可以随机访问的（vector），有只能前后一个一个移动的（list）,也有只能单向移动的（slist），所以一般把迭代器分为五种：  
InputIterator　　　　　　　输入迭代器  
OutputIterator 　　　　　 　输出迭代器  
ForwardIterator　　　　　　单向迭代器  
BidirectionIterator　　　　　双向迭代器  
RandomAccessIterator　　　随机访问迭代器  
比如一个advance(n)函数，对于单向迭代器只能一个一个移动过去，但是这种实现对于随机访问迭代器显然不是理想的处理方式  
处理的方式就是先实现这五个类，用作标记用，在每个迭代器里面都定义迭代器类型的类型成员iterator\_catagory，再对不同版本的迭代器实现不同的advance(n)处理方式

## 模板函数对常数类型输入返回1

template <typename T>  
struct is\_const : std::false\_type {};   
  
template <typename T>  
struct is\_const<const T> : std::true\_type {};

#include <type\_traits>

#include <iostream>

struct trivial

{

int val;

};

int main()

{

std::cout << "is\_const<trivial> == " << std::boolalpha

<< std::is\_const<trivial>::value << std::endl;

std::cout << "is\_const<const trivial> == " << std::boolalpha

<< std::is\_const<const trivial>::value << std::endl;

std::cout << "is\_const<int> == " << std::boolalpha

<< std::is\_const<int>::value << std::endl;

std::cout << "is\_const<const int> == " << std::boolalpha

<< std::is\_const<const int>::value << std::endl;

return (0);

}

is\_const<trivial> == false

is\_const<const trivial> == true

is\_const<int> == false

is\_const<const int> == true

## 模板声明和定义要放在同一文件中？

通常情况下，你会在.h文件中声明函数和类，而将它们的定义放置在一个单独的.cpp文件中。但是在使用模板时，这种习惯性做法将变得不再有用，因为当实例化一个模板时，编译器必须看到模板确切的定义，而不仅仅是它的声明。因此，最好的办法就是将模板的声明和定义都放置在同一个.h文件中。这就是为什么所有的STL头文件都包含模板定义的原因。

对C++编译器而言，当调用函数的时候，编译器只需要看到函数的声明。当定义类类型的对象时，编译器只需要知道类的定义，而不需要知道类的实现代码。因此，因该将类的定义和函数声明放在头文件中，而普通函数和类成员函数的定义放在源文件中。  
  但在处理模板函数和类模板时，问题发生了变化。要进行实例化模板函数和类模板，要求编译器在实例化模板时必须在上下文中可以查看到其定义实体；而反过来，在看到实例化模板之前，编译器对模板的定义体是不处理的——原因很简单，**编译器怎么会预先知道 typename 实参是什么呢？因此模板的实例化与定义体必须放到同一翻译单元中。**

## 显式接口与隐式接口（模板）

**显式接口：**

是指类继承层次中定义的接口或是某个具体类提供的接口，总而言之，我们能够在源代码中找到这个接口.显式接口以函数签名为中心，例如

void AnimalShot(Animal & anim)

{

anim.shout();

}

我们称shout为一个显式接口。在运行期多态中的接口皆为显式接口。

**隐式接口（template）：**

而对模板参数而言，接口是隐式的，奠基于有效表达式。例如：

template <typename T>

void AnimalShot(T & anim)

{

anim.shout();

}

对于anim来说，必须支持哪一种接口，要由模板参数执行于anim身上的操作来决定，在上面这个例子中，T必须支持shout()操作，那么shout就是T的一个隐式接口。

**Pod**

POD全称Plain Old Data。通俗的讲，一个类或结构体通过二进制拷贝后还能保持其数据不变，那么它就是一个POD类型。当class/struct是极简的(trivial)、属于标准布局(standard-layout)，以及他的所有非静态(non-static)成员都是POD时，会被视为POD。

算术类型、枚举类型、指针类型和指向成员类型的指针是 POD。

POD 类型的 cv 限定版本本身就是 POD 类型。

POD 数组本身就是 POD。

所有非静态数据成员都是 POD 的结构或联本身就是 POD，前提是它：

没有用户声明的构造函数。

没有私有或受保护的非静态数据成员。

没有基类。

没有虚函数。

没有引用类型的非静态数据成员。

没有用户定义的复制赋值运算符。

没有用户定义的析构函数。

* #include <type\_traits>
* #include <iostream>
* struct trivial
* {
* int val;
* };
* struct throws
* {
* throws() throw(int)
* {
* }
* throws(const throws&) throw(int)
* {
* }
* throws& operator=(const throws&) throw(int)
* {
* }
* int val;
* };
* int main()
* {
* std::cout << "is\_pod<trivial> == " << std::boolalpha
* << std::is\_pod<trivial>::value << std::endl;
* std::cout << "is\_pod<int> == " << std::boolalpha
* << std::is\_pod<int>::value << std::endl;
* std::cout << "is\_pod<throws> == " << std::boolalpha
* << std::is\_pod<throws>::value << std::endl;
* return (0);
* }
* /\* Output:
* is\_pod<trivial> == true
* is\_pod<int> == true
* is\_pod<throws> == false
* \*/

平凡的定义

1.有平凡的构造函数

2.有平凡的拷贝构造函数

3.有平凡的移动构造函数

4.有平凡的拷贝赋值运算符

5.有平凡的移动赋值运算符

6.有平凡的析构函数

7.不能包含虚函数

8.不能包含虚基类

标准布局的定义

1.所有非静态成员有相同的访问权限

2.继承树中最多只能有一个类有非静态数据成员

3.子类的第一个非静态成员不可以是基类类型

4.没有虚函数

5.没有虚基类

6.所有非静态成员都符合标准布局类型

POD的使用

当一个数据类型满足了”平凡的定义“和”标准布局“，我们则认为它是一个POD数据。可以通过std::is\_pod来判断一个类型是否为POD类型。

#include "stdafx.h"

#include <iostream>

#include <Windows.h>

using namespace std;

class A

{

public:

int x;

double y;

};

int \_tmain(int argc, \_TCHAR\* argv[])

{

if (std::is\_pod<A>::value)

{

std::cout << "before" << std::endl;

A a;

a.x = 8;

a.y = 10.5;

std::cout << a.x << std::endl;

std::cout << a.y << std::endl;

size\_t size = sizeof(a);

char \*p = new char[size];

memcpy(p, &a, size);

A \*pA = (A\*)p;

std::cout << "after" << std::endl;

std::cout << pA->x << std::endl;

std::cout << pA->y << std::endl;

delete p;

}

system("pause");

return 0;

}

[**函数多个参数怎么传递**](#_------*指针引用/数组链表队列*------)

**函数调用及返回的步骤**

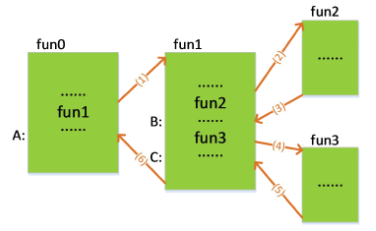
**调用**  
1、保存调用信息（参数，返回地址）  
2、分配数据区（局部变量）  
3、控制转移给被调函数的入口

**返回**  
1、保存返回信息  
2、释放数据区  
3、控制转移到上级函数（主调用函数）

**1、返回地址的存储**

　　执行一条指令时，是根据PC中存放的指令地址，将指令由内存取到指令寄存器IR中。程序在执行时按顺序依次执行每一条语句，PC通过加1来指向下一条将要执行的程序语句。但也有一些例外：(1)调用函数(2)函数调用后的返回(3)控制结构（if else while for等）

主调函数是指调用其他函数的函数，被调函数是指被其他函数调用的函数，一个函数既调用别的函数又被另外的函数调用

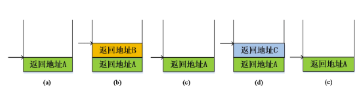


上图中，fun0函数调用fun1，fun0函数就是主调函数，fun1是被调函数

发生函数调用时，程序会跳转到被调函数的第一条语句，然后按顺序依次执行被调函数中的语句。函数调用后返回时，程序会返回到主调函数中调用函数的语句的后一条语句继续执行。换句话说，也就是“从哪里离开，就回到哪里”。

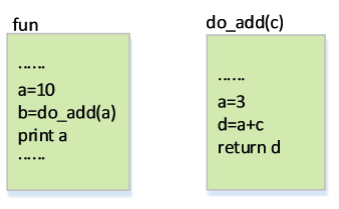
　　CPU执行程序时，并不知道整个程序的执行步骤是怎样的，完全是“走一步，看一步”。前面我们提到过，CPU都是根据PC中存放的指令地址找到要执行的语句。函数返回时，是“从哪里离开，就回到哪里”。但是当函数要从被调函数中返回时，PC怎么知道调用时是从哪里离开的呢？答案就是——将函数的“返回地址”保存起来。因为在发生函数调用时的PC值是知道的。在主调函数中的函数调用的下一条语句的地址即为当前PC值加1，也就是函数返回时需要的“返回地址”。我们只需将该返回地址保存起来，在被调函数执行完成后，要返回主调函数中时，将返回地址送到PC。这样，程序就可以往下继续执行了。

　　函数调用的特点是：越早被调用的函数，越晚返回。比如fun1函数比fun2函数先调用，但是返回的时候fun1晚于fun2返回。这一特点正是"后进先出"，所以我们采用栈来保存返回地址



如上图调用过程(1)发生时，需要压入保存返回地址A，栈的状态如图中(a)所示；调用过程(2)发生时，需要压入保存返回地址B，栈的状态如图中(b)所示；返回过程(3)发生时，需要弹出返回地址B，栈的状态如图中(c)所示；调用过程过程(4)发生时，需要压入保存返回地址C，栈的状态如图中(d)所示；返回过程(5)发生时，需要弹出返回地址C，栈的状态如图中(e)所示；返回过程(6)发生时，需要弹出返回地址A，此时栈被清空，图中未画出具体情况

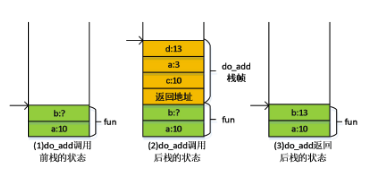
**2、函数调用时栈的管理**



如上图所示，fun函数里的变量a和do\_add函数里的变量a是两个不同的变量，这两个变量需要存放在不同的地方。局部变量a只在do\_add函数内才有意义；局部变量的存储一定是和函数的开始与结束息息相关的。局部变量如同返回地址般也是存在栈里。当函数开始执行时，这个函数的局部变量在栈里被设立（压入），当函数结束时，这个函数的局部变量和返回地址都会被弹出。

　　当函数调用时，do\_add函数里局部变量c就复制fun函数里变量a的值。在函数返回时，与参数传递同理，在传递返回值时也是将do\_add函数里的值赋值给主调函数中的变量b。局部变量只在函数内有意义，离开函数后该局部变量就失效。比如do\_add函数里的局部变量d，执行do\_add函数时d是有意义的。但执行完do\_add函数后，返回到fun函数中，do\_add函数里的局部变量d就失效了。因此在弹出d时需要用一个寄存器将返回值d保存起来，所以在外面的调用函数可以来读取这个值。

　　局部变量的调用是和栈的操作模式“后进先出”的形式是相同的。这就是为什么返回地址是压入栈里，同样的，局部变量也会压到相对应的栈里面。**当函数执行时，这个函数的每一个局部变量就会在栈里有一个空间。在栈中存放此函数的局部变量和返回地址的这一块区域叫做此函数的栈帧(frame)。当此函数结束时，这一块栈帧就会被弹出。**



调用do\_add()函数前执行的操作:(1)fun的局部变量a压入栈中，其值为10(2)局部变量b压入栈中，由于b的值还未知，因此先为b预留空间

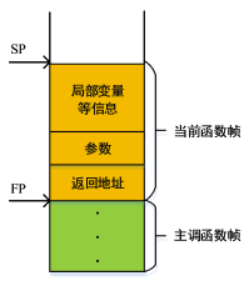
调用do\_add()函数时执行的操作:(1)返回地址压到栈中(2)局部变量c的值10压入栈中（c的值是通过复制fun函数中变量a得到的）(3)压入do\_add中的局部变量a，其值为3(4)执行a+c,其中a=3,c=10,相加后得d的值为13

do\_add()函数返回时执行的操作:(1)do\_add()函数执行完后，依次弹出do\_add()的局部变量，由于需要将d的值返回，因此在弹出d的时候需要一个寄存器将返回值d保存起来(2)弹出返回地址，将返回地址传到PC(3)返回到fun函数，fun中的局部变量b的值即为do\_add()中的返回值d，此时将寄存器中的值赋给b。

在函数调用时，用一个寄存器将栈顶地址保存起来，称为栈顶指针SP。另外还有一个帧指针FP，用来指向栈中函数信息的底端。这样，栈就被分成了一段一段的空间。每个栈帧对应一次函数调用，在栈帧中存放了前面介绍的函数调用中的返回地址、局部变量值等。每次发生函数调用时，都会有一个栈帧被压入栈的最顶端；调用返回后，相应的栈帧便被弹出。当前正在执行的函数的栈帧总是处于栈的最顶端。

　　由于函数调用时，要不断的将一些数据压入栈中，SP的位置是不断变化的，而FP的位置相对于局部变量的位置是确定的，因此函数的局部变量的地址一般通过帧指针FP来计算，而非栈指针SP。

　　综合前面所讲，可以总结出:(1)一个函数调用过程就是将数据（包括参数和返回值）和控制信息（返回地址等）从一个函数传递到另一个函数。(2)在执行被调函数的过程中，还要为被调函数的局部变量分配空间，在函数返回时释放这些空间。这些工作都是由栈来完成的。所传参数的地址可以简单的从FP算出来。下图展示了栈帧的通用结构



**系统调用与函数调用的区别**

所谓系统调用，就是用户在程序中调用[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)所提供的一些子功能。它是通过系统调用命令，中段现行程序而转去执行相应的子程序，以完成特定的系统功能。完成后，控制又返回到发出系统调用命令之后的一条指令，被中断的程序将继续执行下去。

系统调用与一般过程调用不同，其主要区别是：  
1.运行的状态不同。在程序中的过程一般或者都是用户程序，或者都是系统程序，即都是运行在同一个系统状态的（用户态或系统态）。  
2.进入的方式不同。一般的过程调用可以直接由调用过程转向被调用的过程。而执行系统调用时，由于调用过程与被调用过程是处于不同的状态，因而不允许由调用过程直接转向被调用过程，通常是通过访问管中断（即软中断）进入，先进入[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)，经分析后，才能转向相应的命令处理程序。  
3.返回方式的不同。  
4.代码层次不同。一般过程调用中的被调用程序是用户级程序，而系统调用是[操作系统](https://www.baidu.com/s?wd=%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)中的代码程序，是系统级程序。

|  |  |
| --- | --- |
| 函数库调用 | 系统调用 |
| 在所有的ANSI C编译器版本中，C库函数是**相同**的 | 各个操作系统的系统调用是**不同**的 |
| 它调用**函数库**中的一段程序（或函数） | 它调用**系统内核**的服务 |
| 与**用户程序**相联系 | 是**操作系统**的一个入口点 |
| **在用户地址空间执行** | **在内核地址空间执行** |
| 它的运行时间属于“**用户时间**” | 它的运行时间属于“**系统时间**” |
| 属于**过程调用，调用开销较小** | 需要在**用户空间和内核上下文环境间切换，开销较大** |
| 在C函数库libc中有大约300个函数 | 在UNIX中大约有90个系统调用 |
| 典型的C函数库调用：system fprintf malloc | 典型的系统调用：chdir fork write brk； |

库函数 printf 本质上是调用了系统调用 write() 函数，实现了终端信息的打印功能。

**\*C++调用C函数使用什么**

（#ifdef \_\_cplusplus extern "C"将实现与声明包含在里面）

**\*C语言调用C++函数**

（目前还没有使用过这个，我目前可以想到的是：C++比C多了函数重载，编译过程中会添加标识，所以的话，应该需要对C++的一个重载的接口进行转换）

**内联函数**

在类声明的内部声明或定义的成员函数叫做内联（INLINE)函数.

内联函数不能是虚函数 如果修饰内联函数 如果内联函数被virtual修饰，计算机会忽略inline使它变成存粹的虚函数。

内联函数具有一般函数的特性，他与一般函数不同之处只在于函数调用的处理。一般函数进行调用时要将程序执行权转到被调用函数中，然后在返回调用他的函数中；而内敛函数在调用时，是将调用表达式用内联函数体来替换。

**内联函数使用注意要点：**  
1.在内敛函数内不允许用循环语句和开关语句。若有则编译器将该函数视为普通函数那样昌盛函数调用代码，递归函数（自己调用自己）是不能作为内敛函数的。内联函数只适用于1——5行的小函数。对于含有很多语句的大函数，函数调用和返回的开销相对于来说微不足道，所以没不要用内联函数。  
2.内联函数的定义必须出现在内联函数第一次被调用之前。  
  
内联函数有两种实现方式：  
1.在类声明的内部声明，而在类声明外部定义叫做**显式内联函数**，如：  
　　class display  
　　{   
　　int t;  
　　public:  
　　void output(void)  
　　}  
　　display object;  
　　inline void display::output(void)  
　　{  
　　cout << "i is " << i <<"\n";  
　　}  
  
2.在类声明的内部定义，叫做**隐式内联函数**，如：  
　　class display  
　　{   
　　int t;  
　　public:  
　　inline void

**内联函数和define的区别**

（1）内联函数在运行时可调试，而宏定义不可以;

（2）编译器会对内联函数的参数类型做安全检查或自动类型转换（同普通函数），而宏定义则不会；     
（3）内联函数可以访问类的成员变量，宏定义则不能；

（4）在类中声明同时定义的成员函数，自动转化为内联函数。

**公共使用的内联函数要定义于头文件里**

因为编译时编译单元之间互不知道，如果内联被定义于.cpp文件中，编译其他使用该函数的编译单元的时候没有办法找到函数的定义，因此无法对函数进行展开（内联函数不展开，即不采用在使用处标记函数代码再跳转的方式，而是直接将代码嵌入）。所以如果内联函数定义于.cpp里，那么就只有这个.cpp文件能使用它。

.h中的inline 函数可以被多个cpp包含而不造成符号冲突，因为它会被直接嵌入到调用的地方，内部联结不形成外部符号,对外不可见。

**malloc和new**

**malloc/free：**

void \* malloc(size\_t size);

void free( void \* memblock );

int \*p1 = (int \*) malloc(sizeof(int) \* length);

为什么free 函数不象malloc 函数那样复杂呢？这是因为指针p 的类型以及它所指的内存的容

量事先都是知道的，语句free(p)能正确地释放内存。如果p 是NULL 指针，那么free对p 无论操作多少次都不会出问题。如果p 不是NULL 指针，那么free 对p连续操作两次就会导致程序运行错误。

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。  
1、malloc 返回值的类型是void \*，所以在调用malloc 时要显式地进行类型转换，将void \* 转换成所需要的指针类型。  
2、 malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。

**new/delete：**

int \*p2 = new int[length];

new 内置了sizeof、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，**new 在**

**创建动态对象**的同时完成了初始化工作。如果对象有多个构造函数，那么new 的语句也可以有多种形式。

如果用new 创建对象数组，那么只能使用对象的无参数构造函数。例如  
Obj \*objects = new Obj[100];       // 创建100 个动态对象  
不能写成  
Obj \*objects = new Obj[100](1);      // 创建100 个动态对象的同时赋初值1

在用delete 释放对象数组时，留意不要丢了符号‘[]’。例如  
delete []objects; // 正确的用法  
delete objects; // 错误的用法  
后者相当于delete objects[0]，漏掉了另外99 个对象。

**为什么要有new/delete？**

对于非内部数据类的对象而言，光用maloc/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malloc/free。因此C++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。

**为什么还保留malloc/free？**

既然new/delete的功能完全覆盖了malloc/free，为什么C++还保留malloc/free呢？因为C++程序经常要调用C函数，而C程序只能用malloc/free管理动态内存。如果用free释放“new创建的动态对象”，那么该对象因无法执行析构函数而可能导致程序出错。如果用delete释放“malloc申请的动态内存”，理论上讲程序不会出错，但是该程序的可读性很差。所以new/delete、malloc/free必须配对使用。

**不同点：**

1. 属性：malloc与free是C++/C 语言的标准库函数，需要头文件支持。new/delete 是C++的

运算符，需要编译器支持。

1. 参数：使用new操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行

计算。而malloc则需要手工计算字节数，显式地指出所需内存的尺寸。

1. 返回类型：new操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无

须进行类型转换，故new是符合类型安全性的操作符。而malloc内存分配成功则是返回void \* ，需要通过强制类型转换将void\*指针转换成我们需要的类型。

1. 分配失败：new内存分配失败时，会抛出bac\_alloc异常。malloc分配内存失败时返回NULL。
2. 自定义类型（**面向对象的区别**）：new会先调用operator new函数，申请足够的内存（通常底

层使用malloc实现）。然后调用类型的构造函数constructor，初始化成员变量，最后返回自定义类型指针。delete先调用析构函数，然后调用operator delete函数（调用destructor）释放内存（通常底层使用free实现）。malloc/free是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

1. 重载：C++允许重载new/delete操作符，特别的，布局new的就不需要为对象分配内存，而

是指定了一个地址作为内存起始区域，new在这段内存上为对象调用构造函数完成初始化工作，并返回此地址。而malloc不允许重载。

1. 内存区域：new操作符从自由存储区（free store）上为对象动态分配内存空间，而malloc函

数从堆上动态分配内存。自由存储区是C++基于new操作符的一个抽象概念，凡是通过new操作符进行内存申请，该内存即为自由存储区。而堆是操作系统中的术语，是操作系统所维护的一块特殊内存，用于程序的内存动态分配，C语言使用malloc从堆上分配内存，使用free释放已分配的对应内存。自由存储区不等于堆，如上所述，布局new就可以不位于堆中。

1. 安全性：

new是类型安全的，而malloc不是，比如：

int\* p = new float[2]; // 编译时指出错误；

int\* p = malloc(2\*sizeof(float)); // 编译时无法指出错误。

new在不重载空间分配器的前提下，它和malloc都是线程安全的。malloc函数线程安全但是

不可重入的，因为malloc函数在 用户空间 要自己管理 各进程共享的 内存链表，由于有共享资源访问，本身会造成线程不安全。为了做到线程安全，需要加锁进行保护。同时这个锁必须是递归锁，因为如果当程序调用malloc函数时收到信号，在 信号处理函数 里再调用malloc函数，如果使用一般的锁就会造成死锁（信号处理函数中断了原程序的执行），所以要使用递归锁。malloc创建的堆应该是是serialized的，能阻止多线程的同时分配/释放操作。但不能阻止两个线程同时修改堆里面的同一个数据。那个需要自己用关键区、信号量、互斥量等方式同步。

虽然使用递归锁能够保证malloc函数的线程安全性，但是不能保证它的可重入性。按上面的场景，程序调用malloc函数时收到信号，在信号处理函数里再调用malloc函数就可能破坏共享的内存链表等资源，因而是不可重入的。

至于malloc函数访问内核的共享数据结构可以正常的加锁保护，因为一个进程程调用malloc函数进入内核时，必须等到返回用户空间前夕才能执行信号处理函数，这时内核数据结构已经访问完成，内核锁已释放，所以不会有问题。

在不重载的情况下new默认调用malloc，malloc线程安全所以它也安全。但如果重载了，就不一定怎么实现，也不知道是否安全。所以严谨的说“在不重载空间分配器”的前提下，也就是默认调用malloc的版本是线程安全的。

**线程安全：**需要解决多个线程调用函数时访问共享资源的冲突

**可重入：**需要不在函数内部使用静态或全局数据，不返回静态或全局数据，也不调用不可重入函数。

**memcpy/strcpy/memset**

**函数原型：**

char \*strcpy(char \*dest,const char \* src) // 实现src到dest的复制

void \*memcpy(void \*desc,const void \* src,size\_t size)

void \*memset(void \*s,int c,size\_t n)

void是一个通用类型，使用的时候可以用int、char等等。

**1.memcpy：**

memcpy(dest,src,sizeof(src));

这里求长度一定要用sizeof，否则字符串的结束符不会被算进去

memcpy用来做内存拷贝，可以拷贝任何数据类型的对象并指定拷贝数据的长度：

char a[100],b[50];

memcpy(b, a, sizeof(b));

**2.strcpy：**

拷贝字符串，遇到'\0'时结束拷贝。

## strcpy在栈里拷贝到一个字符数组会出现的情况。

strcpy拷贝过程判断'\0'，字符数组没有该字符时，会把后面地址的数据进行覆盖

**strcpy和memcpy的区别：**1、复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意内容，例如字符数组、整型、结构体、类等。  
2、复制的方法不同。strcpy不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy则是根据其第3个参数决定复制的长度。  
3、用途不同。通常在复制字符串时用strcpy，而需要复制其他类型数据时则一般用memcpy

**3.memset**：

memset() 函数用来对一段内存空间全部设置为某个字符，一般用在对定义的字符串进行**初始化**为‘ ’或‘\0’，常用于内存空间初始化：

char str[100];   
memset(str, '\0', sizeof(str)); //将已开辟内存空间 str 的首100个字节的值设为值0。

另外比较方便的是对结构体的操作，memset可以方便的**清空**一个结构类型的变量或数组。比如有结构体struct sample\_strcut  stTest，一般清空结构体的话得用如下方式：

struct sample\_struct

{

char csName[16];

int iSeq;

int iType;

};  
  
stTest.csName[0]='\0';  
stTest.iSeq=0;  
stTest.iType=0;

而如果用memset就非常方便了：

  memset(&stTest,0,sizeof(struct sample\_struct));

**memcmp不能判断结构体相等**

未初始化的全局变量，static变量，编译器会自动初始化为0。但是局部变量的值是不确定的，必须先赋值再使用。比如char a后面3个字节不知道原本内存中是什么数据，那么A和B是没法进行比较是否相等的

**构造/析构函数**

## 拷贝构造函数什么时候用

拷贝构造函数调用的几种情况：   
1. 当用类的一个对象去初始化该类的另一个对象（或引用）时系统自动调用拷贝构造函数实现拷贝赋值。   
2. 若函数的形参为类对象，调用函数时，实参赋值给形参，系统自动调用拷贝构造函数。   
3. 当函数的返回值是类对象时，系统自动调用拷贝构造函数。

1当用类的一个对象初始化该类的另一个对象时.例如:

C/C++ code

int main()

{

point A(1,2);

point B(A);//用对象A初始化对象B,拷贝构造函数被调用.

}

2 如果函数的形参是类的对象,调用函数时,进行形参和实参结合时. 

C/C++ code

void f(point p)

{

}

main()

{

point A(1,2);

f(A);//函数的形参为类的对象时,当调用函数时,拷贝构造函数被调用.

}

3 如果函数的返回值是类的对象,函数执行完成返回调用者时. 

C/C++ code

point g()

{

point A(1,2);

return A;//函数的返回值是类的对象,返回函数值时,调用拷贝构造函数.

}

void main()

{

point B;

B = g();

}

4、需要产生一个临时类对象时。

## 拷贝构造函数形参不能用值只能引用（无限递归）

函数对于非引用的参数值而言，都会对其在栈内进行拷贝，而对类进行拷贝，又调用了拷贝构造函数，这样会**造成无限的调用，直至函数栈溢出。**

当一个对象需要以值方式传递时，编译器会生成代码调用它的拷贝构造函数以生成一个复本。如果类A的拷贝构造函数是以值方式传递一个类A对象作为参数的话，当 需要调用类A的拷贝构造函数时，需要以值方式传进一个A的对象作为实参；  
而以值方式传递需要调用类A的拷贝构造函数；结果就是调用类A的拷贝构造函数导 致又一次调用类A的拷贝构造函数，这就是一个无限递归。

## 构造/析构函数可以抛出异常吗？

不建议在构造函数中抛出异常，因为构造函数抛出异常时，析构函数将不会被执行,需要手动的去释放内存。

析构函数不能抛出异常，处理析构函数异常的正确方式是将异常封装在析构函数内部，而不是抛出异常，如下代码所示。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | ~ClassName(){      try{          do\_something();      }      catch{          //...      }  } |

原因如下：C++异常处理模型有责任处理那些因为出现异常而失效的对象，处理方式是调用这些失效对象的析构函数，释放掉它们占用的资源。如果析构函数再抛出异常，则会陷入无尽递归嵌套之中，因此这是不被允许的。存在一种冲突状态，程序将直接崩溃：异常的被称为“栈展开(stack unwinding)”的过程中时，从析构函数抛出异常，C++运行时系统会处于无法决断的境遇，因此C++语言担保，当处于这一点时，会调用 terminate()来杀死进程。

## [析构函数为什么用虚析构函数](https://www.cnblogs.com/liushui-sky/p/5824919.html)

C++中基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发[动态绑定](#_动态绑定（dynamically_bound）)，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。所以，为了防止这种情况的发生，C++中基类的析构函数应采用virtual虚析构函数。

### 动态绑定（dynamically bound）

又名后期绑定（late binding），绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；比如常见的，virtual函数是动态绑定，non-virtual函数是静态绑定，缺省参数值也是静态绑定。以virtual函数为例。当某个virtual函数通过指针或引用调用时，编译器产生的代码直到运行时才能确定应该调用哪个版本的函数。**被调用的函数是与绑定到指针或引用上的对象的动态类型相匹配的那一个**。需要注意的是：动态绑定只有当我们通过指针或引用调用虚函数时才会发生。

### 静态绑定（statically bound）

又名前期绑定（eraly binding），绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期。

**软拷贝硬拷贝**

印刷机、打印机、[复印机](https://www.baidu.com/s?wd=%E5%A4%8D%E5%8D%B0%E6%9C%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YLPjP9mWb1nWI-m1F9myDs0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3ErjR3Pj0sn1c)等得到的印件叫作硬拷贝，而把显示器等装置上面显示的图像、文字等叫作软拷贝。

软拷贝是相对于硬拷贝而言的，是指对非印刷形式的计算机文件的复制。

**深拷贝浅拷贝**

浅拷贝：只是增加了一个指针，指向已存在对象的内存。在多个对象指向一块空间的时候，释放一个空间会导致其他对象所使用的空间也被释放了，再次释放便会出现错误。

深拷贝：是增加了一个指针，并新开辟了一块空间，让指针指向这块新开辟的空间。

**unsigned char a=-1,printf("%d",a)输出-1**

unsigned int a＝-1  
说到底,a=0xffffffff【4294967295】  
而0xffffffff当做有符号数，就是-1.

1、在C语言中整型常量的数据类型默认为int，所以unsigned int a = -1中，-1的类型是int类型。实际上是有一个隐式转换，即将int类型转成unsigned int类型。 这个转换的意义不大，因为没有超出unsigned类型的表示范围，所以a的机器码依然是-1的机器码。

2、printf输出函数的[控制字符](https://www.baidu.com/s?wd=%E6%8E%A7%E5%88%B6%E5%AD%97%E7%AC%A6&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)是%d，即输出有符号数，所以结果是-1，如果要输出无符号数，请使用:%u

**int printf%s会出什么情况**

打印字符串。如成功，ret为打印字符数量。否则为一负值。

**printf的格式化输出怎么实现的**

printf函数中，最为核心的是两个函数，一个是vsprintf（用于转换格式），一个puts函数（用于输出字符）。如果我们把一个64位的整数，以%ld打印出来的时候，vsprintf就会把这个64位数拆分成两个32位的数，然后用puts函数分别把这两个数打印出来。所以两个%ld只能打印出一个64位的数，第二个64位的数根本就打印不出来。

说明:

（1）“格式控制”是用双撇号括起来的字符串，也称“转换控制字符串”，它包括两种信息：

　　①格式说明：由“%”和格式字符组成，它的作用是将输出的数据转换为指定的格式输出。

　　②普通字符，即需要原样输出的字符。

（2）“输出表列”是需要输出的一些数据，可以是表达式

（3）printf函数的一般形式可以表示为

　　printf(参数1，参数2，……，参数n)

　　功能是将参数2～参数n按参数1给定的格式输出

int printf(const char \*format,[argument])  
{  
  char printf\_buf[1024];  
  va\_list args;  
  int printed;  
  
  va\_start(args, fmt);  
  printed = vsprintf(printf\_buf, fmt, args);  
  va\_end(args);  
  
  puts(printf\_buf);  
  
  return printed;  
}

format 参数输出的格式，定义格式为：%[flags][width][.perc] [F|N|h|l]type

1.type 含义如下：

　　d 有符号10进制整数

　　i 有符号10进制整数

　　o 有符号8进制整数

　　u 无符号10进制整数

　　x 无符号的16进制数字，并以小写abcdef表示

　　X 无符号的16进制数字，并以大写ABCDEF表示

　　F/f 浮点数

　　E/e 用科学表示格式的浮点数

　g 使用%f和%e表示中的总的位数表示最短的来表示浮点数 G 同g格式，但表示为指数

　　c 单个字符

　　s 字符串

　　% 显示百分号本身

　　p 显示一个指针，near指针表示为：XXXX

　　far 指针表示为：XXXX：YYYY

　　n 相连参量应是一个指针，其中存放已写字符的个数

2.flags 规定输出格式，取值和含义如下：

　　无 右对齐，左边填充0和空格

　　- 左对齐，右边填充空格

　　+ 在数字前增加符号 + 或 -

　　一个空格 只对负数显示符号

　　# 当type=c,s,d,i,u时没有影响

　　type=o,x,X时，分别在数值前增加'0',"0x","0X"

　　type=e,E,f时，总是使用小数点

type=g,G时，除了数值为0外总是显示小数点

3.width 用于控制显示数值的宽度，取值和含义如下n(n=1,2,3...) 宽度至少为n位，不够以空格填充。0n(n=1,2,3...) 宽度至少为n位，不够左边以0填充 \* 格式列表中，下一个参数还是width

4.prec 用于控制小数点后面的位数，取值和含义如下：

　　无 按缺省精度显示

　　0 当type=d,i,o,u,x时，没有影响

　　type=e,E,f时，不显示小数点

　　n(n=1,2,3...) 当type=e,E,f时表示的最大小数位数

　　type=其他，表示显示的最大宽度 .\*

　　格式列表中，下一个参数还是width

5.F|N|h|l 表示指针是否是远指针或整数是否是长整数

　　F 远指针

　　n 近指针

　　h短整数或单精度浮点数

　　l 长整数或双精度浮点数

　　1.一般格式

　　printf(格式控制，输出表列）

　　例如：printf("i=%d,ch=%c/n",i,ch);

**printf("%s%d", s, d)的压栈情况，执行Printf函数时sp指针的变化。**

sp就是堆栈指针，直接移动即可，注意堆栈是从高到底，相加相减分清楚。

printf函数的压栈顺序是从右到左压栈，计算表达式值的顺序也是从右到左，由于输入数据类型的多样性，压栈的时候将数据或数据的地址mov到寄存器中，然后将寄存器压栈。由于参数按照从右向左顺序压栈，因此最开始的参数在最接近栈顶的位置，因此当采用不定个数参数时，第一个参数在栈中的位置肯定能知道，只要不定的参数个数能够根据第一个明确的参数确定下来，就可以使用不定参数，这也是C语言的一大特色。

**printf函数怎么传参，printf("%d%d",i)会发生什么，为什么**

 首先，变量将值传给计算机，计算机把它们放置到被称为堆栈（stack）的一块内存区域中来。计算机根据变量的类型而非转换说明符（like %c %f）把这些值放到堆栈中。然后，标识符根据读取数据的类型在stack中读取，并打印出来。

函数的堆栈映像图，反序入栈，根据控制字符串，检索参数个数和参数类型

printf("%d%d",i)会造成获取一个错误的参数，具体数据是不确定的

**main函数的参数（也就是argc、argv）**

（main之前有启动例程，启动例程从内存获取命令行参数和环境变量值，传递给main）

**如何获取main函数的返回值**

bash：**echo $?**

fork：wait族函数

**main之前之后可以运行什么代码**

（1）全局对象的构造函数会在main 函数之前执行，全局对象的析构函数会在main函数之后执行；  
（2）用atexit注册的函数也会在main之后执行。  
（3）一些全局变量、对象和静态变量、对象的空间分配和赋初值就是在执行main函数之前,而main函数执行完后，还要去执行一些诸如释放空间、释放资源使用权等操作

（4）进程启动后，要执行一些初始化代码（如设置环境变量等），然后跳转到main执行。

**pthread\_join和pthread\_detach两个函数的功能**

在任何一个时间点上，线程是可结合的（joinable）或者是分离的（detached）。一个可结合的线程能够被其他线程收回其资源和杀死。在被其他线程回收之前，它的存储器资源（例如栈）是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的，它的存储器资源在它终止时由系统自动释放。

默认情况下，线程被创建成可结合的。为了避免存储器泄漏，每个可结合线程都应该要么被显示地回收，即调用pthread\_join；要么通过调用pthread\_detach函数被分离。

**pthread\_detach**

创建一个线程默认的状态是joinable, 如果一个线程结束运行但没有被join,则它的状态类似于进程中的Zombie Process,即还有一部分资源没有被回收（退出状态码），所以创建线程者应该调用pthread\_join来等待线程运行结束，并可得到线程的退出代码，回收其资源（类似于wait,waitpid)

但是调用pthread\_join(pthread\_id)后，如果该线程没有运行结束，调用者会被阻塞，在有些情况下我们并不希望如此，比如在Web服务器中当主线程为每个新来的链接创建一个子线程进行处理的时候，主线程并不希望因为调用pthread\_join而阻塞（因为还要继续处理之后到来的链接），这时可以在子线程中加入代码   
pthread\_detach(pthread\_self())   
或者父线程调用   
pthread\_detach(thread\_id)（非阻塞，可立即返回）   
这将该子线程的状态设置为detached,则该线程运行结束后会自动释放所有资源。   
  
**pthread\_join**

调用pthread\_join的线程会阻塞，直到指定的线程返回，调用了pthread\_exit，或者被取消。如果线程简单的返回，那么rval\_ptr被设置成线程的返回值，参见范例1；如果调用pthread\_exit则可将一个无类型指针返回，在pthread\_join中对其进行访问，参见范例2；如果线程被取消，rval\_ptr被设置成PTHREAD\_CANCELED。如果我们不关心线程的返回值，那么我们可以把rval\_ptr设置为NULL。

**C++类内可以定义 引用数据成员**

必须通过成员函数 初始化列表初始化。  
C++中除了在类中简单声明数据成员外，还可创建static数据成员，const成员，引用成员，const

引用等。

## **静态数据成员**

应用背景：有时没必要让所有类的对象都包含某个变量的副本，或者说你需要一个只对类有意义的数据成员，而不适用于针对某个对象。比如设计一个电子表格的类，每个电子表格都有一个唯一的ID，我们可以为类设计一个从0开始的计数器，这样每个对象可从这个计数器得到自身的ID，但显然没必要为每个对象包含这个计数器，这样的麻烦是因为你还得让计数器保持同步。解决的办法是使用静态数据成员。

静态数据成员属于类，而不具体属于类的某个实例，可将静态数据成员当做属于类的全局变量，所有该类的实例对象都只有和可见这么一份。比如：  
class MyClass{  
 //......  
 protected: static int counter\_=0;  
}  
  
注意：在C++11中是可以在类的定义中初始化静态成员的。下面介绍具体的两个用法   
**1.在类的方法内访问静态数据成员**在类的方法内部可像使用普通数据成员一样使用静态数据成员。比如：  
类的声明部分：  
class MyClass{  
 public: //.....  
 int get\_id() const;

protected: //.....  
 static int count=0;

int id\_;  
};  
  
类的实现部分：  
MyClass::MyClass(int in\_width,int int height)：  
width\_(in\_width),height\_(height)  
{  
 id\_=count++; //......}  
  
也就是说在构造函数中可以访问静态数据成员counter就像访问普通数据成员一样。  
**2.在方法外访问静态数据成员**  
访问控制限定符也适用于静态数据成员。上面的counter是protected的，因此不能在类外访问，若是公有的则可，具体也是用作用域解析运算符指出这个变量是类的一部分，例如：  
int i=MyClass::count;  
  
当然，并不建议这么用，而是为类提供get/set方法来授权访问权限。如果想要访问静态数据成员，应该提供静态的get/set方法。

## **常量数据成员**

类的数据成员还可以声明为cosnt，这意味着在创建并初始化之后数据成员的值不能再改变。然而在对象层次上常量通常没有意义，因此常量数据成员通常也是静态的。如果某个常量只适用于类，应该使用静态常量数据成员而不是全局常量。例如设计一个表格类时，我们可能会指定表格表格的最大高度和宽度，这个值可设置成一个静态常量数据成员，当用户想要创建的表格高度或宽度大于该最大值时使用最大值，代码片段如下：  
class SpreedSheet{  
 public: //......  
 static const int kMaxHeight=100; static const int kMaxWidth=100;  
};  
  
为什么说在对象层次上常量通常没有意义呢，私下认为，对象本身就是个变量，但对象里面却包含常量很不可思议，是不？换句话说：变得东西里含有不变的成分，那么不变的那部分还有什么意义呢。  
有了上述类中的静态常量，我们就可以在类的构造函数中这样使用这个新的常量了。  
SpreedSheet::SpreadSheet(int width,int hegiy):  
 width\_(width<<span>kMaxWidth?width:kMaxWidth), height(height<<span>kMaxHeight?height:kMaxHight)  
{ id\_=count++;  
 //......}  
  
这样当输入宽度或高度大于最大值时，会自动使用最大高度和宽度构造对象。由于kMaxHeight以及kMaxWith是公有的，因此可在程序的任何位置对其进行访问。但必须带作用域解析符，以说明该变量是哪个类的一部分。例：  
cout<<span><<span>SpreadSheet::kMaxHeight<<endl;

## **引用数据成员（循环引用问题）**

考虑一个架构问题：电子表格如何与应用程序通信？至少我们应该让应用程序存储电子表格，电子表格也该存储应用程序对象的引用。即具体来说SpreadSheet类必须知道SpreadSheetApplication类，SpreadsheetApplication类也必须知道SpreadSheet类，即通信是双方的，必须相互认识才能有效传递消息么。那么这里就存在一个问题，我们在类的实现里需要相互#inlcude，这就是传说中的循环引用问题，相互#include是不能解决问题的。解决方案是在其中一个头文件中使用前置声明。比如下面含前置声明的类的定义：  
class SpreadsheetApplication; //前置声明classSpreadsheet{  
 public: Spreadsheet(int in\_width,Int in\_height,SpreadsheetApplication**&** the\_app); //......  
 protected: //...... SpreadsheetApplication& the\_app\_;  
};  
  
这里类的定义将一个SpreadsheetApplicatin引用作为数据成员添加进来，建议使用引用，因为引用总是引用一个SpreadsheetApplication，而指针无法保证这点。在类的构造函数中，每个Spreeadsheet都得到一个应用程序的引用。若不引用某些事物，引用将无法存在，因此在构造函数中必须给the\_app\_一个值。  
Spreadsheet::Spreadsheet(int in\_width,int in\_height,SpreadsheetApplication & the\_app): widthe\_(in\_width<<span>kMaxWidth?in\_width:kMaxWidth),height\_(in\_height<<span>kMaxHeight?in\_height:kMaxHight),the\_app\_(the\_app){}  
  
也就是说类中的引用数据成员必须在类的所有构造函数中初始化它。包括复制构造函数：  
Spreeadsheet::Spreadsheet(const Spreadsheet& src): the\_app\_(src.the\_app\_){}  
  
在初始化一个引用之后，不能改变它的引用对象。因此无需在赋值运算符中对引用赋值。

## **常量引用数据成员**

和普通引用可以引用常量对象一样，引用成员也可以引用常量对象。可以这么看，常量不过是普通的一个特殊而已。例如：  
class Spreadsheet{  
 public:  
 Spreadsheet(int in\_width,int in\_height,const SpreadsheetApplication& the\_app); //......  
 protected: //......  
 const SpreadsheetApplication& the\_app\_;  
};

常量引用和非常量引用的一个重要区别是：比如这里的常量引用SpreadsheetApplication的数据成员 the\_app\_只能调用SpreadsheetApplication对象上的常量方法。如果试图通过常量引用调用非常量方法，编译报错。

**访问类private成员的方法**

通过一个对象调用其public成员函数，此成员函数可以访问到自己的或者同类其他对象的public/private/protected数据成员和成员函数(类的所有对象共用)，而且还需要指明是哪个对象的数据成员(调用函数的对象自己的成员不用指明，因为有this指针；其他对象的数据成员可以通过引用或指针间接指明)

c++中类的private成员对外是不可见的，以下方法可以突破private成员的访问权限。

// accessPrivateMember.h

class A

{

private:

    int mPrivate;

    int nPrivate;

public:

    A(): mPrivate(3), nPrivate(4){}

**template<typename T> void func(const T &t){}**

    const int GetValueN()

    {

        return nPrivate;

    }

};

**1、操作指针修改内存数据**

#include <iostream>

#include "accessPrivateMember.h"

int main()

{

    A obj = A();

    int tmp = 7;

**int \*ptr = (int \*)(&obj);**

**\*(ptr+1) = tmp;**

    std::cout<<"nPrivate: "<<obj.GetValueN()<<std::endl;

    return 0;

}

**2、使用友元声明**

在类内声明一个友元函数，在类外进行定义。

#include <iostream>

class A

{

private:

**friend void hiJack(A &);**

    int mPrivate;

    int nPrivate;

public:

    A(): mPrivate(3), nPrivate(4){}

    template<typename T> void func(const T &t){}

    const int GetValueN()

    {

        return nPrivate;

    }

};

**void hiJack(A &a)**

**{**

**a.nPrivate = 2;**

**}**

int main()

{

    A obj = A();

    hiJack(obj);

    std::cout<<"nPrivate: "<<obj.GetValueN()<<std::endl;

    return 0;

}

**3、使用模板**

类A中定义了一个模板函数func，在类外加一个模板函数，导致模板推演的过程中多出一个自己写的并且加入了备选组中，相当于多了一个重载。

类外函数的参数是在匿名空间中定义的特定类型，所以避免扰乱原本该函数的功能。

#include <iostream>

#include "accessPrivateMember.h"

**namespace {struct B{};}   // struct member default public**

**template<> void A::func(const B &)**

{

    nPrivate = 2;

}

int main()

{

    A obj;

    obj.func(B());

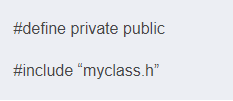
    std::cout<<"nPrivate: "<<obj.GetValueN()<<std::endl;

    return 0;

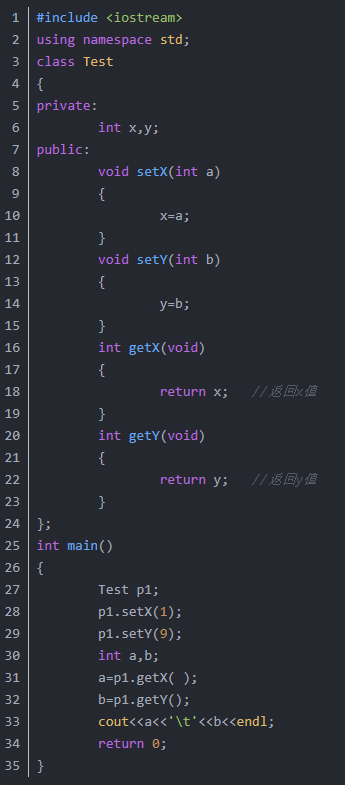
}

**4、#define private public**

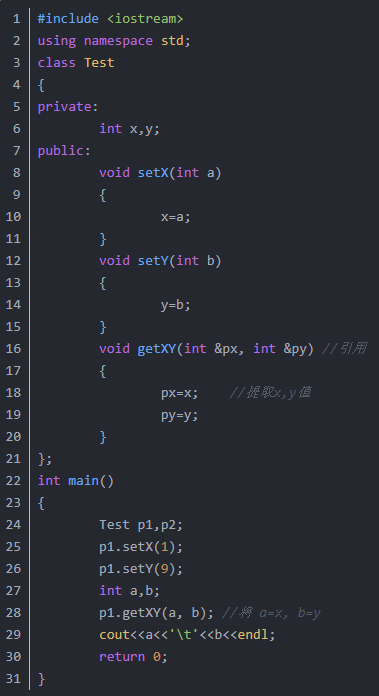
使用#define把private定义为public，这样类中定义的private起始就是public，不过这要放在#include头文件之前。不推荐使用这种方法。



**5、通过公共函数访问私有数据成员**



**6、通过引用访问私有数据成员**



**如何测试私有成员**

1、最简单的，使用#define private public粗暴地将private变成public，不过这要放在#include头文件之前。如：

#define private public

#include “myclass.h”

2、使用friend。这个会相对友好一点，但是却需要修改原有的代码。

3、使用一些旁门左道，例如：

a.在知道类对象内存布局的情况下，通过地址偏移直接操作地址访问类的数据成员；

b.创建一个具有完全相同内存布局的Dummy类，然后进行强制转换并访问数据。例如



**抽象类和接口的区别**

抽象类用来抽象自然界一些具有相似性质和行为的对象。

而接口用来抽象行为的标准和规范，用来告诉接口的实现者必要按照某种规范去完成某个功能。

**抽象类：**抽象类是特殊的类，只是不能被实例化（将定义了纯虚函数的类称为抽象类）；除此以外，具有类的其他特性；重要的是抽象类可以包括抽象方法，这是普通类所不能的，但同时也能包括普通的方法。抽象方法只能声明于抽象类中，且不包含任何实现，派生类必须覆盖它们。另外，抽象类可以派生自一个抽象类，可以覆盖基类的抽象方法也可以不覆盖，如果不覆盖，则其派生类必须覆盖它们。

虽然不能定义抽象类的实例，但是可以定义它的指针，并且指向抽象类的指针实际上在赋值时 是指向其继承类的实例化对象的，这样通过统一的使用该指针可以很好的封装不同子类的实现过程，这正是用抽象类实现接口的重点所在。

**接口：**接口是一个概念。它在C++中用抽象类来实现。

接口是引用类型的，类似于类,和抽象类的相似之处有三点：

1、不能实例化；  
2、包含未实现的方法声明；  
3、派生类必须实现未实现的方法，抽象类是抽象方法，接口则是所有成员（不仅是方法包括其他成员）；

另外，接口有如下特性：接口除了可以包含方法之外，还可以包含属性、索引器、事件，而且这些成员都被定义为公有的。除此之外，不能包含任何其他的成员，例如：常量、域、构造函数、析构函数、静态成员。一个类可以直接继承多个接口，但只能直接继承一个类（包括抽象类）。



**抽象类和接口的区别：**  
1.概念：接口是对动作的抽象，抽象类是对根源的抽象。抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。

2.抽象类在定义类型方法的时候，可以给出方法的实现部分，也可以不给出；而对于接口来说，其中所定义的方法都不能给出实现部分。  
3.继承类对于两者所涉及方法的实现是不同的。继承类对于抽象类所定义的抽象方法，可以不用重写，也就是说，可以延用抽象类的方法；而对于接口类所定义的方法或者属性来说，在继承类中必须要给出相应的方法和属性实现。    
4.接口可以用于支持回调,而继承并不具备这个特点.       
5.抽象类不能被密封，一个类一次可以实现若干个接口,但是只能扩展一个(抽象类)父类。     
6.抽象类实现的具体方法默认为虚的，但实现接口的类中的接口方法却默认为非虚的，当然您也可以声明为虚的。  
7.接口与非抽象类类似，抽象类也必须为在该类的基类列表中列出的接口的所有成员提供它自己的实现。但是，允许抽象类将接口方法映射到抽象方法上。     
8.抽象类实现了oop中的一个原则，把可变的与不可变的分离。抽象类和接口就是定义为不可变的，而把可变的座位子类去实现。     
9.好的接口定义应该是具有专一功能性的，而不是多功能的，否则造成接口污染。（如果一个类只是为实现了这个接口的中一个功能，而但是却不得不去实现接口中的其他方法，就叫接口污染。）  
10.尽量避免使用继承来实现组建功能，而是使用黑箱复用，即对象组合。因为继承的层次增多，造成最直接的后果就是当你调用这个类群中某一类，就必须把他们全部加载到栈中！后果可想而知。（结合堆栈原理理解）。同时，有心的朋友可以留意到微软在构建一个类时，很多时候用到了对象组合的方法。比如asp.net中，Page类，有ServerRequest等属性，但其实他们都是某个类的对象。使用Page类的这个对象来调用另外的类的方法和属性，这个是非常基本的一个设计原则。     
11.如果抽象类实现接口，则可以把接口中方法映射到抽象类中作为抽象方法而不必实现，而在抽象类的子类中实现接口中方法.

**类和对象的关系**

**（1）对象：**对象是人们要进行研究的任何事物，它不仅能表示具体的事物，还能表示抽象的规则、计划或事件。对象具有状态，一个对象用数据值来描述它的状态。对象还有操作，用于改变对象的状态，对象及其操作就是对象的行为。对象实现了数据和操作的结合，使数据和操作封装于对象的统一体中。

**（2）类：**具有相同特性（数据元素）和行为（功能）的对象的抽象就是类。因此，对象的抽象是类，类的具体化就是对象，也可以说类的实例是对象，类实际上就是一种数据类型。类具有属性，它是对象的状态的抽象，用数据结构来描述类的属性。类具有操作，它是对象的行为的抽象，用操作名和实现该操作的方法来描述。

类的实例化的结果就是对象，而对对象的抽象就是类，类描述了一组有相同特性（属性）和相同行为的对象。

**类和结构体的区别**

首先类是C++中面向对象独有的，但是C和C++中都有结构体，下面我们来看一下C和C++中结构体的区别。这里主要从封装、多态、继承、封装和访问权限几个方面来说。  
**1、C和C++中结构体的区别**  
（1）多态：C的结构体内不允许有函数存在，但是有默认的构造函数，就是把所有的成员属性设置为0，不能自定义。但是C的结构体是没有构造函数、析构函数、和this指针的，所以没有多态而言；C++允许有内部成员函数，且允许该函数是虚函数可以多态。  
（2）继承：C语言的结构体是不可以继承的，C++的结构体是可以从其他的结构体或者类继承过来的，和类一样，实现了代码的复用。  
（3）封装：C的结构体只是把数据变量给包裹起来了，并不涉及算法，是一种“复合类型”，其功能基本与int ，double的用法相同，它主要解决多类型问题。而C++中是把数据变量及对这些数据变量的相关算法给封装起来，并且给对这些数据和类不同的访问权限。  
（4）访问权限：C的结构体对内部成员变量的访问权限只能是public，而C++允许public,protected,private三种。  
  以上三点都是表面的区别，实际区别就是面向过程和面向对象编程思路的区别。  
**2、C++的结构体和C++类的区别**  
 主要是访问权限的区别：  
（1）C++结构体内部成员变量及成员函数默认的访问级别是public,而c++类的内部成员变量及成员函数的默认访问级别是private。

（2）C++结构体的继承默认是public，而c++类的继承默认是private。

（3）C#中如果没有标明成员函数或者成员变量的访问权限级别，struct和class都是private。区别是struct定义的是值类型，值类型的实例在栈上分配内存；class定义的是引用类型，引用类型的实例在堆上分配内存。

**修改结构体对齐数（#param pack）**

**空结构体sizeof返回值1**

sizeof(空类/空结构体) = 1；   
　　空类，没有任何成员变量或函数，即没有存储任何内容；但是由于空类仍然可以实例化，例如：

ClassA A;

cout<<"sizeof(A): "<<sizeof(A)<<endl;

一个类能够实例化，编译器就需给它分配内存空间，来指示类实例的地址。这里编译器默认分配了一个字节（如：char），以便标记可能初始化的类实例，同时使空类占用的空间也最少（即1字节）。

**比较两个结构体相等（重载==）**

重载了 “==” 操作符

struct foo {

int a;

int b;

**bool operator==(const foo& rhs) // 操作运算符重载**

**{**

**return( a == rhs.a) && (b == rhs.b);**

**}**

};

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

{

foo a,b;

a.a = 1;a.b = 2;

b.a = 2;b.a = 1;

if (a == b)

{

cout<<"相同"<<endl;

}

else

cout<<"不同"<<endl;

system("pause");

return 0;

}

**Union**

union 维护足够的空间来置放多个数据成员中的**“一种”**，**而不是为每一个数据成员配置空间**，在union 中所有的数据成员共用一个空间，同一时间只能储存其中一个数据成员，所有的数据成员具有相同的起始地址。

**一个union 只配置一个足够大的空间以来容纳最大长度的数据成员。**

**---------\*封装多态继承\*----------**

**封装**

封装就是将抽象得到的数据和行为相结合，形成一个有机的整体，也就是将数据与操作数据的源代码进行有机的结合，形成类，其中数据和函数都是类的成员，目的在于将对象的使用者和设计者分开，以提高软件的可维护性和可修改性。

特性：

1. 结合性，即是将属性和方法结合

2. 信息隐蔽性，利用接口机制隐蔽内部实现细节，只留下接口给外界调用

3. 实现代码重用

**多态**

关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”。

1. **class** A
2. {
3. **public**:
4. **void** foo()
5. {
6. printf("1\n");
7. }
8. **virtual** **void** fun()
9. {
10. printf("2\n");
11. }
12. };
13. **class** B : **public** A
14. {
15. **public**:
16. **void** foo()
17. {
18. printf("3\n");
19. }
20. **void** fun()
21. {
22. printf("4\n");
23. }
24. };
25. **int** main(**void**)
26. {
27. A a;
28. B b;
29. A \*p = &a;
30. p->foo();  1
31. p->fun();  2
32. p = &b;
33. p->foo();  1
34. p->fun();  4
35. **return** 0;
36. }

## 动态多态

以[显式接口](#_显式接口与隐式接口)和运行期多态（虚函数）实现动态多态。

对于相关的对象类型，确定它们之间的一个共同功能集，然后在基类中，把这些共同的功能声明为多个公共的虚函数接口。各个子类重写这些虚函数，以完成具体的功能。操作函数通过指向基类的引用或指针来操作这些对象，由于有了虚函数，因此动态多态是在运行时完成的，也可以叫做运行期多态。

如果我们在子类中定义了从父类继承来的虚函数，对于父类来说情况是不变的，对于子类来说它的虚函数表与之前的虚函数表是一样的，但是此时子类定义了自己的（从父类那继承来的）相应函数，所以它的虚函数表当中管于这个函数的指针就会覆盖掉原有的指向父类函数的指针的值，换句话说就是指向了自己定义的相应函数，这样如果用父类的指针，指向子类的对象，就会通过子类对象当中的虚函数表指针找到子类的虚函数表，从而通过子类的虚函数表找到子类的相应虚函数地址，而此时的地址已经是该函数自己定义的虚函数入口地址，而不是父类的相应虚函数入口地址，所以执行的将会是子类当中的虚函数。这就是多态的原理。

运行期多态通过虚函数发生于运行期。使用时在父类中写一个虚函数，在子类中分别重写，用这个父类指针调用这个虚函数，它实际上会调用各自子类重写的虚函数。运行期多态的实现依赖于[虚函数](#_虚函数)机制。当某个类声明了虚函数时，编译器将为该类对象安插一个虚函数表指针，并为该类设置一张唯一的[虚函数表](#_虚函数表)，虚函数表中存放的是该类虚函数地址。运行期间通过虚函数表指针与虚函数表去确定该类虚函数的真正实现。

优点：

1、OO设计中重要的特性，对客观世界直觉认识。

2、能够处理同一个继承体系下的异质类集合。运行期多态的优势还在于它使处理异质对象集合称为可能：

缺点：

1、运行期间进行虚函数绑定，提高了程序运行开销。

2、庞大的类继承层次，对接口的修改易影响类继承层次。

3、由于虚函数在运行期在确定，所以编译器无法对虚函数进行优化。

4、虚表指针增大了对象体积，类也多了一张虚函数表，当然，这是理所应当值得付出的资源消耗，列为缺点有点勉强。

### 虚函数

用virtual去修饰成员函数使其成为虚函数。作用主要是实现了多态的机制。有了虚函数每个子类在继承父类的基本属性外，同时还具有不同于其它子类的特点，这样代码的重用性提高了。virtual关键字可以修饰普通的成员函数，也可以修饰析构函数，但并不是没有限制。

virtual在函数中的使用限制：

1普通函数不能是虚函数，也就是说这个函数必须是某一个类的成员函数，不可以是一个全局函数，否则会导致编译错误。

2静态成员函数不能是虚函数 static成员函数是和类同生共处的，他不属于任何对象，使用virtual也将导致错误。

3内联函数不能是虚函数 如果修饰内联函数 如果内联函数被virtual修饰，计算机会忽略inline使它变成存粹的虚函数。

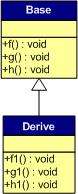
4构造函数不能是虚函数，否则会出现编译错误。

### 虚函数表

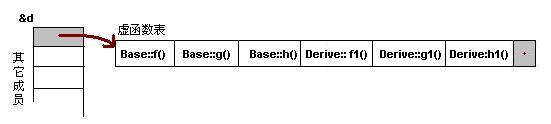
<https://blog.csdn.net/haoel/article/details/1948051/>

虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。C++的编译器应该是保证虚函数表的指针存在于对象实例中最前面的位置（这是为了保证取到虚函数表的有最高的性能——如果有多层继承或是多重继承的情况下）。 这意味着我们通过对象实例的地址得到这张虚函数表，然后就可以遍历其中函数指针，并调用相应的函数。

**一般继承（无虚函数覆盖）**



子类没有重载任何父类的函数。那么，在派生类的实例中，其虚函数表

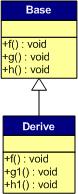


我们可以看到下面几点：

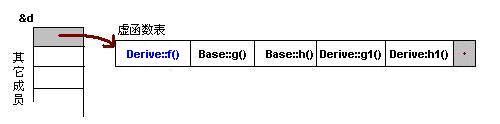
1）虚函数按照其声明顺序放于表中。

2）父类的虚函数在子类的虚函数前面。

**一般继承（有虚函数覆盖）**



如果子类中有虚函数重载了父类的虚函数

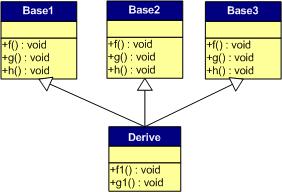


我们从表中可以看到下面几点，

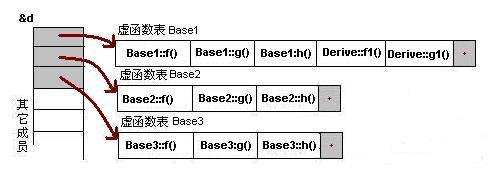
1）覆盖的f()函数被放到了虚表中原来父类虚函数的位置。

2）没有被覆盖的函数依旧。

**多重继承（无虚函数覆盖）**



子类并没有覆盖父类的函数。



我们可以看到：

1）每个父类都有自己的虚表。

2）子类的成员函数被放到了第一个父类的表中。（所谓的第一个父类是按照声明顺序来判断的）

#### 安全性

**一、通过父类型的指针访问子类自己的虚函数**

我们知道，子类没有重载父类的虚函数是一件毫无意义的事情。因为多态也是要基于函数重载的。虽然在上面的图中我们可以看到Base1的虚表中有Derive的虚函数，但我们根本不可能使用下面的语句来调用子类的自有虚函数：

          Base1 \*b1 = new Derive();

          b1->f1();  //编译出错

任何妄图使用父类指针想调用子类中的未**覆盖父类的成员函数**的行为都会被编译器视为非法，所以，这样的程序根本无法编译通过。但在运行时，我们可以通过指针的方式访问虚函数表来达到违反C++语义的行为。

**二、访问non-public的虚函数**

另外，如果父类的虚函数是private或是protected的，但这些非public的虚函数同样会存在于虚函数表中，所以，我们同样可以使用访问虚函数表的方式来访问这些non-public的虚函数，这是很容易做到的。

如：

class Base {

    private:

            virtual void f() { cout << "Base::f" << endl; }

};

class Derive : public Base{

};

typedef void(\*Fun)(void);

void main() {

    Derive d;

    Fun  pFun = (Fun)\*((int\*)\*(int\*)(&d)+0);

    pFun();

}

### 虚函数和纯虚函数

**虚函数：**

C++的虚函数主要作用是“**运行时多态**”，父类中提供虚函数的实现，**为子类提供默认的函数实现**。**子类可以重写父类的虚函数实现子类的特殊化。**

**纯虚函数：**

C++中包含纯虚函数的类，被称为是“抽象类”。**抽象类不能使用new出对象，只有实现了这个纯虚函数的子类才能new出对象。**C++中的纯虚函数更像是“只提供申明，没有实现”，是对子类的约束，是“**接口继承**”。C++中的纯虚函数也是一种“运行时多态”。

**用法区别：**

1.虚函数和纯虚函数可以定义在同一个类(class)中，含有纯虚函数的类被称为抽象类(abstract class)，而只含有虚函数的类(class)不能被称为抽象类(abstract class)。

2.虚函数可以被直接使用，也可以被子类(sub class)重载以后以多态的形式调用，而纯虚函数必须在子类(sub class)中实现该函数才可以使用，因为纯虚函数在基类(base class)只有声明而没有定义。

3.虚函数和纯虚函数都可以在子类(sub class)中被重载，以多态的形式被调用。

4.虚函数和纯虚函数通常存在于抽象基类(abstract base class -ABC)之中，被继承的子类重载，目的是提供一个统一的接口。

5.虚函数的定义形式：virtual {method body}

纯虚函数的定义形式：virtual { } = 0;

在虚函数和纯虚函数的定义中不能有static标识符，原因很简单，被static修饰的函数在编译时候要求前期bind,然而虚函数却是动态绑定(run-time bind)，而且被两者修饰的函数生命周期(life recycle)也不一样。

6.虚函数必须实现，如果不实现，编译器将报错，错误提示为：

error LNK\*\*\*\*: unresolved external symbol "public: virtual void \_\_thiscall  
ClassName::virtualFunctionName(void)"

7.对于虚函数来说，父类和子类都有各自的版本。由多态方式调用的时候动态绑定。

8.实现了纯虚函数的子类，该纯虚函数在子类中就编程了虚函数，子类的子类即孙子类可以覆盖该虚函数，由多态方式调用的时候动态绑定。

9.虚函数是C++中用于实现多态(polymorphism)的机制。核心理念就是通过基类访问派生类定义的函数

10.多态性指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。  
a.编译时多态性：通过重载函数实现  
b 运行时多态性：通过虚函数实现。

11.如果一个类中含有纯虚函数，那么任何试图对该类进行实例化的语句都将导致错误的产生，因为抽象基类(ABC)是不能被直接调用的。必须被子类继承重载以后，根据要求调用其子类的方法。

总结：

1.虚函数必须实现，不实现编译器会报错。  
2.父类和子类都有各自的虚函数版本。由多态方式在运行时动态绑定。  
3.通过作用域运算符可以强行调用指定的虚函数版本。  
4.纯虚函数声明如下：virtual void funtion()=0; 纯虚函数无需定义。包含纯虚函数的类是[抽象基类](#_抽象类和接口的区别)，抽象基类不能创建对象，但可以声明指向抽象基类的指针或引用。  
5.派生类实现了纯虚函数以后，该纯虚函数在派生类中就变成了虚函数，其子类可以再对该函数进行覆盖。  
6.析构函数通常应该是虚函数，这样就能确保在析构时调用正确的析构函数版本。

**虚函数：**

在C++中，基类必须将它的两种成员函数区分开来：一种是基类希望其派生类进行覆盖的函数；另一种是基类希望派生类直接继承而不要改变的函数。对于前者，基类通过在函数之前加上virtual关键字将其定义为虚函数（virtual）。

class Base{ // 基类

public:

  virtual int func(int n) const;

};

class Derive\_Class : public Base{

public:

  int func(int n) const; // 默认也为虚函数

};

  虚函数为了重载和多态的需要，在基类中是有定义的，即便定义是空，所以子类中可以重写也可以不写基类中的此函数！

**纯虚函数：**

class test{

public:

virtual void print();

virtual void order()=0;

int array[20];};

上面声明了一个非常简单的类，它只有两个函数，其中一个是虚函数：打印，另外一个是纯虚函数：排序。其中打印函数的定义如下

void test::print(){

order();

printf("打印结果： ");

for(int i=0; i<20; i++) printf("%d ", array[i]);}

在这个打印函数中，调用了order函数对array进行了排序，然后输出结果。

**当函数没有实现方法或者需要子类来定义实现方法的时候，可以在父类中定义纯虚函数。**当不同的子类继承这个父类的时候，定义不同的实现方法，那么实例化这个子类的时候，这个纯虚函数就有了不同的方法。这也解释了为什么包含纯虚函数的抽象类为什么不能实例化，因为它中间有函数根本不知道是怎么个实现！**纯虚函数在基类中是没有定义的，必须在子类中加以实现。**

当然我们可以用其他方法避免使用纯虚函数，比方说在子类中重写print方法，但是这样一来等于除了order函数代码以外所有的代码都要重新复制一遍，当继承类越来越多的时候，要修改print等于这一堆继承类都要修改，会疯的！所以说纯虚函数是一个很神奇的用法，也是简化了编程使得面向对象的方法更加灵活。

## 静态多态

在模板编程及泛型编程中，是以隐式接口和编译器多态来实现静态多态。

静态多态：对于相关的对象类型，直接实现它们各自的定义，不需要共有基类，甚至可以没有任何关系。只需要各个具体类的实现中要求相同的接口声明，这里的接口称之为隐式接口。客户端把操作这些对象的函数定义为模板，当需要操作什么类型的对象时，直接对模板指定该类型实参即可（或通过实参演绎获得）。

对模板参数而言，多态是通过模板具现化和**函数重载**解析实现的。**以不同的模板参数具现化导致调用不同的函数**，这就是所谓的编译期多态。相比较于运行期多态，实现编译期多态的类之间并不需要成为一个继承体系，它们之间可以没有什么关系，但约束是它们都有相同的隐式接口。

优点：

1、它带来了泛型编程的概念，使得C++拥有泛型编程与STL这样的强大武器。

2、在编译器完成多态，提高运行期效率。

3、具有很强的适配性与松耦合性，对于特殊类型可由模板偏特化、全特化来处理。

缺点：

1、程序可读性降低，代码调试带来困难。

2、无法实现模板的分离编译，当工程很大时，编译时间不可小觑。

3、无法处理异质对象集合。

## 重写/重载/重定义

**重载overload：**

重载一般是用于在一个类内实现若干重载的方法，函数名相同，参数列表不同 重载只是在类的内部存在。但是不能靠返回类型来判断。

成员函数重载特征：  
   a 相同的范围（在同一个类中）

 b 函数名字相同

 c 参数不同

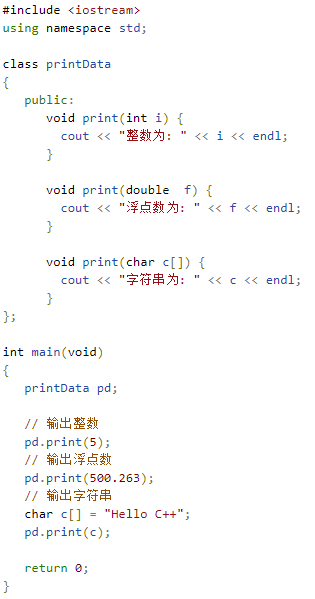
 d virtual关键字可有可无

e 返回值可以不同；

重载的规则：  
1、在使用重载时只能通过相同的方法名、不同的参数形式实现。不同的参数类型可以是不同的参数类型，不同的参数个数，不同的参数顺序（参数类型必须不一样）；  
2、不能通过访问权限、返回类型、抛出的异常进行重载；  
3、方法的异常类型和数目不会对重载造成影响；

**C++ 中的函数重载：**

在同一个作用域内，可以声明几个功能类似的同名函数，但是这些同名函数的形式参数（指参数的个数、类型或者顺序）必须不同。不能仅通过返回类型的不同来重载函数。



**C++ 中的运算符重载：**

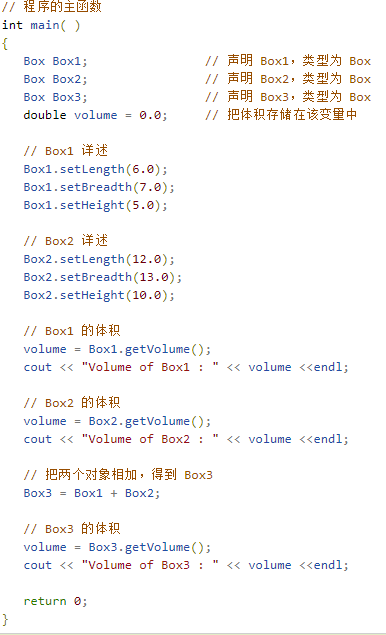
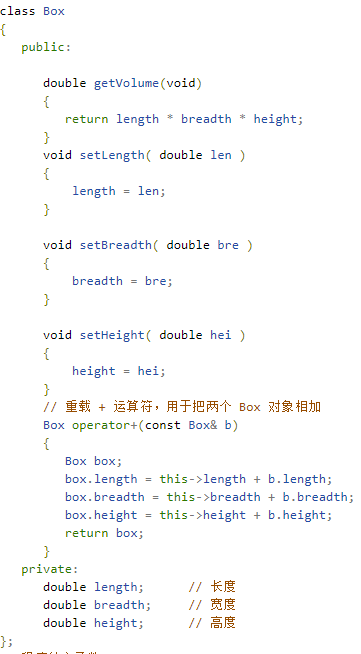
您可以重定义或重载大部分 C++ 内置的运算符。这样，您就能使用自定义类型的运算符。重载的运算符是带有特殊名称的函数，函数名是由关键字 operator 和其后要重载的运算符符号构成的。与其他函数一样，重载运算符有一个返回类型和一个参数列表。

Box operator+(const Box&);

声明加法运算符用于把两个 Box 对象相加，返回最终的 Box 对象。大多数的重载运算符可被定义为普通的非成员函数或者被定义为类成员函数。如果我们定义上面的函数为类的非成员函数，那么我们需要为每次操作传递两个参数，如下所示：

Box operator+(const Box&, const Box&);

下面的实例使用成员函数演示了运算符重载的概念。在这里，对象作为参数进行传递，对象的属性使用 this 运算符进行访问



**重写override：**

重写（覆盖)是指派生类函数覆盖基类函数，子类重新定义父类中有相同名称和参数的虚函数。函数特征相同。但是具体实现不同，主要是在继承关系中出现的。

特征是：

a 不同的范围，分别位于基类和派生类中

   b 函数的名字相同

   c 参数相同

   d 基类函数必须有virtual关键字

e 返回值相同（或是协变），否则报错；<—-协变这个概念我也是第一次才知道…

f 重写函数的访问修饰符可以不同。尽管 virtual 是 private 的，派生类中重写改写为 public,protected 也是可以的

重写需要注意：

1、被重写的函数不能是static的。必须是virtual的

2、重写函数必须有相同的类型，名称和参数列表,否则不能称其为重写而是重载.

3、重写函数的访问修饰符可以不同。尽管virtual是private的，派生类中重写改写为public,protected也是可以的。重写方法的访问修饰符一定要大于被重写方法的访问修饰符（public>protected>default>private）。被重写的方法不能为private，否则在其子类中只是新定义了一个方法，并没s有对其进行重写。  
4、重写的方法所抛出的异常必须和被重写方法的所抛出的异常一致，或者是其子类；  
5、重写的方法的返回值必须和被重写的方法的返回一致；  
6、静态方法不能被重写为非静态的方法（会编译出错）。

**注意区分虚函数中的重载和重写：**

class A{  
public:  
 virtual int fun(){}  
};  
  
class B:public A{  
 int fun(int a){} //这是重载而不是重写：

}  
  
int mian()  
{  
  
}

class B:public A{  
 int fun() // 从A继承来的 fun, 编译器会自己偷偷帮你加上  
 int fun(int a){} // 新的fun, 和前面的只是名字一样的[重载函数](https://www.baidu.com/s?wd=%E9%87%8D%E8%BD%BD%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3uHIWnj0zPjmLmhmvnjc40ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjcznHT3rHnz), 不是[虚函数](https://www.baidu.com/s?wd=%E8%99%9A%E5%87%BD%E6%95%B0&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3uHIWnj0zPjmLmhmvnjc40ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjcznHT3rHnz)  
}

**重定义 (redefining)也叫做隐藏:**

重定义(隐藏)是指派生类的函数屏蔽了与其同名的基类函数，子类重新定义父类中有相同名称的非虚函数 ( 参数列表可以不同 ) 。

如果一个类，存在和父类相同的函数，那么，这个类将会覆盖其父类的方法，除非你在调用的时候，强制转换为父类类型，否则试图对子类和父类做类似重载的调用是不能成功的。

（1）不在同一个作用域（分别位于派生类与基类） ；  
（2）函数名字相同；  
（3）返回值可以不同；  
（4）参数不同。此时，不论有无 virtual 关键字，基类的函数将被隐藏（注意别与重载以及覆盖混淆） 。  
（5）参数相同，但是基类函数没有 virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆） 。

规则如下：

   a 如果派生类的函数和基类的函数同名，但是参数不同，此时，不管有无virtual，基类的函数被隐藏。

   b 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vitual关键字，此时，基类的函数被隐藏。

**继承**

继承是子类使用父类的方法，而多态则是父类使用子类的方法。

继承就是新类从已有类那里得到已有的特性。类的派生指的是从已有类产生新类的过程。原有的类成为基类或父类，产生的新类称为派生类或子类，子类继承基类后，可以创建子类对象来调用基类函数，变量等。继承可以扩展已存在的代码，目的也是为了代码重用。

单一继承：继承一个父类，这种继承称为单一继承，一般情况尽量使用单一继承，使用多重继承容易造成混乱易出问题

多重继承：继承多个父类，类与类之间要用逗号隔开，类名之前要有继承权限，假使两个或两个基类都有某变量或函数，在子类中调用时需要加类名限定符如c.a::i = 1；

菱形继承：多重继承掺杂隔代继承1-n-1模式，此时需要用到虚继承，例如 B，C虚拟继承于A，D再多重继承B，C，否则会出错

**继承权限：**

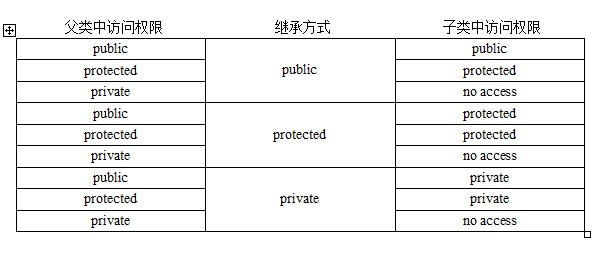
子类继承基类除构造和析构函数以外的所有成员。继承也分为接口继承和实现继承：

　　普通成员函数的接口总是会被继承：　　子类继承一份接口和一份强制实现

　　普通虚函数被子类重写　　　　　：　　子类继承一份接口和一份缺省实现

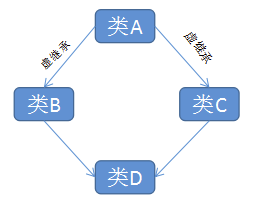
　　纯虚函数只能被子类继承接口　　：　　子类继承一份接口，没有继承实现

**访问权限：**



## 虚继承

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。为了解决从不同途径继承来的同名的数据成员在内存中有不同的拷贝造成数据不一致问题，将共同基类设置为虚基类。这时从不同的路径继承过来的同名数据成员在内存中就只有一个拷贝，同一个函数名也只有一个映射。这样不仅就解决了二义性问题，也节省了内存，避免了数据不一致的问题。



1、D继承了B,C也就继承了两个虚基类指针

2、虚基类表存储的是，虚基类相对直接继承类的偏移（D并非是虚基类的直接继承类，B,C才是）

**语法：**

virtual是关键字，声明该基类为派生类的虚基类。

class 派生类: virtual 基类1，virtual 基类2，...，virtual 基类n

{

...//派生类成员声明

};

1.在多继承情况下，虚基类关键字的作用范围和继承方式关键字相同，只对紧跟其后的基类起作用。

2.声明了虚基类之后，虚基类在进一步派生过程中始终和派生类一起，维护同一个基类子对象的拷贝。

3.观察类构造函数的构造顺序，拷贝也只有一份。

**执行顺序：**

首先执行虚基类的构造函数，多个虚基类的构造函数按照被继承的顺序构造；

执行基类的构造函数，多个基类的构造函数按照被继承的顺序构造；

执行成员对象的构造函数，多个成员对象的构造函数按照申明的顺序构造；

执行派生类自己的构造函数；

析构以与构造相反的顺序执行；

从虚基类直接或间接派生的派生类中的构造函数的成员初始化列表中都要列出对虚基类构造函数的调用。但只有用于建立对象的最派生类的构造函数调用虚基类的构造函数，而该派生类的所有基类中列出的对虚基类的构造函数的调用在执行中被忽略，从而保证对虚基类子对象只初始化一次。

在一个成员初始化列表中同时出现对虚基类和非虚基类构造函数的调用时，虚基类的构造函数先于非虚基类的构造函数执行。

不同编译器对间接存取的方法不同，以下以GCC和VC为例，均采用以下代码进行实验：

1. **class** SuperBase
2. {
3. **public**:
4. **int** m\_nValue;
5. **void** Fun(){}
6. **virtual** ~SuperBase(){}
7. };
8. **class** Base1:  **virtual** **public** SuperBase
9. {
10. **public**:
11. **virtual** ~ Base1(){}
12. };
13. **class** Base2:  **virtual** **public** SuperBase
14. {
15. **public**:
16. **virtual** ~ Base2(){}
17. };
18. **class** Der:**public** Base1, **public** Base2
19. {
20. **public**:
21. **virtual** ~ Der(){}
22. };

1) GCC中结果为8, 12, 12, 16

解析：sizeof(SuperBase) = sizeof(int) + 虚函数表指针

sizeof(Base1) = sizeof(Base2) = sizeof(int) + 虚函数指针 + 虚基类指针

sizeof(Der) = sizeof(int) + Base1中虚基类指针 + Base2虚基类指针 + 虚函数指针

**GCC共享虚函数表指针，也就是说父类如果已经有虚函数表指针，那么子类中共享父类的虚函数表指针空间，不在占用额外的空间，这一点与VC不同，VC在虚继承情况下，不共享父类虚函数表指针**，详见如下。

2）VC中结果为：8, 16, 16, 24

解析：sizeof(SuperBase) = sizeof(int) + 虚函数表指针

sizeof(Base1) = sizeof(Base2) = sizeof(int) + SuperBase虚函数指针 + 虚基类指针 + 自身虚函数指针

sizeof(Der) = sizeof(int) + Base1中虚基类指针 + Base2中虚基类指针 + Base1虚函数指针 + Base2虚函数指针 + 自身虚函数指针

如果去掉虚继承，结果将和GCC结果一样，A,B,C都是8，D为16，**原因就是VC的编译器对于非虚继承，父类和子类是共享虚函数表指针的。**

**继承中的内存布局**

1. 虚函数指针(vptr)放最前，之后放变量。  
2. 多个父类排着放，再放子类  
3. 子类的覆盖的虚函数将所有祖先的同名虚函数都覆盖。  
4. 子类其它的虚函数指针放在第一个父类的虚函数表里。  
5. 虚拟继承的情况只需要在钻石继承中有必要使用（避免二义性），子类中最先的祖先放最后。

**实现：**

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）。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。

## 虚继承和虚函数对比

虚继承和虚函数是完全无相关的两个概念。

**相似点：**

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）。

**不同点：**

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。

**二义性：**

8: #include <iostream>

9: using namespace std;

12: class Base{

14: public:

15: Base(){cout << "Base called..."<< endl;}

16: void print(){cout << "Base print..." <<endl;}

17: private:

18: };

21: class Sub{//定义一个类 Sub

23: public:

24: Sub(){cout << "Sub called..." << endl;}

25: void print(){cout << "Sub print..." << endl;}

26: private:

27: };

28:

29: //Child

30: class Child : public Base , public Sub{ //定义一个类Child 分别继承自 Base ，Sub

32: public:

33: Child(){cout << "Child called..." << endl;}

34: private:

35: };

36:

37: int main(int argc, char\* argv[]){

39: Child c;

41: //不能这样使用，会产生二意性，VC下error C2385

42: //c.print();

44: //只能这样使用

45: c.Base::print();

46: c.Sub::print();

48: system("pause");

49: return 0;

50: }

**多重继承**

#include <iostream>

9: using namespace std;

10:

11: int gFlag = 0;

13: class Base

14: {

15: public:

16: Base(){cout << "Base called : " << gFlag++ << endl;}

17: void print(){cout << "Base print" <<endl;}

18: };

20: class Mid1 : public Base

21: {

22: public:

23: Mid1(){cout << "Mid1 called" << endl;}

24: private:

25: };

27: class Mid2 : public Base

28: {

29: public:

30: Mid2(){cout << "Mid2 called" << endl;}

31: };

33: class Child:public Mid1, public Mid2

34: {

35: public:

36: Child(){cout << "Child called" << endl;}

37: };

38:

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

40: {

41: Child d;

42:

43: //不能这样使用，会产生二意性

//d.print();

45:

46: //只能这样使用

47: d.Mid1::print();

48: d.Mid2::print();

49:

50: system("pause");

51: return 0;

52: }

**虚拟继承**

#include "stdafx.h"

8: #include <iostream>

9: using namespace std;

10:

11: int gFlag = 0;

13: class Base

14: {

15: public:

16: Base(){cout << "Base called : " << gFlag++ << endl;}

17: void print(){cout << "Base print" <<endl;}

18: };

19:

20: class Mid1 : **virtual** public Base {

22: public:

23: Mid1(){cout << "Mid1 called" << endl;}

24: private:

25: };

26:

27: class Mid2 : **virtual** public Base {

29: public:

30: Mid2(){cout << "Mid2 called" << endl;}

31: };

32:

33: class Child:public Mid1, public Mid2 {

35: public:

36: Child(){cout << "Child called" << endl;}

37: };

38:

39: int main(int argc, char\* argv[]) {

41: Child d;

42:

43: //这里可以这样使用

44: d.print();

45:

46: //也可以这样使用

47: d.Mid1::print();

48: d.Mid2::print();

49:

50: system("pause");

51: return 0;

52: }

**C++动态性(虚函数/抽象类/动态联编/多态)**

动态性就是指在程序在运行的时候才能确定具体的功能；

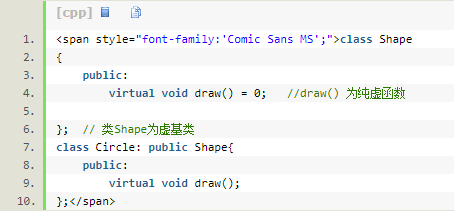
在c++中的动态性具体体现为类中的虚函数；虚基类；动态联编和多态性。

1.[虚函数](#_虚函数)

虚函数就是在类中函数的前面加上关键字virtual，一旦一个类中的函数被申明为虚函数，那么它的子类中的对应函数也是虚函数，为了提高可读性，可以显示加上virtual关键字。

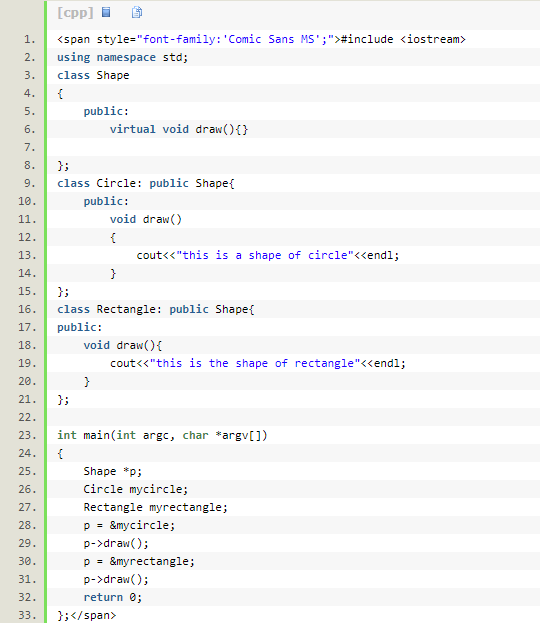
2.[抽象基类](#_抽象类和接口的区别)

抽象基类就是将类中的虚函数申明为纯虚函数，那么该类就是抽象类，抽象类不能实例化，其作用就是为了让其派生类继承它的接口。



**3.动态联编**

动态联编就是用基类的指针去调用子类中的对象，程序会在动态运行的时候决定调用哪个一个子类的函数。



**4.多态性**

多个派生类一起继承了一个基类，每个派生类的对象都可以当成基类对象来使用，比如形参中当成基类对象。将虚函数和动态联编和在一起就成了多态。

**---------\*关键字\*----------**

**[volatile关键字](http://www.cppblog.com/mzty/archive/2006/08/08/10959.html)**

　　volatile关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。  
  
使用该关键字的例子如下：

int volatile nVint;

　　当要求使用volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。  
  
例如：

volatile int i=10;

int a = i;

...

//其他代码，并未明确告诉编译器，对i进行过操作

int b = i;

　　volatile 指出 i是随时可能发生变化的，每次使用它的时候必须从i的地址中读取，因而编译器生成的汇编代码会重新从i的地址读取数据放在b中。而优化做法是，由于编译器发现两次从i读数据的代码之间的代码没有对i进行过操作，它会自动把上次读的数据放在b中。而不是重新从i里面读。这样以来，如果i是一个寄存器变量或者表示一个端口数据就容易出错，所以说volatile可以保证对特殊地址的稳定访问。

**Static**

static的作用主要有两种:第一个作用是限定作用域；第二个作用是保持变量内容持久化；

**c语言中static的用法：**

1、全局静态变量：

　　静态存储区，在整个程序运行期间一直存在。未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2、局部静态变量：

　　静态存储区；未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；

　　作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；

3、静态函数：

　　在函数返回类型前加关键字static，函数就定义成静态函数。函数的定义和生命在默认情况下都是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用；

**c++中static的用法：**

**1、类的静态成员**：在程序的全局数据区分配，定义时要分配空间，所以不能在类声明中定义

　　class A{

　　private:

　　　　static int val;

　　};

在cpp中必须对他进行初始化，初始化时使用作用域运算符来标明他所属类，其属于该类的所有成员共有，只有一个拷贝；

**2、类的静态成员函数**：

静态成员函数与类相联系，不与类的对象相联系。

静态成员函数不能访问非静态数据成员。原因很简单，非静态数据成员属于特定的类实例，静态成员函数主要用于对静态数据成员的操作。静态成员函数和静态数据成员都没有this指针。

　　class A{

　　private:

　　　　static int func(int x);

　　};

　　 实现的时候也不需要static的修饰，因为static是声明性关键字；类的静态函数是该类的范畴内的全局函数，不能访问类的私有成员，只能访问类的静态成员，不需要类的实例即可调用；实际上，他就是增加了类的访问权限的全局函数；

　　void  A::func(int);

　　静态成员函数可以继承和覆盖，但无法是虚函数；

**3、只在cpp内有效的全局变量**：

　　在cpp文件的全局范围内声明：

　　static int val = 0；

　　这个变量的含义是该cpp内有效，但是其他的cpp文件不能访问这个变量；如果有两个cpp文件声明了同名的全局静态变量，那么他们实际上是独立的两个变量；

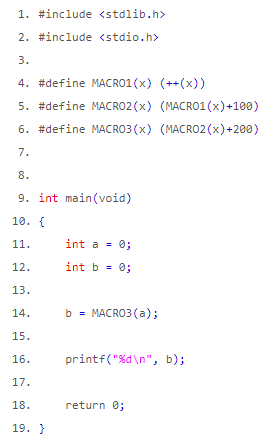
**4、只在cpp内有效的全局函数：**

　　函数的实现使用static修饰，那么这个函数只可在本cpp内使用，不会同其他cpp中的同名函数引起冲突；

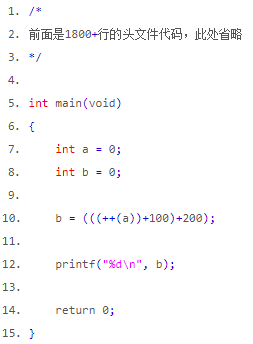
　　warning：不要再头文件中声明static的全局函数，不要在cpp内声明非static的全局函数，如果你要在多个cpp中复用该函数，就把它的声明提到头文件里去，否则cpp内部声明需加上static修饰；

**gdb调试宏**

第一个方法是通过gcc -E 产生预编译后的源代码，即源代码经过预编译后的结果，所有的预编译动作都已完成。如头文件的插入，宏定义的展开。



使用gcc -E test.c > test.e，得到预编译后的代码：



但是从这个例子也可以看到这个方法的局限性。

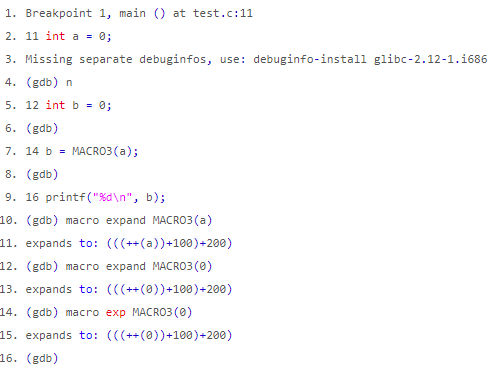
1. 由于预编译处理会执行所有的预处理代码，包括头文件的插入，这导致最后的代码行数太多。

2. 得到的了一个新的代码文件。这样的话，在大型工程中，如果需要调试多个文件中的宏定义，需要我们一个一个的预编译，太麻烦了。

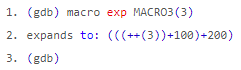
下面看看第二个方法，这个方法要比第一种方法方便得多。

我们都知道为了调试程序，需要使用-g选项，它的作用就是将调试信息加入到最后的二进制可执行文件中。但是你可知道-g 也通-o一样，是分级别的。当不指定级别的时候，其level为2。为了调试宏定义，我们可以使用更高的级别-g3。

下面为我使用-g3编译上面的代码，然后进行调试：



在调试的过程中，可以使用macro expand/exp 来展开宏定义。从上面的调试过程中，可以直接看到宏定义展开后的结果。并且我们还可以给宏传入任何的一个值，如：



第二个方法无疑比第一个方法要方便简单得多。我们只需要在全局的Makefile中添加新的编译参数-g3，就可以支持整个工程代码中所有的宏的调试。当然这个方法也有一个缺点，就是g3的调试信息会比默认的g2的调试信息要大——自然嘛，不然gdb如何知道怎样展开宏定义呢。

**#define和const**

C++ 语言可以用const 来定义常量，也可以用#define 来定义常量。但是前者比后者有更多的优点：  
（1） const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应）。  
（2） 有些集成化的调试工具可以对const 常量进行调试，但是不能对宏常量进行调试。规则5-2-1：在C++ 程序中只使用const 常量而不使用宏常量，即const 常量完全取代宏常量。

**2.实现机制**

宏是预处理命令，即在预编译阶段进行字节替换，它不做语法检查。const常量是变量，在执行时const定义的只读变量在程序运行过程中只有一份拷贝（因为它是全局的只读变量，存放在静态存储区的只读数据区。根据c/c++语法，当你声明该量为常量，即告诉程序和编译器，你不希望此量被修改。 程序的实现，为了保护常量，特将常量都放在受保护的静态存储区内。凡是试图修改这个区域内的值，都将被视为非法，并报错。这个跟数据类型没有关系，而是这个量是变量还是常量的问题。

例如，一个字符串变量就是可以被修改的。 这种静态存储区域的保护机制是由编译器实现的，而非存储该值的内存的电器属性。换言之，实质上内存永远都可以被用户随意修改，只是编译器给用户的代码注入了一些自己的保护代码，通过软件手段将这段内存软保护起来。这种保护在汇编级别可以轻松突破，其保护也就无效了。）。

**3.用法区别**

define宏定义和const常变量区别：

1.define是宏定义，程序在预处理阶段将用define定义的内容进行了替换。因此程序运行时，常量表中并没有用define定义的常量，系统不为它分配内存。const定义的常量，在程序运行时在常量表中，系统为它分配内存。

2.define定义的常量，预处理时只是直接进行了替换。所以编译时不能进行数据类型检验。const定义的常量，在编译时进行严格的类型检验，可以避免出错。

3.define定义表达式时要注意“边缘效应”，例如如下定义：  
#define N 2+3 //我们预想的N值是5，我们这样使用N，int a = N/2; //我们预想的a的值是2，可实际上a的值是3。原因在于在预处理阶段，编译器将 a = N/2处理成了 a = 2+3/2；这就是宏定义的字符串替换的“**边缘效应**”因此要如下定义：#define N (2+3)。

const定义的表达式则没有上述问题。const定义的常量叫做常变量原因有二：const定义常量像变量一样检查类型；const可以在任何地方定义常量，编译器对它的处理过程与变量相似，只是分配内存的地方不同。

**方式 1：**

**宏定义（替换）方式定义常量（只是实现类似效果）**：（严格来讲并不是常量，没有类型，也不会分配内存空间，所以不能称为量，但可以实现类似常量的效果）

  (1) 为一些反复使用常数定义标识

       eg. #define PI 3.14

   a.  程序怎么处理宏定义：程序开始编译之前，就是**预编译阶段**，会使用3.14**替换**程序中使用的PI。

   b. PI作为常数的宏替换，是**不能作为左值**使用，所以常把这种宏定义称为定义常量。

   c. 宏定义一定要记住的一个核心要点： 它只是  替换！！！

      eg.

        常见错误：由于宏是预编译阶段的文本替换，很容易在忽视运算符优先级的情况下发生错误。

            #define  M  10

            #define  N   M+5

            int x= M\*X;

            在以上的代码中，编写者很可能希望表达式M\*N的值等10\*15，即150.但实际上，表达式M\*N将被替换为10\*10+5，即105.由于宏定义的缺陷，不论C还是C++，都推荐使用const，即下面这种方式。

**方式 2 修饰符 const：**

   使用：const限定符修饰变量时，被修饰的变量成为常量。常量不能做左值，**声明时必须定义**。例如：

        const double PI=3.14;

#define 与 const 的比较

(1) const 常量有数据类型,而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换,没有类型安全检查,并且在字符替换可能会产生意料不到的错误(边际效应) 。        
(2) 有些集成化的调试工具可以对 const 常量进行调试,  但是不能对宏常量进行调试。

**const**

|  |  |  |  |
| --- | --- | --- | --- |
| No. | 作用 | 说明 | 参考代码 |
| 1 | 可以定义const常量 |  | const int Max = 100; |
| 2 | 便于进行类型检查 | const常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误 | void f(const int i) { .........} //对传入的参数进行类型检查，不匹配进行提示 |
| 3 | 可以保护被修饰的东西 | 防止意外的修改，增强程序的健壮性。 | void f(const int i) { i=10;//error! } //如果在函数体内修改了i，编译器就会报错 |
| 4 | 可以很方便地进行参数的调整和修改 | 同宏定义一样，可以做到不变则已，一变都变 |  |
| 5 | 为函数重载提供了一个参考 |  | class A {            ......   void f(int i){......} //一个函数   void f(int i) const {......} //上一个函数的重载            ...... }; |
| 6 | 可以节省空间，避免不必要的内存分配 | const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序运行过程中只有一份拷贝，而#define定义的常量在内存中有若干个拷贝 | #define PI 3.14159 //常量宏 const doulbe  Pi=3.14159;  //此时并未将Pi放入ROM中              ...... double i=Pi;   //此时为Pi分配内存，以后不再分配！ double I=PI;  //编译期间进行宏替换，分配内存 double j=Pi;  //没有内存分配 double J=PI;  //再进行宏替换，又一次分配内存！ |
| 7 | 提高了效率 | 编译器通常不为普通const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高 |  |

**二、Const的使用**

**1、定义常量**  
(1)const修饰变量，以下两种定义形式在本质上是一样的。它的含义是：const修饰的类型为TYPE的变量value是不可变的。

 TYPE const ValueName = value;   
 const TYPE ValueName = value;  
(2)将const改为外部连接,作用于扩大至全局,编译时会分配内存,并且可以不进行初始化,仅仅作为声明,编译器认为在程序其他地方进行了定义.

 extend const int ValueName = value;

**2、指针使用CONST**  
(1)指针本身是常量不可变  
     char\* const pContent;   
(2)指针所指向的内容是常量不可变  
     const char \*pContent;   
(3)两者都不可变  
     const char\* const pContent;   
(4)还有其中区别方法，沿着\*号划一条线：  
如果const位于\*的左侧，则const就是用来修饰指针所指向的变量，即指针指向为常量；  
如果const位于\*的右侧，const就是修饰指针本身，即指针本身是常量。

**3、函数中使用CONST**

**(1)const修饰函数参数**  
a.传递过来的参数在函数内不可以改变(无意义，因为Var本身就是形参)

void function(const int Var);

b.参数指针所指内容为常量不可变

void function(const char\* Var);

c.参数指针本身为常量不可变(也无意义，因为char\* Var也是形参)

void function(char\* const Var);

d.参数为引用，为了增加效率同时防止修改。修饰引用参数时：

void function(const Class& Var); //引用参数在函数内不可以改变

void function(const TYPE& Var); //引用参数在函数内为常量不可变

这样的一个const引用传递和最普通的函数按值传递的效果是一模一样的,他禁止对引用的对象的一切修改,唯一不同的是按值传递会先建立一个类对象的副本, 然后传递过去,而它直接传递地址,所以这种传递比按值传递更有效.另外只有引用的const传递可以传递一个临时对象,因为临时对象都是const属性, 且是不可见的,他短时间存在一个局部域中,所以不能使用指针,只有引用的const传递能够捕捉到这个家伙

**(2)const修饰函数返回值**  
const修饰函数返回值其实用的并不是很多，它的含义和const修饰普通变量以及指针的含义基本相同。  
    a.const int fun1() //这个其实无意义，因为参数返回本身就是赋值。  
    b.const int \* fun2() //调用时 const int \*pValue = fun2();   
               //我们可以把fun2()看作成一个变量，即指针内容不可变。  
    c.int\* const fun3()   //调用时 int \* const pValue = fun2();   
                //我们可以把fun2()看作成一个变量，即指针本身不可变。

一般情况下，函数的返回值为某个对象时，如果将其声明为const时，多用于操作符的重载。通常，不建议用const修饰函数的返回值类型为某个对象或对某个对象引用的情况。原因如下：如果返回值为某个对象为const（const A test = A 实例）或某个对象的引用为const（const A& test = A实例） ，则返回值具有const属性，则返回实例只能访问类A中的公有（保护）数据成员和const成员函数，并且不允许对其进行赋值操作，这在一般情况下很少用到。  
**4、类相关CONST**

**(1)const修饰成员变量**  
const修饰类的成员函数，表示成员常量，不能被修改，同时它只能在初始化列表中赋值。  
    class A  
    {   
        …  
        const int nValue;         //成员常量不能被修改  
        …  
        A(int x): nValue(x) { } ; //只能在初始化列表中赋值  
     }   
**(2)const修饰成员函数**  
const修饰类的成员函数，则该成员函数不能修改类中任何非const成员函数。一般写在函数的最后来修饰。const成员函数可以被具有相同参数列表的非const成员函数重载。在这种情况下，类对象的常量性决定调用哪个函数。  
    class A  
    {   
        …  
       void function()const; //常成员函数,它不改变对象的成员变量.

//也不能调用类中任何非const成员函数。

}

对于const类对象/指针/引用，只能调用类的const成员函数，因此，const修饰成员函数的最重要作用就是限制对于const对象的使用。

a. const成员函数不被允许修改它所在对象的任何一个数据成员。 值得注意的是，把一个成员函数声明为const可以保证这个成员函数不修改数据成员，但是，如果数据成员是指针，则const成员函数并不能保证不修改指针指向的对象，编译器不会把这种修改检测为错误。

b. const成员函数能够访问对象的const成员，而其他成员函数不可以。

1）const成员函数可以访问非const对象的非const数据成员、const数据成员，也可以访问const对象内的所有数据成员；

2）非const成员函数可以访问非const对象的非const数据成员、const数据成员，但不可以访问const对象的任意数据成员；

3）作为一种良好的编程风格，在声明一个成员函数时，若该成员函数并不对数据成员进行修改操作，应尽可能将该成员函数声明为const 成员函数。

**(3)const修饰类对象/对象指针/对象引用**

const修饰类对象表示该对象为常量对象，其中的任何成员都不能被修改。对于对象指针和对象引用也是一样。

const修饰的对象，该对象的任何非const成员函数都不能被调用，因为任何非const成员函数会有修改成员变量的企图。  
例如：  
class AAA  
{   
    void func1();   
void func2() const;   
}   
const AAA aObj;   
aObj.func1(); ×  
aObj.func2(); 正确  
  
const AAA\* aObj = new AAA();   
aObj-> func1(); ×  
aObj-> func2(); 正确

**三、将Const类型转化为非Const类型的方法**

采用const\_cast 进行转换。    
用法：const\_cast <type\_id>  (expression)   
该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type\_id和expression的类型是一样的。

             常量指针被转化成非常量指针，并且仍然指向原来的对象；

             常量引用被转换成非常量引用，并且仍然指向原来的对象；

             常量对象被转换成非常量对象。

**四、使用const的一些建议**

1、要大胆的使用const，这将给你带来无尽的益处，但前提是你必须搞清楚原委；

2、要避免最一般的赋值操作错误，如将const变量赋值，具体可见思考题；

3、在参数中使用const应该使用引用或指针，而不是一般的对象实例，原因同上；

4、const在成员函数中的三种用法（参数、返回值、函数）要很好的使用；

5、不要轻易的将函数的返回值类型定为const;

6、除了重载操作符外一般不要将返回值类型定为对某个对象的const引用;

7、任何不会修改数据成员的函数都应该声明为const 类型。

**五、补充重要说明**

1、类内部的常量限制：使用这种类内部的初始化语法的时候，常量必须是被一个常量表达式

2、初始化的整型或枚举类型，而且必须是static和const形式。

3、如何初始化类内部的常量：

一种方法就是static 和 const 并用，在外部初始化，例如：

class A {

public: A() {}

private: static const int i; [//注意必须是静态的](file://注意必须是静态的)！

}；

const int A::i=3;

另一个很常见的方法就是初始化列表：

class A {

public: A(int i=0):test(i) {}

private: const int i;

}；

还有一种方式就是在外部初始化，如果在非const成员函数中，this指针只是一个类类型的；如果在const成员函数中，this指针是一个const类类型的；如果在volatile成员函数中,this指针就是一个volatile类类型的。new返回的指针必须是const类型的。

**const和static的区别**

1、const 常量的在超出其作用域的时候会被释放，但是 static 静态变量在其作用域之外并没有释放，只是不能访问。

2、static 修饰的是静态变量，静态函数。对于类来说，静态成员和静态函数是属于整个类的，而不是属于对象。可以通过类名来访问，但是其作用域限制于包含它的文件中。

3、static 变量在类内部声明，但是必须在类的外部进行定义和初始化。const 常量在类内部声明，但是定义只能在构造函数的初始化列表进行。

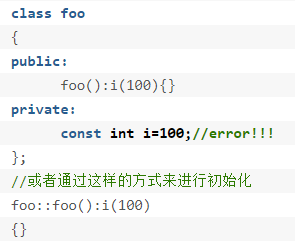
4、const数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的。因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。所以不能在类的声明中初始化const数据成员，因为类的对象没被创建时，编译器不知道const数据成员的值是什么。

5、const数据成员的初始化只能在类的构造函数的初始化列表中进行。要想建立在整个类中都恒定的常量，应该用类中的枚举常量来实现，或者static cosnt。

6、cosnt成员函数主要**目的**是防止成员函数修改对象的内容。即const成员函数不能修改成员变量的值，但可以访问成员变量。当方法成员函数时，该函数只能是const成员函数。static成员函数主要目的是作为类作用域的全局函数。不能访问类的非静态数据成员。类的静态成员函数没有this指针，这导致：1、不能直接存取类的非静态成员变量，调用非静态成员函数2、不能被声明为virtual

关于static、const、static cosnt、const static成员的初始化问题：

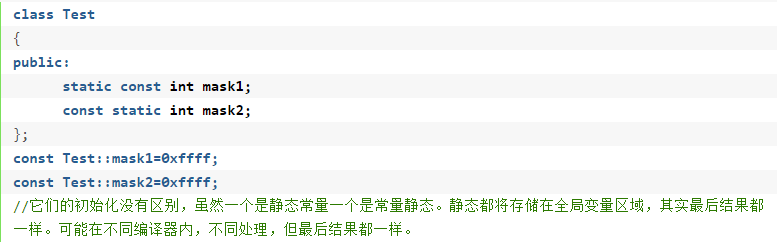
1、类里的const成员初始化：在一个类里建立一个const时，不能给他初值



2、类里的static成员初始化：类中的static变量是属于类的，不属于某个对象，它在整个程序的运行过程中只有一个副本，因此不能在定义对象时对变量进行初始化，就是不能用构造函数进行初始化，其正确的初始化方法是：**数据类型 类名::静态数据成员名=值**；



3、类里的static cosnt 和 const static成员初始化



**extern**

extern有两个作用，第一个,当它与"C"一起连用时，如: extern "C" void fun(int a, int b); 则告诉编译器在编译fun这个函数名时**按着C的规则去翻译**相应的函数名而不是C++的， C++的规则在翻译这个函数名时会把fun这个名字变得面目全非，可能是fun@aBc\_int\_int#%$也可能是别的，这要看编译器的"脾气"了 (不同的编译器采用的方法不一样)，为什么这么做呢，因为C++支持函数的重载啊，在这里不去过多的论述这个问题，如果你有兴趣可以去网上搜索，相信你可 以得到满意的解释!

当extern不与"C"在一起修饰变量或函数时，如在头文件中: extern int g\_Int;  它的作用就是**声明函数或全局变量的作用范围**的关键字，其声明的函数和变量可以在本模块活其他模块中使用，记住它是一个声明不是定义!也就是说B模块(编译 单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可, 在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

## 用extern修饰的全局变量

 以上已经说了extern的作用，下面我们来举个例子,如:  
    在test1.h中有下列声明:  
    #ifndef TEST1H  
    #define TEST1H  
    extern char g\_str[]; // 声明全局变量g\_str  
    void fun1();  
    #endif  
    在test1.cpp中  
    #include "test1.h"  
     
    char g\_str[] = "123456"; // 定义全局变量g\_str  
     
    void fun1()  
    {  
        cout << g\_str << endl;  
    }  
     
    以上是test1模块， 它的编译和连接都可以通过,如果我们还有test2模块也想使用g\_str,只需要在原文件中引用就可以了  
    #include "test1.h"  
  
    void fun2()  
    {  
        cout << g\_str << endl;  
    }  
    以上test1和test2可以同时编译连接通过，如果你感兴趣的话可以用ultraEdit打开test1.obj,你可以在里面着"123456"这 个字符串,但是你却不能在test2.obj里面找到，这是因为g\_str是整个工程的全局变量，在内存中只存在一份, test2.obj这个编译单元不需要再有一份了，不然会在连接时报告重复定义这个错误!  
    有些人喜欢把全局变量的声明和定义放在一起，这样可以防止忘记了定义，如把上面test1.h改为  
    extern char g\_str[] = "123456"; // 这个时候相当于没有extern  
    然后把test1.cpp中的g\_str的定义去掉,这个时候再编译连接test1和test2两个模块时，会报连接错误，这是因为你把全局变量 g\_str的定义放在了头文件之后，test1.cpp这个模块包含了test1.h所以定义了一次g\_str,而 test2.cpp也包含了test1.h所以再一次定义了g\_str, 这个时候连接器在连接test1和test2时发现两个g\_str。如果你非要把g\_str的定义放在test1.h中的话，那么就把test2的代码 中#include "test1.h"去掉 换成:  
    extern char g\_str[];  
    void fun2()  
    {  
        cout << g\_str << endl;  
    }  
    这个时候编译器就知道g\_str是引自于外部的一个编译模块了，不会在本模块中再重复定义一个出来，但是我想说这样做非常糟糕，因为你由于无法在 test2.cpp中使用#include "test1.h", 那么test1.h中声明的其他函数你也无法使用了，除非也用都用extern修饰，这样的话你光声明的函数就要一大串，而且头文件的作用就是要给外部提 供接口使用的，所以 请记住， 只在头文件中做声明，真理总是这么简单。

## 2. 用static修饰的全局变量

extern和static不能同时修饰一个变量；

static修饰的全局变量声明与定义同时进行，也就是说当你在头文件中使用static声明了全局变量后，它也同时被定义了；

static修饰全局变量的作用域 只能是本身的编译单元，也就是说它的“全局”只对本编译单元有效，其他编译单元则看不到它,如:  
    test1.h:  
    #ifndef TEST1H  
    #define TEST1H  
    static char g\_str[] = "123456";  
    void fun1();  
    #endif  
  
    test1.cpp:  
    #include "test1.h"  
     
    void fun1()  
    {  
        cout << g\_str << endl;  
    }  
     
    test2.cpp  
    #include "test1.h"  
     
    void fun2()  
    {  
        cout << g\_str << endl;  
    }  
     
    以上两个编译单元可以连接成功, 当你打开test1.obj时，你可以在它里面找到字符串"123456", 同时你也可以在test2.obj中找到它们，它们之所以可以连接成功而没有报重复定义的错误是因为虽然它们有相同的内容，但是存储的物理地址并不一样， 就像是两个不同变量赋了相同的值一样，而这两个变量分别作用于它们各自的编译单元。  
    也许你比较较真，自己偷偷的跟踪调试上面的代码,结果你发现两个编译单元（test1, test2）的g\_str的内存地址相同，于是你下结论static修饰的变量也可以作用于其他模块，但是我要告诉你，那是你的编译器在欺骗你，大多数编 译器都对代码都有优化功能，以达到生成的目标程序更节省内存，执行效率更高，当编译器在连接各个编译单元的时候，它会把相同内容的内存只拷贝一份，比如上 面的"123456", 位于两个编译单元中的变量都是同样的内容，那么在连接的时候它在内存中就只会存在一份了， 如果你把上面的代码改成下面的样子，你马上就可以拆穿编译器的谎言:  
    test1.cpp:  
    #include "test1.h"  
     
    void fun1()  
    {  
        g\_str[0] = 'a';  
        cout << g\_str << endl;  
    }  
  
    test2.cpp  
    #include "test1.h"  
     
    void fun2()  
    {  
        cout << g\_str << endl;  
    }  
     
    void main()  
    {  
        fun1(); // a23456  
        fun2(); // 123456  
    }  
     
    这个时候你在跟踪代码时，就会发现两个编译单元中的g\_str地址并不相同，因为你在一处修改了它，所以编译器被强行的恢复内存的原貌，在内存中存在了两份拷贝给两个模块中的变量使用。  
  
    正是因为static有以上的特性，所以一般定义static全局变量时，都把它放在原文件中而不是头文件，这样就不会给其他模块造成不必要的信息污染，同样记住这个原则吧！

## 3. const修饰的全局常量

const修饰的全局常量用途很广，比如软件中的错误信息字符串都是用全局常量来定义的。const修饰的全局常量据有跟static相同的特性，即它们只能作用于本编译模块中，但是const可以与extern连用来声明该常量可以作用于其他编译模块中, 如  
    extern const char g\_str[];  
    然后在原文件中别忘了定义:  
    const char g\_str[] = "123456";  
  
    所以当const单独使用时它就与static相同，而当与extern一起合作的时候，它的特性就跟extern的一样了！所以对const我没有什么 可以过多的描述，我只是想提醒你，const char\* g\_str = "123456" 与 const char g\_str[] = "123465"是不同的， 前面那个const 修饰的是char \* 而不是g\_str,它的g\_str并不是常量，它被看做是一个定义了的全局变量（可以被其他编译单元使用）， 所以如果你像让char \*g\_str遵守const的全局常量的规则，最好这么定义const char\* const g\_str="123456".

**explicit（防止隐性转换和拷贝初始化，然后举拷贝初始化失败的一个例子）**

通过使用explicit关键字，可以强制性的要求类的使用者按照规范的方式编写代码，提高代码的可读性，并降低发生错误的风险。

C++中的explicit关键字只能用于修饰只有一个参数的类构造函数, 它的作用是表明该构造函数是显示的。explicit关键字的作用就是防止类构造函数的隐式自动转换。

explicit关键字只对有一个参数的类构造函数有效, 如果类构造函数参数大于或等于两个时, 是不会产生隐式转换的, 所以explicit关键字也就无效了. 但是, 也有一个例外, 就是当除了第一个参数以外的其他参数都有默认值的时候, explicit关键字依然有效, 此时, 当调用构造函数时只传入一个参数, 等效于只有一个参数的类构造函数,

C++中的关键字explicit主要是用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以显示的方式进行类型转换。类构造函数默认情况下声明为隐式的即implicit。

explicit关键字的作用：**禁止隐式调用类内的单参数构造函数，**这主要包括如下三层意思：

（1）该关键字只能用来修饰类内部的构造函数

（2）禁止隐式调用拷贝构造函数

（3）禁止类对象之间的隐式转换

## 隐式转换和显示转换

**1）C++类型(char,int,float,long,double等)的隐式转换：**

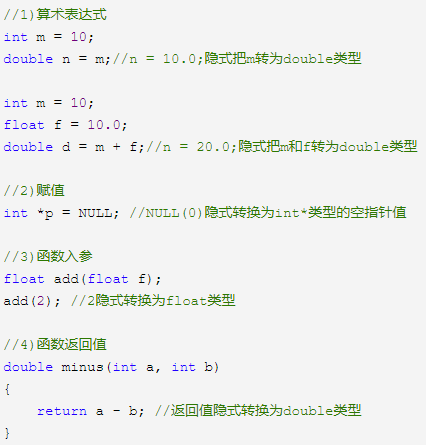
隐式转换即是可以由单个实参来调用的构造函数 定义了一个从形参类型到该类类型的隐式转换。编译器在试图编译某一条语句时，如果某一函数的参数类型不匹配，编译器就会尝试进行隐式转换，如果隐式转换后能正确编译，编译器就会继续执行编译过程，否则报错。

隐式转换：说白了就是在转换时不给系统提示具体的显示模型，让其自动转换，但是要记住一条编译器一般只支持**自下而上**的类型转换，例如int 转 float

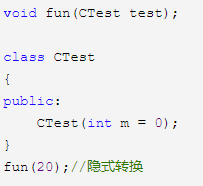
算术表达式隐式转换顺序为：

1、char - int - long - double

2、float - double

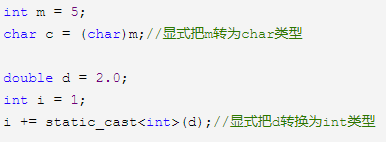


**2）C++类对象的隐式转换：**



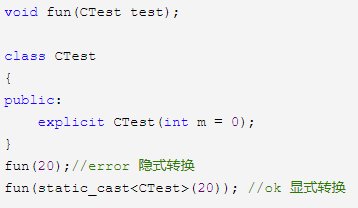
**1）C++类型(char,int,float,long,double等)的显式转换：**

显式转换也叫强制转换，是自己主动让这个类型转换成别的类型。



**2）C++类对象的显式转换：**

当类构造函数只有一个参数或除了第一个参数外其余参数都有默认值时，则此类有隐含的类型转换操作符(隐式转换)，但有时隐式转换并不是我们想要的，可在构造函数前加上关键字explicit，来指定显式调用。



**C++ 同时提供了四种新的强制转型形式**（通常称为新风格的或 C++ 风格的强制转型）：   
　　const\_cast(expression)   
　　dynamic\_cast(expression)   
　　reinterpret\_cast(expression)   
　　static\_cast(expression)   
  
　　每一种适用于特定的目的：

**static\_cast**

static\_cast < type-id > ( expression )，该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性，它主要有如下几种用法：

（1）用于基本数据类型之间的转换，如把int转换为char，把int转换成enum，但这种转换的安全性需要开发者自己保证（这可以理解为保证数据的精度，即程序员能不能保证自己想要的程序安全），如在把int转换为char时，如果char没有足够的比特位来存放int的值（int>127或int<-127时），那么static\_cast所做的只是简单的截断，及简单地把int的低8位复制到char的8位中，并直接抛弃高位。

（2）把空指针转换成目标类型的空指针

（3）把任何类型的表达式类型转换成void类型

（4）用于类层次结构中父类和子类之间指针和引用的转换。

对于以上第（4）点，存在两种形式的转换，即上行转换（子类到父类）和下行转换（父类到子类）。对于static\_cast，上行转换时安全的，而下行转换时不安全的，为什么呢？因为static\_cast的转换时粗暴的，它**仅根据类型转换语句中提供的信息（尖括号中的类型）来进行转换**，这种转换方式对于上行转换，由于子类总是包含父类的所有数据成员和函数成员，因此从子类转换到父类的指针对象可以没有任何顾虑的访问其（指父类）的成员。而对于下行转换为什么不安全，是因为**static\_cast只是在编译时进行类型检查，没有运行时的类型检查**，具体原理在dynamic\_cast中说明。

**dynamic\_cast**

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。

**作用：**

将一个基类对象指针（或引用）cast到继承类指针，dynamic\_cast会根据**基类指针是否真正指向继承类指针**来做相应处理，对指针进行dynamic\_cast，失败返回null，成功返回正常cast后的对象指针；

对引用进行dynamic\_cast，失败抛出一个异常，成功返回正常cast后的对象引用。

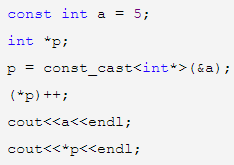
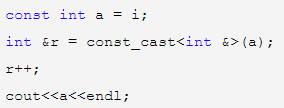
dynamic\_cast与一般强转区别：1、dynamic\_cast应用的范围要更小一些，他只能对指针或者引用来进行强转。2、dynamic\_cast多了一步安全的检查。

dynamic\_cast与static\_cast区别：对于上行转换，static\_cast和dynamic\_cast效果一样，都安全；对于下行转换：你必须确定要转换的数据确实是目标类型的数据，即需要注意要转换的父类类型指针是否真的指向子类对象，如果是，static\_cast和dynamic\_cast都能成功；如果不是static\_cast能返回，但是不安全，可能会出现访问越界错误，而dynamic\_cast在运行时类型检查过程中，判定该过程不能转换，返回NULL。

dynamic\_cast < type-id > ( expression)

该运算符把expression转换成type-id类型的对象。Type-id必须是类的指针、类的引用或者void\*；如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。dynamic\_cast运算符可以在执行期决定真正的类型。如果downcast是安全的（也就说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果downcast不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）。

**const\_cast** 一般用于强制消除对象的常量性。用于去除复合类型中const和volatile属性（没有真正去除）。变量本身的const属性是不能去除的，要想修改变量的值，一般是去除指针（或引用）的const属性，再进行间接修改。它是唯一能做到这一点的 C++ 风格的强制转型。

   
  
**reinterpret\_cast**

reinterpret\_cast用在任意指针（或引用）类型之间的转换；以及指针与足够大的整数类型之间的转换；从整数类型（包括枚举类型）到指针类型，无视大小。

CBase\* pBase = new CBase( ) ;

CDerived\* pDerived = reinterpret\_cast<CDerived\*>(pBase) ;

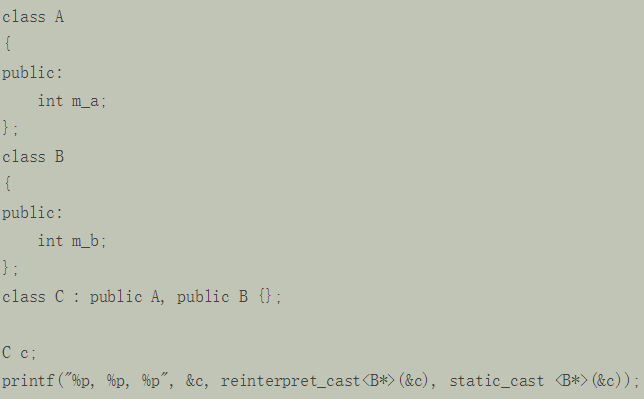
这种类型转换实际上是强制编译器接受static\_cast通常不允许的类型转换，它并没有改变指针值的二进制表示，只是改变了编译器对源对象的解释方式。

reinterpret\_cast不能像const\_cast那样去除const修饰符。reinterpret\_cast只能改变指针(或引用)的解释方式，不能把其上的const锁转换掉。

static\_cast和reinterpret\_cast的区别主要在于多重继承：

前两个的输出值是相同的，最后一个则会在原基础上偏移4个字节。

这是因为static\_cast计算了父子类指针转换的偏移量，并将之转换到正确的地址（c里面有m\_a,m\_b，转换为B\*指针后指到m\_b处），而reinterpret\_cast却不会做这一层转换。



explicit关键字只能用于类内部的构造函数声明上，而不能用在类外部的函数定义(函数实现)上，它的作用是不能进行隐式转换；

explicit关键字作用于单个参数的构造函数，如果构造函数有多个参数，但是从第二个参数开始，如果各参数均有默认赋值，也可以应用explicit关键字。

当构造函数只有一个参数时，会进行自动隐式转换，当构造函数参数个数超过或等于两个时自动取消隐式转换，当只有一个必须输入的参数，其余的为有默认值的参数时使用explicit也起作用。

一般只将有单个参数的构造函数声明为explicit，而拷贝构造函数不要声明为explicit。

explicit关键字只能对用户自己定义的对象起作用，不对默认构造函数起作用。此关键只能够修饰构造函数。无参数的构造函数和多参数的构造函数总是显示调用，这种情况在构造函数前加explicit无意义。

当不希望进行自动类型转换时用explicit，标准库的许多构造函数都是explicit的。

（1）explicit关键字只需用于类内的单参数构造函数前面。由于无参数的构造函数和多参数的构造函数总是显示调用，这种情况在构造函数前加explicit无意义。

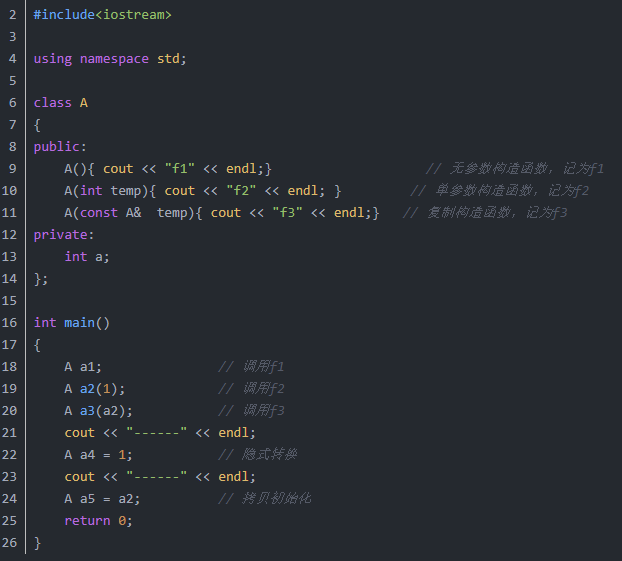
（2）如果想禁止类A对象被隐式转换为类B对象，可在类B中使用关键字explicit，即定义这样的转换构造函数

google的c++规范中提到explicit的优点是可以避免不合时宜的类型变换，缺点无。所以google约定所有单参数的构造函数都必须是显示的，只有极少数情况下拷贝构造函数可以不声明称explicit。例如作为其他类的透明包装器的类。  
　　effective c++中说：被声明为explicit的构造函数通常比其non-explicit兄弟更受欢迎。因为它们禁止编译器执行非预期（往往也不被期望）的类型转换。除非我有一个好理由允许构造函数被用于隐式类型转换，否则我会把它声明为explicit，鼓励大家遵循相同的政策。

**直接初始化与拷贝初始化**

在C++中，变量的初始化有两种方式：直接初始化（用（）运算符，如int a(1);）和复制初始化（用=运算符，如int a = 1;）。

直接初始化直接调用与实参匹配的构造函数，复制初始化总是调用复制构造函数。复制初始化首先使用指定构造函数创建一个临时对象，然后用复制构造函数将那个临时对象复制到正在创建的对象。所以当复制构造函数被声明为私有时，所有的复制初始化都不能使用。





 从运行结果可以看出，在直接初始化（即main函数的第二三句代码）过程中，会根据（）中的数据的类型去调用不同构造函数，其中就包括复制构造函数。A a4 = 1和A a5 = a2这两句代码是复制初始化。但是其中的A a4 = 1这句代码令人困惑，为什么一个数字可以初始化一个类对象呢？其实这个地方发生了隐式转换。 虽然上面的代码是可以正常运行的，但是像A a4 = 1这样的代码的可读性非常的差，因此，为了避免出现这种情况，我们可以使用explicit关键字来修饰构造函数，从而禁止隐式转换和复制初始化。



在单参数构造函数和拷贝构造函数之前加上explicit关键字，然后再编译代码。这时，编译不能通过。这就是因为explicit关键字禁止了隐式转换和复制初始化，因此编译不能通过。注释掉A a4 = 1和A a5 = a2这两句代码就可以编译通过了。 通过使用explicit关键字，可以强制性的要求类的使用者按照规范的方式编写代码，提高代码的可读性，并降低发生错误的风险。

**注意：**

1、拷贝构造函数和拷贝初始化不要混淆。拷贝初始化过程要调用拷贝构造函数，但拷贝构造函数也可以用在直接初始化过程。

2、explicit关键字只对单形参的构造函数起作用，对于多形参的构造函数，不会出现隐式转换的问题。 《C++ Primer》中有指出“可以用单个实参来调用的构造函数定义了从形参类型到该类型的一个隐式转换。”

3、隐式转换不仅会出现在初始化过程中，另一个很常见的情况是以类类型为形参的函数的参数传递过程，比如下面的代码，如果加上explicit关键字，是不能编译通过的。

**Default和delete**

有时候当我们仅创建了有参构造函数后，如果你想调用无参构造函数编译是会报错的。因为一旦你自己定义了构造函数，系统的默认构造函数是被屏蔽的，也就是说此时是没有无参构造函数的，所以我们需要自己定义一个无参构造函数。但是现在在C++11中，如果我们仅定义了有参构造函数，可以通过default关键字让默认构造函数恢复。

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

class Some

{

public:

    Some(Some&&);

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

    ...

};

相反地，如果要禁用编译器提供的默认函数，可以使用delete:

class Some

{

public:

    Some(Some&&);

    Some() = default; // use default constructor

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

    ...

};

当然要想禁用某个编译器提供的函数也可以显式声明为private，但是使用delete更方便且不易出错。default关键字只能用于6个特殊函数，而delete却能够用于任何成员函数。default在原有4个特殊成员函数（默认构造函数，复制构造函数，复制赋值运算符和析构函数）的基础上，C++11新增了两个：移动构造函数，移动赋值运算符。这些成员函数式编译器在各种情况下自动提供的。如果类定义了移动构造函数或移动赋值运算符，编译器将不会自动提供复制构造函数和复制赋值运算符。如果类定义了造函数，复制构造函数或析构函数，编译器将不会自动提供移动构造函数和移动赋值运算符

**auto**

auto和其他变量类型有明显的区别：

1.auto声明的变量必须要初始化，否则编译器不能判断变量的类型。

2.auto不能被声明为返回值，auto不能作为形参，auto不能被修饰为模板参数

用于代替冗长复杂、变量使用范围专一的变量声明。

decltype可以在编译的时候判断出一个变量或者表达式的类型

使用auto关键字做类型自动推导时，依次施加一下规则：

1、如果初始化表达式是引用，则去除引用语义。

2、如果初始化表达式为const或volatile（或者两者兼有），则除去const/volatile语义

3、如果auto关键字带上&号，则不去除const语意。

4、初始化表达式为数组时，auto关键字推导类型为指针。

auto的对应类型不是使用new出来的变量，而是static变量  
static变量是程序接收的时候才释放对象的，但它不需要手动释放。  
static如果在一个函数内申明，这每次进入这个函数时，还是使用第一次声明的变量，并且还保存的上次使用的值(auto变量这时函数结束是即释放了，再次调用这个函数是，有重新定义了一个新的变量）  
static变量如果在结构和类中使用，这结构或类定义的一切对象，都将共享唯一static变量。

**Override**

使用虚说明符override指出要覆盖的函数：将其放在参数类别后面。如果与基类方法不匹配，编译器将视为错误

* 1.确保在派生类中声明的重写函数跟基类的虚函数有相同的签名

virtual void func(double) override;

**final**

final:禁止派生类覆盖特定的虚方法

1.阻止类的进一步派生，禁用继承

2.阻止虚函数的进一步重写。禁用重写，C++中还允许将方法标记为fianal，这意味着无法在子类中重写该方法。这时final关键字至于方法参数列表后面。只能对虚函数使用

class TaskManager final {/\*..\*/} ;

class B: A

{

pulic:

void func() const override final; //OK

};

**-----------\*STL\*------------**

**STL中如何进行内存管理 内存池**

内存管理模块叫空间配置器，有两级，第一级是通过C语言的malloc()等分配内存的；第二级利用了内存池技术。如果申请的内存区域大于128B的时候，就会调用一级适配器，直接用malloc申请内存，用free释放内存。\_\_default\_alloc\_template 就是一个内存池了。

大多数情况下，STL默认的allocator就已经足够了。这个allocator是一个由两级分配器构成的内存管理器，当申请的内存大小大于128byte时，就启动第一级分配器通过malloc直接向系统的堆空间分配，如果申请的内存大小小于128byte时，就启动第二级分配器，从一个预先分配好的内存池中取一块内存交付给用户，这个内存池由16个不同大小（8的倍数，8~128byte）的空闲列表组成，allocator会根据申请内存的大小（将这个大小round up成8的倍数）从对应的空闲块列表取表头块给用户。

这种做法有两个优点：   
1）小对象的快速分配。小对象是从内存池分配的，这个内存池是系统调用一次malloc分配一块足够大的区域给程序备用，当内存池耗尽时再向系统申请一块新的区域，整个过程类似于批发和零售，显然是快捷了。当然，这里的一个问题时，内存池会带来一些内存的浪费，比如当只需分配一个小对象时，为了这个小对象可能要申请一大块的内存池，但这个浪费还是值得的，况且这种情况在实际应用中也并不多见。   
2）避免了内存碎片的生成。程序中的小对象的分配极易造成内存碎片，给操作系统的内存管理带来了很大压力，系统中碎片的增多不但会影响内存分配的速度，而且会极大地降低内存的利用率。以内存池组织小对象的内存，从系统的角度看，只是一大块内存池，看不到小对象内存的分配和释放。

STL中的内存分配器实际上是基于空闲列表(free list)的分配策略，最主要的特点是通过组织16个空闲列表，对小对象的分配做了优化。   
1）小对象的快速分配和释放。当一次性预先分配好一块固定大小的内存池后，对小于128字节的小块内存分配和释放的操作只是一些基本的指针操作，相比于直接调用malloc/free，开销小。   
2）避免内存碎片的产生。零乱的内存碎片不仅会浪费内存空间，而且会给OS的内存管理造成压力。   
3）尽可能最大化内存的利用率。当内存池尚有的空闲区域不足以分配所需的大小时，分配算法会将其链入到对应的空闲列表中，然后会尝试从空闲列表中寻找是否有合适大小的区域，   
  
但是，这种内存分配器局限于STL容器中使用，并不适合一个通用的内存分配。因为它要求在释放一个内存块时，必须提供这个内存块的大小，以便确定回收到哪个free list中，而STL容器是知道它所需分配的对象大小的，比如上述：   
stl::vector<int> array;  
array是知道它需要分配的对象大小为sizeof(int)。

一个通用的内存分配器是不需要知道待释放内存的大小的，类似于free(p)。

**Hash/map/hashmap/hashtable**

## Hash和红黑树的效率比较

权衡因素: 查找速度, 数据量, 内存使用，可扩展性，有序性。  
总体来说，hash查找速度会比RB树快，而且查找速度基本和数据量大小无关，属于常数级别;

而RB树的查找速度是log(n)级别。并不一定常数就比log(n) 小，因为hash还有hash函数的耗时。当元素达到一定数量级时，考虑hash。

但若你对内存使用特别严格，希望程序尽可能少消耗内存，那么hash可能会让你陷入尴尬，特别是当你的hash对象特别多时，你就更无法控制了，而且 hash的构造速度较慢。

红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。

如果只需要判断Map中某个值是否存在之类的操作，当然是Hash实现的要更加高效。

如果是需要将两个Map求并集交集差集等大量比较操作，就是红黑树实现的Map更加高效。

在实际的系统中，例如，需要使用动态规则的防火墙系统，使用红黑树而不是散列表被实践证明具有更好的伸缩性。Linux内核在管理vm\_area\_struct时就是采用了红黑树来维护内存块的。

**总结：**

红黑树是有序的，Hash是无序的，根据需求来选择。

红黑树占用的内存更小（仅需要为其存在的节点分配内存），而Hash事先就应该分配足够的内存存储散列表（即使有些槽可能遭弃用）。

红黑树查找和删除的时间复杂度都是O(logn)，Hash查找和删除的时间复杂度都是O(1)。

## Map和hashmap

1、STL的map底层是用红黑树实现的，查找时间复杂度是log(n)；  
2、STL的hash\_map底层是用hash表存储的，查询时间复杂度是O(1)；  
3、什么时候用map，什么时候用hash\_map？这个看具体的应用，不一定常数级别的hash\_map一定比log(n)级别的map要好，hash\_map的hash函数以及解决地址冲突等都要耗时间，而且众所周知hash表是以空间换时间的，因而hash\_map的内存消耗肯定要大，一般情况下，如果记录非常大，考虑hash\_map，查找效率会高很多，如果要考虑内存消耗，则要谨慎使用hash\_map。

## hashMap

https://blog.csdn.net/qq51931373/article/details/44196859

hash\_map基于hash table（哈希表）。哈希表最大的优点是：把数据存储和查询消耗的时间大大降低，几乎可以看成是常数时间；而代价仅仅是消耗比较多的内存。然后在当前可利用内存越来越多的情况下，用空间换时间的做法是值得的。另外，编码比较容易也是它的特点之一。  
  其基本原理是：使用一个下标范围比较大的数组来存储元素。可以设计一个函数（哈希函数，也叫散列函数），使得每个元素的关键字都与一个函数值（即数组下标，hash值）相对应，于是用这个数组单元来存储这个元素；也可以简单的理解为，按照关键字为每一个元素“分类”，然后将这个元素存储在相应“类”所对应的地方，称为桶。  
  但是，这不能够保证每个元素的关键字与函数值是一一对应的，因此极有可能出现对不不同的元素，却计算出相同的函数值，这样就产生了“冲突”。换句话说，就是把不同的元素分在了相同的“类”中。总的来说，“直接定址”和“解决冲突”是哈希表的两大特点。  
  hash\_map，首先分配一大片内存，形成许多桶。是利用hash函数，对key进行映射到不同区域进行保存。其插入过程：  
    1、得到key；  
    2、通过hash函数得到hash值；  
    3、得到桶号（一般都为hash值对桶数求模）；  
    4、存放key和value在桶内；  
    其取值过程是：  
    1、得到key；  
    2、通过hash函数得到hash值；      
    3、得到桶号；  
    4、比较桶的内部元素是否与key相等，若不相等，则没有找到；  
    5、取出相等的记录的value；  
    hash\_map中直接地址用hash函数生成，解决冲突，用比较函数解决。这里可以看出，如果每个桶内部只有一个元素，那么查找的时候只有一次比较。当许多桶没有值时，许多查询就会更快了（指查不到的时候）。

## hashTable

要实现哈希表，和用户相关的是：hash函数和比较函数。这两个参数刚好是我们在使用hash\_map时需要指定的参数。

可以提供任何有名项的存取操作和删除操作，由于操作对象是有名项，故可被视为一种字典结构。用意是在常数时间内提供基本操作。常见的，我们可以把int存到相应值的数组里面，就可以通过O(1)的时间存取和删除。

主要思想是通过hash函数，把对象映射到一个较小的容器里面，并且保证时间复杂度。映射到较小容器很可能出现碰撞问题，解决的方法常见的有：线性探测，二次探测，开链法。STL利用vector来当容器，采用开链法来解决冲突，从而实现hashtable。hashtable只能处理char,int,short等类型，不能处理string,double,float类型，想要处理的话必须自己加hash function.

## hashMap和HashTable的区别

HashMap和HashTable都实现了Map接口，主要区别有：线程安全性、同步、速度

1、线程安全：

HashMap是非synchronization的，HashTable是synchronization的，这意味着HashTable是线程安全的。HashMap是Hashtable的轻量级实现（非线程安全的实现），他们都完成了Map接口，主要区别在于HashMap允许空（null）键值（key）,由于非线程安全，效率上可能高于Hashtable。HashMap允许null key和null value，而hashtable不允许。HashMap允许将null作为一个entry的key或者value，而Hashtable不允许。 HashMap把Hashtable的contains方法去掉了，改成containsValue和containsKey。hashmap是一个接口，是map接口的子接口，是将键映射到值的对象，其中键和值都是对象，并且不能包含重复键，但可以包含重复值。

2、同步：

因为HashMap是非synchronization的，HashTable是synchronization的，所以需要同步的话用HashTable。

3、速度：

由于HashTable是线程安全的也是synchronized，所以在单线程环境下它比HashMap要慢

## Hashmap和hashset

**hashset**重点在于set几个字母上，主要实现集合功能，只不过底层函数由hashtable提供，而set是由红黑树作底层，两者的唯一区别就是set默认是排好序的，而hashset不是。

**hashmap**重点在于map几个字母上，主要提供对象间的一一映射功能，hashmap底层由hashtable

提供，而map底层由红黑树提供，区别同上。

**STL算法**

大家都能取得的一个共识是函数库对数据类型的选择对其可重用性起着至关重要的作用。举例来说，一个求方根的函数，在使用浮点数作为其参数类型的情况下的可重用性肯定比使用整型作为它的参数类性要高。而C++通过模板的机制允许推迟对某些类型的选择，直到真正想使用模板或者说对模板进行特化的时候，STL就利用了这一点提供了相当多的有用算法。它是在一个有效的框架中完成这些算法的——你可以将所有的类型划分为少数的几类，然后就可以在模版的参数中使用一种类型替换掉同一种类中的其他类型。

STL提供了大约100个实现算法的模版函数，比如算法for\_each将为指定序列中的每一个元素调用指定的函数，stable\_sort以你所指定的规则对序列进行稳定性排序等等。这样一来，只要我们熟悉了STL之后，许多代码可以被大大的化简，只需要通过调用一两个算法模板，就可以完成所需要的功能并大大地提升效率。

算法部分主要由头文件<algorithm>，<numeric>和<functional>组成。

<algorithm>是所有STL头文件中最大的一个（尽管它很好理解），它是由一大堆模版函数组成的，可以认为每个函数在很大程度上都是独立的，其中常用到的功能范围涉及到比较、交换、查找、遍历操作、复制、修改、移除、反转、排序、合并等等。

<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数，包括加法和乘法在序列上的一些操作。

<functional>中则定义了一些模板类，用以声明函数对象。

**STL容器**

所谓STL容器，就是将最常用的一些数据结构实现出来。常用的数据结构不外乎数组array、链表list、树tree、堆栈stack、队列queue、散列表hash table、集合set、映射表map等，根据“数据在容器中的排列”特性，这些数据结构可分为序列式和关联式两种。因此，容器大概也分为序列式容器和关联式容器两种。



**向量**(vector)连续存储的元素<vector>

**列表**(list)由节点组成的双向链表，每个结点包含着一个元素<list>

**双队列**(deque)连续存储的指向不同元素的指针所组成的数组<deque>

**集合**(set)由节点组成的红黑树，每个节点都包含着一个元素，节点之间以某种作用于元素对的谓词排列，没有两个不同的元素能够拥有相同的次序 <set>

**多重集合**(multiset)允许存在两个次序相等的元素的集合 <set>

**栈**(stack)后进先出的值的排列 <stack>

**队列**(queue)先进先出的执的排列 <queue>

**优先队列**(priority\_queue)元素的次序是由作用于所存储的值对上的某种谓词决定的的一种队列 <queue>

**映射**(map)由{键，值}对组成的集合，以某种作用于键对上的谓词排列 <map>

**多重映射**(multimap) 允许键对有相等的次序的映射 <map>

## STL容器效率问题String/vector/list/deque

C++ STL 之所以得到广泛的赞誉，也被很多人使用，不只是提供了像vector, string, list等方便的容器，更重要的是STL封装了许多复杂的数据结构算法和大量常用数据结构操作。

vector封装数组，list封装了链表，map和 set封装了二叉树等，在封装这些数据结构的时候，STL按照程序员的使用习惯，以成员函数方式提供的常用操作，如：插入、排序、删除、查找等。C++ STL中标准关联容器set, multiset, map, multimap内部采用的就是一种非常高效的平衡检索二叉树：红黑树，也称为RB树(Red-Black Tree)。RB树的统计性能要好于一般的平衡二叉树，所以被STL选择作为了关联容器的内部结构。

顺序存储容器：string、vector、list、deque

关联存储容器：map底层采用的是树型结构,多数使用平衡二叉树实现,查找某一值是常数时间,遍历起来效果也不错, 只是每次插入值的时候,会重新构成底层的平衡二叉树,效率有一定影响.

set 和map都是无序的保存元素,只能通过它提供的接口对里面的元素进行访问

set:集合,用来判断某一个元素是不是在一个组里面,使用的比较少

map:映射,相当于字典,把一个值映射成另一个值,如果想创建字典的话使用它好了

## string

string 是basic\_string<char> 的实现,在内存中是连续存放的.为了提高效率,都会有保留内存,如string s= "abcd",这时s使用的空间可能就是255, 当string再次往s里面添加内容时不会再次分配内存.直到内容>255时才会再次申请内存,因此提高了它的性能.当内容>255时,string会先分配一个新内存,然后再把内容复制过去,再复制先前的内容.(Copy-On-Write技术)

对string的操作:

1、如果是添加到最后时,一般不需要分配内存,所以性能最快;

2、如果是对中间或是开始部分操作,如往那里添加元素或是删除元素,或是代替元素,这时需要进行内存复制,性能会降低.

3、如果删除元素,string一般不会释放它已经分配的内存,为了是下次使用时可以更高效.

由于string会有预保留内存,所以如果大量使用的话,会有内存浪费,这点需要考虑.还有就是删除元素时不释放过多的内存,这也要考虑. string中内存是在堆中分配的,所以串的长度可以很大,而char[]是在栈中分配的,长度受到可使用的最大栈长度限制. 如果对知道要使用的字符串的最大长度,那么可以使用普通的char[],实现而不必使用string. string用在串长度不可知的情况或是变化很大的情况.如果string已经经历了多次添加删除,现在的尺寸比最大的尺寸要小很多,想减少string使用的大小,可以使用:

string s = "abcdefg";

string y(s); // 因为再次分配内存时,y只会分配与s中内容大一点的内存,所以浪费不会很大

s.swap(y); // 减少s使用的内存

如果内存够多的话就不用考虑这个了 。capacity是查看现在使用内存的函数。大家可以试试看string分配一个一串后的capacity返回值,还有其它操作后的返回值

**1：string对象的定义和初始化以及读写**

string s1;      默认构造函数，s1为空串

string s2(s1);   将s2初始化为s1的一个副本

string s3("valuee");   将s3初始化一个字符串面值副本

string s4(n,'c');   将s4 初始化为字符'c'的n个副本

cin>>s5;  读取有效字符到遇到空格

getline(cin,s6);  读取字符到遇到换行，空格可读入，知道‘\n’结束（练习在下一个代码中），

getline(cin,s7,'a'); 一个直到‘a’结束，其中任何字符包括'\n'都能够读入

**2：string对象操作**

**s.empty()**  判断是否为空，bool型

**s.size()** 或 s.length() 返回字符的个数

**s[n]**  返回位置为n的字符，从0开始计数

**s1+s2** 连接，看下面例子：

    可用此方法给字符串后面添加字符如：s=s+'a';

    a:  string s2=s1+", ";  //对，把一个string对象和一个字符面值连接起来是允许的

    b:  string s4="hello "+", ";   //错，不能将两个字符串面值相加

    c:  string s5=s1+", "+"world";   //对，前面两个相加相当于一个string对象；

    d:  string s6="hello" + ", " +  s2;  //错

（注：字符串尾部追加还可用s.append("abc")函数添加）

**s1=s2**  替换

**s1==s2**  相等，返回true或false

!=,<,<=,>,>=  字符串比较，两个字符串短的与长的前面匹配，短的小于长的

**3:string对象中字符的处理（头文件cctype）**

**isalnum(c)**  如果c是字母或数字，返回 true

**isalpha(c)**  如果c是字母，返回true

**iscntrl(c)**  c是控制符，返回true

**isdigit(c)**  如果c是数字，返回true

**isgraph(c)**  如果c不是空格，则可打印，，则为true

**islower(c)**  如果c是小写字母，则为true

**isupper(c)**  如果c是大写字符，则为true

**isprint(c)**  如果c是可打印的字符，则为true

**ispunct(c)**  如果c是标点符号，则为true

**isspace(c)** 如果c是空白字符，则为true

**isxdigit(c)** 如果c是十六进制数，则为true

**tolower(c)** 如果c是大写字符，则返回其小写字母，否则直接返回c

**toupper(c)**  跟tolower相反

**4：string对象中一些函数**

/\*-------------------------插入函数----------------------------------包括迭代器操作和下标操作，下标操作更灵活\*/

s.insert( it , p );  把字符串p插入到it的位置

s.insert(p,n,t)；   迭代器p元素之前插入n个t的副本

s.insert(p,b,e);      迭代器p元素之前插入迭代器b到e之间的所有元素

s.insert(p,s2,poe2,len); 在下标p之前插入s2下标从poe2开始长度为len的元素

s.insert(pos,cp,len);  下标pos之前插入cp数组的前len个元素。

/\*-----------------------替换函数-------------------------------\*/

s.assign(b,e);  用迭代器b到e范围内的元素替换s

s.assign(n,t)；  用n个t的副本替换s

a.assign(s1,pos2,len);从s1的下标pos2开始连续替换len个。

s.replace ( 3 , 3 , " good " ) ;   从第三个起连续三个替换为good

s.substr(i,j)   截取s串中从i到j的子串  //string::npos  判断字符串是否结束

/\*-----------------------删除函数-----------------------------\*/

s.erase( 3 )||s.erase ( 0 , 4 ) ;  删除第四个元素或第一到第五个元素

/\*----------------------其他函数-----------------------------\*/

s.find ( " cat " ) ;  超找第一个出现的字符串”cat“，返回其下标值，查不到返回 4294967295，也可查找字符；

s.append(args); 将args接到s的后面

s.compare ( " good " ) ;  s与”good“比较相等返回0，比"good"大返回1，小则返回-1；

reverse ( s.begin(), s.end () );  反向排序函数，即字符串反转函数

## vector

**实现：**

**它以两个迭代器start和finish分别指向配置得来的连续空间中目前已经被使用的空间。迭代**

**器end\_of\_storage指向整个连续空间的尾部。**  
为了降低空间配置时候的速度，vector实际配置的大小可能比客户端需求量更大一些，以备将来

可能的扩充。如果vector在增加一个元素的时候，超过了自身最大的容量。vector则将自身的容量扩充至原来的两倍。

**扩充空间需要经过的步骤：重新配置空间，元素移动，释放旧内存空间。**（vector机制将旧空

间中的备用空间也拷贝到新空间来了，感觉没必要）

**一旦vector空间重新配置，则指向原来vector的所有迭代器都失效了，因为vector的地址改**

**变了。**

**定义：**

1. vector是表示可变大小数组的序列（顺序）容器。
2. 就像数组一样，也是在堆中分配内存,元素连续存放,有保留内存,如果减少大小后内存也不会

释放.如果新值大于当前大小时才会再分配内存。vector也采用的连续存储空间来存储元素，也就是意味着可以采用下标对vector的元素进行访问，和数组一样高效。

但它比数组更优越，一般来说数组不能动态拓展，因此在程序运行的时候不是浪费内存，就是造成越界。它的大小是可以动态改变的，而且它的大小会被容器自动处理：vector在每次扩张容量的时候，将容量扩展2倍，这样对于小对象来说，效率是很高的。

1. 本质讲，vector使用动态分配数组来存储它的元素。当新元素插入时候，这个数组需要被重

新分配大小为了增加存储空间。其做法是，分配一个新的数组，然后将全部元素移到这个数组。就时间而言，这是一个相对代价高的任务，因为每当一个新的元素加入到容器的时候，vector并不会每次都重新分配大小。

1. vector分配空间策略：vector会分配一些额外的空间以适应可能的增长，因为存储空间比实

际需要的存储空间更大。不同的库采用不同的策略权衡空间的使用和重新分配。但是无论如何，重新分配都应该是对数增长的间隔大小，以至于在末尾插入一个元素的时候是在常数时间的复杂度完成的。因此，vector占用了更多的存储空间，为了获得管理存储空间的能力，并且以一种有效的方

式动态增长。

1. vector的属性与string差不多,同样可以使用capacity看当前保留的内存,使用swap来减少

它使用的内存。

1. 访问方面,对任何元素的访问都是O(1),也就是是常数的,所以vector常用来保存需要经常进

行随机访问的内容,并且不需要经常对中间元素进行添加删除操作。比起lists和forward\_lists统一的迭代器和引用更好。

1. 与其它动态序列容器相比（deques, lists and forward\_lists），vector在访问元素的时候

更加高效，它的随机访问快，在末尾添加和删除元素相对高效,因为此时一般不需要移动内存,只有保留内存不够时才需要；但是在中间插入和删除慢，效率低。对中间和开始处进行添加删除元素操作需要移动内存,如果你的元素是结构或是类,那么移动的同时还会进行构造和析构操作，所以性能不高。最好将结构或类的指针放入vector中，而不是结构或类本身，这样可以避免移动时的构造与析构。

**用法：**

**1.头文件**

#include <vector>

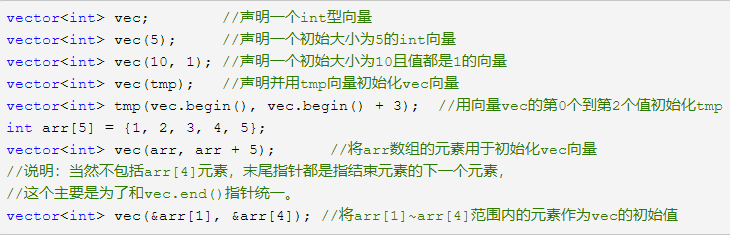
**2.定义方式**

a) vector<int>v1;//vector元素为 int 型

b) vector<string>v2;// vector元素为string型

c) vector<node>v3;//入队元素为结构体型，结构体可以自行定义

vector<int>::iterator it;//定义一个迭代器



**3.常用操作**

v1.push\_back() //在数组的最后添加一个数据

v1.pop\_back() //去掉数组的最后一个数据   
v1.front() 　　　　//返回第一个元素(栈顶元素)

v1.begin() //得到数组头的指针，用迭代器接受

v1.end() //得到数组的最后一个单元+1的指针，用迭代器接受

v1.clear() // 移除容器中所有数据

v1.empty() //判断容器是否为空

v1.erase(pos) //删除pos位置的数据

v1.erase(beg,end)// 删除[beg,end)区间的数据

v1.size() //回容器中实际数据的个数  
v1.insert(pos,data) //在pos处插入数据

**4.基本操作**

(1). 容量

* 向量大小： vec.size();
* 向量最大容量： vec.max\_size();
* 更改向量大小： vec.resize();
* 向量真实大小： vec.capacity();
* 向量判空： vec.empty();
* 减少向量大小到满足元素所占存储空间的大小： vec.shrink\_to\_fit();

(2). 修改

* 多个元素赋值： vec.assign(); //类似于初始化时用数组进行赋值
* 末尾添加元素： vec.push\_back();
* 末尾删除元素： vec.pop\_back();
* 任意位置插入元素： vec.insert();
* 任意位置删除元素： vec.erase();
* 交换两个向量的元素： vec.swap();
* 清空向量元素： vec.clear();

(3)迭代器

* 开始指针：vec.begin();
* 末尾指针：vec.end(); //指向最后一个元素的下一个位置
* 指向常量的开始指针： vec.cbegin(); //意思就是不能通过这个指针来修改所指的内容，但还是可以通过其他方式修改的，而且指针也是可以移动的。
* 指向常量的末尾指针： vec.cend();

(4)元素的访问

* 下标访问： vec[1]; //并不会检查是否越界
* at方法访问： vec.at(1); //以上两者的区别就是at会检查是否越界，是则抛出out of range异常
* 访问第一个元素： vec.front();
* 访问最后一个元素： vec.back();
* 返回一个指针： int\* p = vec.data(); //可行的原因在于vector在内存中就是一个连续存储的数组，所以可以返回一个指针指向这个数组。这是是C++11的特性。

**5.算法**

* 遍历元素

vector<int>::iterator it;

for (it = vec.begin(); it != vec.end(); it++)

cout << \*it << endl;

//或者

for (size\_t i = 0; i < vec.size(); i++) {

cout << vec.at(i) << endl;

}

* 元素翻转

#include <algorithm>

reverse(vec.begin(), vec.end());

* 元素排序

#include <algorithm>

sort(vec.begin(), vec.end()); //采用的是从小到大的排序

//如果想从大到小排序，可以采用上面反转函数，也可以采用下面方法:

bool Comp(const int& a, const int& b) {

return a > b;

}

sort(vec.begin(), vec.end(), Comp);

## list

list就是双向链表,元素也是在堆中存放,每个元素都是放在一块内存中.list没有空间预留习惯,所以每分配一个元素都会从内存中分配,每删除一个元素都会释放它占用的内存。list在哪里添加删除元素性能都很高,不需要移动内存,当然也不需要对每个元素都进行构造与析构了,所以常用来做随机操作容器.但是访问list里面的元素时就开始和最后访问最快访问其它元素都是O(n) ,所以如果需要经常随机访问的话,还是使用其它的好

## [vector和list的区别](http://www.cnblogs.com/shijingjing07/p/5587719.html)

1.vector数据结构

vector和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间复杂度为o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为o(n)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。

2.list数据结构

list是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以list的随机存取非常没有效率，时间复杂度为o(n);但由于链表的特点，能高效地进行插入和删除。

## deque

双端队列,也是在堆中保存内容的.每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是list和vector的结合品,不过确实也是如此。deque可以让你在前面快速地添加删除元素,或是在后面快速地添加删除元素,然后还可以有比较高的随机访问速度

## vector/list/deque比较

**vector：**对象数量变化小，对象简单，随机访问元素频繁，而不在乎插入和删除的效率。可以快速地在最后添加删除元素,并可以快速地访问任意元素。

**list：**对象数量变化大，对象复杂，插入和删除频繁，而不关心随即存取。可以快速地在所有地方添加删除元素,但是只能快速地访问最开始与最后的元素。

**deque：**需要随即存取，而且关心两端数据的插入和删除。在开始和最后添加元素都一样快,并提供了随机访问方法,像vector一样使用[]访问任意元素,但是随机访问速度比不上vector快,因为它要内部处理堆跳转.deque也有保留空间.另外,由于deque不要求连续空间,所以可以保存的元素比vector更大,这点也要注意一下.还有就是在前面和后面添加元素时都不需要移动其它块的元素,所以性能也很高

## set&multiset

有序集合，使用平衡二叉树存储，按照给定的排序规则（默认按less排序）对set中的数据进行排序；set中不允许有重复元素，multiset中运行有重复元素；两者不支持直接存取元素的操作；因为是自动排序，查找元素速度比较快；不能直接改变元素值，否则会打乱原本正确的顺序，必须先下删除旧元素，再插入新的元素。

## map&multimap

字典库，一个值映射成另一个值，使用平衡二叉树存储，按照给定的排序规则对map中的key值进行排序；map中的key值不允许重复，multimap中的key允许重复；根据已知的key值查找元素比较快；插入和删除操作比较慢。

set和multiset的区别就是后者允许键值重复，map和multimap也一样。实现时通过insert\_equal()和insert\_unique来控制。

## 为何map和set的插入删除效率比用其他序列容器高

map和set容器内所有元素都是以红黑树节点的方式来存储，其节点结构和链表差不多，指向父节点和子节点。因此插入的时候只需要稍做变换，把节点的指针指向新的节点就可以了。删除的时候类似，稍做变换后把指向删除节点的指针指向其他节点就OK了。这里的一切操作就是指针换来换去，和内存移动没有关系。

## 为何map和set不能像vector一样有个reserve函数来预分配数据？

在map和set内部存储的已经不是元素本身了，而是包含元素的节点。也就是说map内部使用的Alloc并不是map<Key, Data, Compare, Alloc>声明的时候从参数中传入的Alloc。例如：map<int, int, less<int>, Alloc<int> > intmap;这时候在intmap中使用的allocator并不是Alloc<int>, 而是通过了转换的Alloc，具体转换的方法时在内部通过Alloc<int>::rebind重新定义了新的节点分配器。

## 当数据元素增多时，map和set的插入和搜索速度变化如何？（10000到20000个比较）

在map和set中查找是使用二分查找，也就是说，如果有16个元素，最多需要比较4次就能找到结果，有32个元素，最多比较5次。那么有10000个呢？最多比较的次数为log10000，最多为14次，如果是20000个元素呢？最多不过15次。看见了吧，当数据量增大一倍的时候，搜索次数只不过多了1次，多了1/14的搜索时间而已。

**STL迭代器**

迭代器是一种检查容器内元素并遍历元素的数据类型。迭代器是一种行为类似指针的对象，它提供类似指针的功能，对容器的内容进行走访，而指针的各种行为中最常见也最重要的便是“内容提领”和“成员访问”。因此，迭代器最重要的编程工作就是对operator\*和operator->进行重载工作。

**1、容器的迭代器类型**

几乎STL提供的所有算法都是通过迭代器存取元素序列进行工作的，每一个容器都定义了其本身所专有的迭代器类型，用以存取容器中的元素，如vector：

vector<int>::iterator iter; // 将iter声明为int类型的向量迭代器

这条语句定义了一个名为iter的变量，它的数据类型是由vector定义的iterator类型。每个标准库容器类型都定义了一个名为iterator的成员，这里的iterator与迭代器的实际类型相同。术语迭代器通常指的是概念上的迭代器，而术语迭代器类型则是有容器定义的具体的iterator的类型，如vector。   
**常见的迭代器类型主要有：迭代器类型iterator、元素的只读迭代器类型const\_iterator、按逆序寻址元素的迭代器reverse\_iterator、元素的只读逆序迭代器const\_reverse\_iterator。**

**2、迭代器的begin和end操作**

vector<int>::iterator iter = vec.begin();

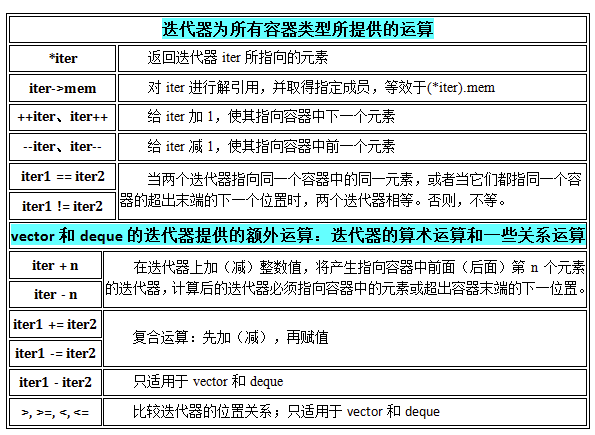
begin返回的迭代器指向容器内第一个元素，end返回的迭代器指向容器内“最后一个元素的下一个位置”

end操作返回的迭代器并不指向vector中任何实际的元素，它只是起到一个哨兵的作用，表示我们已经处理完vector中的所有元素。

rbegin返回一个逆序迭代器，它指向容器的最后一个元素，rend返回一个逆序的迭代器，它指向容器的第一个元素前面的位置。

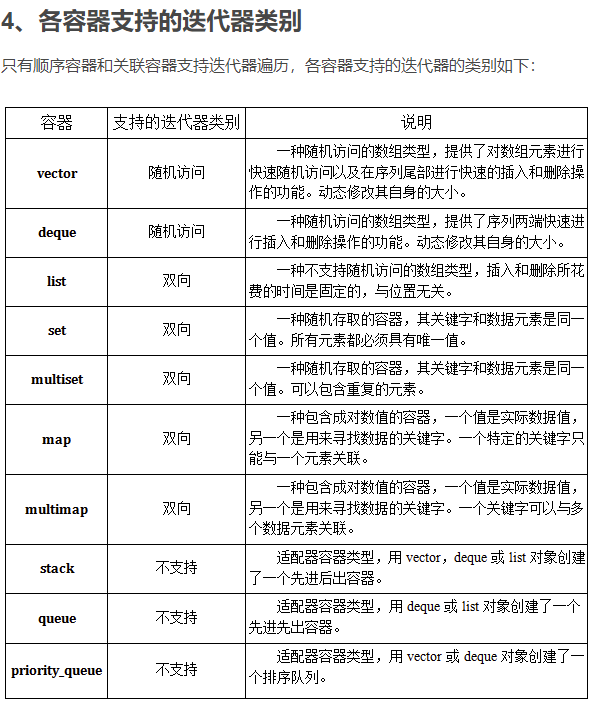
**3、迭代器的自增和解引用**

迭代器类型可使用解引用操作符（\*操作符）来访问迭代器所指向的元素，解引用操作符返回迭代器当前所指向的元素。但是，不能对end操作返回的迭代器进行解引用。迭代器也可以使用自增操作符向前移动迭代器指向容器中下一个元素，这在形式上与int型对象的自增操作类似。



关系操作符只适用于vector和deque容器，这是因为只有这两种容器为其元素提供快速、随机的访问。它们确保可根据元素位置直接有效地访问指定的容器元素。这两种容器都支持通过元素的位置实现随机访问，因此它们的迭代器可以有效地实现算术和关系运算。

C++语言使用一对迭代器来标记迭代器的范围。迭代器范围：[first, last)是一个左闭合区间，表示范围从first开始，到last结束，但不包括last，即last指向容器内最后一个元素的下一个元素。注意：如果first不等于last，则对first反复做自增运算必须能够到达last；否则，即last位于first之前，则将发生未定义行为。迭代器范围使用左闭合的意义：因为这样可以统一表示空集，就无需特别处理。当first和last相等时，迭代器的范围是空。



## 每次insert之后以前保存的iterator不会失效

iterator这里就相当于指向节点的指针，内存没有变，指向内存的指针怎么会失效呢(当然被删除的那个元素本身已经失效了)。相对于vector来说，每一次删除和插入，指针都有可能失效，调用push\_back在尾部插入也是如此。因为为了保证内部数据的连续存放，iterator指向的那块内存在删除和插入过程中可能已经被其他内存覆盖或者内存已经被释放了。即使时push\_back的时候，容器内部空间可能不够，需要一块新的更大的内存，只有把以前的内存释放，申请新的更大的内存，复制已有的数据元素到新的内存，最后把需要插入的元素放到最后，那么以前的内存指针自然就不可用了。特别时在和find等算法在一起使用的时候，牢记这个原则：不要使用过期的iterator。