



c++/c

c++三大特性

封装

最开始接触代码是C语言，那么开始写一些逻辑代码的时候会很麻烦，因为你要在函数中定义变量，然后按顺序写对应的逻辑，接着可以将逻辑封装成函数。当时会感觉很麻烦，因为很散装，知道后面学了struct结构体，把对应逻辑需要的数据可以放到一个结构体里面，这样就会比较好看。接着出现的问题就是数据封装到了一起，但是处理数据对应的逻辑即函数却还是在外面。因此就有了将数据和对应逻辑进行封装的类的出现。

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，可以隐藏实现细节包括包含私有成员，使得代码模块增加安全指数，同时提高软件的可维护性和可修改性。

所以总结来说封装这个特性包含两三点：

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

继承

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的函数，变量等。

一般来说有如下三种继承方式：

1. 单一继承：继承一个父类，这种继承称为单一继承，这也是我们用的做多的继承方式。
2. 多重继承：一个派生类继承多个基类，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obj.classA::i = 1；
3. 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。后面有将具体虚继承怎么做。

此外还有继承权限的问题， 如下图：

+	父类中访问权限		继承方式		子类中访问权限		多态
	public	protected	public	protected	private	no access	
	public	protected	public	protected	protected	protected	可以简单概括为“一个接口，多种方法”，即用的是同一个接口，但是效果各不相同
	protected	private		protected	protected	no access	
	private	no access		protected	protected	no access	
	public	protected	protected	protected	protected	protected	
	protected	private		protected	protected	no access	
	private	no access		protected	protected	no access	
	public	protected	private	protected	protected	protected	
	protected	private		protected	protected	no access	
	private	no access		protected	protected	no access	

同， 多态有两种形式的多态， 一种是静态多态， 一种是动态多态。

静态多态。静态多态的设计思想：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， 静态多态本质上就是模板的具现化。

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。具体实现就是c++的虚函数。

多态是以封装和继承为基础实现的性质， 一个形态的多种表现方式。硬要解释的话可以说是一个接口， 多个功能， 在用父类指针调用函数时， 实际调用的是指针指向的实际类型（子类）的成员函数。

c++多态有以下几种：

1. 重载。函数重载和运算符重载， 编译期。
2. 虚函数。子类的多态性， 运行期。

在继承关系中， 对于父类的方法我们也同样使用。但是正常来说， 我们希望方法的行为取决于调用方法的对象， 而不是指针或引用指向的对象有关。

3. 模板， 类模板， 函数模板。编译期

比如你家有亲属结婚了， 让你们家派个人来参加婚礼， 邀请函写的是让你爸来， 但是实际上你去了， 或者你妹妹去了， 这都是可以的， 因为你们代表的是你爸， 但是在你们去之前他们也不知道谁会去， 只知道是你们家的人。可能是你爸爸， 可能是你们家的其他人代表

你爸参加。这就是多态。

面向对象和面向过程语言的区别

首先要知道这两个都是一种编程思想

面向过程就是分析出解决问题所需要的步骤，然后用函数把这些步骤一步一步实现，使用的时候一个一个依次调用就可以了。

面向对象是把构成问题事务分解成各个对象，建立对象的目的不是为了完成一个步骤，而是为了描述某个事物在整个解决问题的步骤中的行为。

举一个例子：

例如五子棋，面向过程的设计思路就是首先分析问题的步骤：1、开始游戏，2、黑子先走，3、绘制画面，4、判断输赢，5、轮到白子，6、绘制画面，7、判断输赢，8、返回步骤2，9、输出最后结果。把上面每个步骤用分别的函数来实现，问题就解决了。

而面向对象的设计则是从另外的思路来解决问题。整个五子棋可以分为1、黑白双方，这两方的行为是一模一样的，2、棋盘系统，负责绘制画面，3、规则系统，负责判定诸如犯规、输赢等。第一类对象（玩家对象）负责接受用户输入，并告知第二类对象（棋盘对象）棋子布局的变化，棋盘对象接收到了棋子的变化就要负责在屏幕上面显示出这种变化，同时利用第三类对象（规则系统）来对棋局进行判定。

可以明显地看出，面向对象是以功能来划分问题，而不是步骤。同样是绘制棋局，这样的行为在面向过程的设计中分散在了总多步骤中，很可能出现不同的绘制版本，因为通常设计人员会考虑到实际情况进行各种各样的简化。而面向对象的设计中，绘图只可能在棋盘对象中出现，从而保证了绘图的统一。功能上的统一保证了面向对象设计的可扩展性。

面向过程

优点：性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源；比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

缺点：没有面向对象易维护、易复用、易扩展

面向对象

优点：易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦

合的系统，使系统 更加灵活、更加易于维护

缺点：性能比面向过程低

结构体(struct)和共同体(union)的区别

结构体struct：把不同类型的数据组合成一个整体。struct里每个成员都有自己独立的地址。
sizeof(struct)是内存对齐后所有成员长度的加和。（引申出[内存对齐](#)的问题）

共同体union：各成员共享一段内存空间，一个union变量的长度等于各成员中最长的长度，以达到节省空间的目的。所谓的共享不是指把多个成员同时装入一个union变量内，而是指该union变量可被赋予任一成员值，但每次只能赋一种值，赋入新值则冲去旧值。sizeof(union)是最长的数据成员的长度。

总结：struct和union都是由多个不同的数据类型成员组成，但在任何同一时刻，union中只存放了一个被选中的成员，而struct的所有成员都存在。在struct中，各成员都占有自己的内存空间，它们是同时存在的。一个struct变量的总长度等于所有成员长度之和。*在Union中，所有成员不能同时占用它的内存空间，它们不能同时存在。*Union变量的长度等于最长的成员的长度。对于union的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于struct的不同成员赋值是互不影响的。

struct的内存对齐规则

为什么要字节对齐？

需要字节对齐的根本原因在于CPU访问数据的效率问题。假如没有字节对齐，那么一个double类型的变量会存储在4-11上（正常是0-7）这样计算机取这个数据的会会取两次，降低效率。而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如sparc系统，如果取未对齐的数据会发生错误。

对齐规则

由于在x86下，GCC默认按4字节对齐，但是可以使用 `__attribute__` 选项改变对齐规则，vs studio上用 `#pragma pack (n)` 方式改变

举例子：

```

//sizeof stu = 4 +4 + 12 = 20
struct stu{
    char sex;                //4
    int length;              //4
    char name[10];           //12
};

// size of str1 = 1 + 1+(2)+4 = 8
//两个char后需要再补充两个字节凑够4字节
struct node1
{
    char c1;
    char c2;
    int a;
}str1 ;

// 5+2+1+4 = 12
//char前4个元素占4字节，第5个元素和short一共3字节，需要补1个字节
struct str4 {
    char c1[5];
    short c;
    int b;
}str4 ;

// 1+(7)+8+1+(7) = 24
struct str6 {
    char c1;
    double a;
    char c2;
}str6 ;

```

易错点

```
//sizeof = 8
struct str{
    char p;
    int a;
    int b[0];
}

//sizeof = 4
struct str{
    int b[0];
}

//sizeof = 1
struct str{
}
}
```

上面三个结构体都包含空数组，空数组指的是长度为0的数组 `int[]` 或 `int[0]`

这种定义只能在类或者结构体中定义，在外部是非法定义。空数组不占空间，也无需初始化

空数组名是一个指针，（但是又不占空间）指向一个位置；对于结构体，空数组名这个指针指向了前面一个成员结束的下一个空间。

不能再函数中定义 `int a[0]` 或 `int a[]`;这种

arr和&arr[0]和&arr的不同

首先如果打印的话，三个打印的完全一样，都是数组首元素的地址。

首先 `&arr` 应该是整个元素的地址，但是打印出来却是首元素的地址，不同之处在于对地址做加法运算后有不同，如下：

```
int arr[10]={0};
printf("%p\n",arr);//首元素的地址
printf("%p\n",arr+1);

printf("%p\n",&arr[0]);//首元素的地址
printf("%p\n",&arr[0]+1);

printf("%p\n",&arr);//整个数组元素的地址
printf("%p\n",&arr+1);
```

输出首元素地址都是相同的，arr+1和&arr[0]+1都是只移动了4个字节，但是&arr+1移动了40个字节

结论：&arr代表的是整个数组的地址，虽然它具体表现为首个元素的地址，但是在对其进行操作时，是以整个数组为单位的。

补充

arr 本身是左值（但不可仅凭此表达式修改），指代数组对象。不过 arr 会在大多数场合隐式转换成右值表达式 &(arr[0])，为指针类型，指向 arr[0]。&arr 是右值表达式，为指针类型，指向 arr 本身。简单来说就是 arr 本身不是地址而是指代整个数组，只不过会隐式转成指针罢了。

char a,char a[],char *a,char *[],char **a 之间的区别

1. char a

定义了一个存储空间，存储的是char类型的变量

2. char a[]

是一个字符数组，数组中的每一个元素是一个char类型的数据

3. char *a

字符串的本质（在计算机眼中）是其第一个字符的地址，c和c++中操作字符串是通过内存中其存储的首地址来完成的

对于char a[]来说a代表的是数组的首地址，那么对char *a来说a代表的也是字符串的首地址

因此char a[]和char *a可以放到一块看，这两个没有本质区别。

但是要注意对于char s[]和char* a我们可以有 a=s，但不能有 s=a，因为创建数组的时候s的地址不为空已经确定，但是a是一个空指针，不能将非空的地址指向空指针

4. char *a[]

* 的优先级是低于 [] 的，因此要先看 a[] 再看 *

因此这是一个char数组，数组中的每一个元素都是指针，这些指针指向char类型

```
char *a[ ] = {"China","French","America","German"}
```

5. char **a

两个**代表相同的优先级，因此从右往左看，即 char*(*a)

char *a不就是一个字符串数组，a代表首地址。那么char * (*a)就是和char *a[]一样的数据结构

一维数组名和二维数组名的区别

不管是一维还是多维数组，都是内存中一块线性连续空间，因此在内存级别上，其实都只是一维。

所以一维数组名是指向该数组的指针，二维数组名也是指向该数组的指针，但是+1之后，跳过的是一行。

问：二维数组名为什么不能直接赋值给二级指针？

答：一句话来说就是二维数组名是行指针，也就是指向数组的指针。而二级指针是指向指针的指针，它们不是同一类型。

定义一维数组 `int a[i]` 和二维数组 `int b[i][j]`，a相当于 `int (*)`，而b相当于 `int (*)(j)`。想要获得 `a[i]` 中第 x 个元素，可以直接使用 `*(a+x)`。而想要获得 `b[i][j]` 中第 x 行第 y 个元素，则需用 `*(*(b+x)+y)`，因为 b 相当于数组指针，`(b+x)` 则是指向第 x 个数组的指针，注意，是指向数组，而不是数组元素！所以 `*(b+x)` 获得的是第 x 个数组的数组名，即该数组的首元素地址，这时再结合偏移量 y 就可以取得该元素。

数组指针和指针数组

其实就是数组的指针和指针的数组

数组的指针：指向一个数组的指针就是数组指针

指针的数组：一个数组的每一个元素都是指针

C++内存布局/程序分段

也可以叫做进程逻辑地址空间

内存从上到下分别是：

- 栈stack |高地址|
- 堆heap
- bss段
- data段
- 代码段text |低地址|

栈：保存函数的局部变量，参数以及返回值。在函数调用后，系统会清楚栈上保存的栈帧和局部变量，函数参数等信息。栈是从高到低增长的。

堆：动态内存分配的都放在堆上。堆是从低到高的。

bss段：（Block Started by Symbol）存放程序中未初始化的全局变量的一块内存区域，在程序载入时由内核置为0。

data段：static变量和所有初始化的全局变量都在data段中。

bss段和data段都是静态内存分配，也就是说在编译的时候自动分配的。

bss和data段也有一种说法合起来叫数据段，有三种类型：

1. 只读数据段，常量与const修饰的全局变量
2. 可读可写数据段，存放初始化的全局变量和static变量
3. bss段，存放未初始化的全局变量

text段：代码段，text段在内存中被映射为只读，但.data和.bss是可写的。由编译器在编译连接时自动计算的，当你在链接定位文件中将该符号放置在代码段后，那么该符号表示的值就是代码段大小，编译连接时，该符号所代表的值会自动代入到源程序中。

static 和const分别怎么用，类里面static和const可以同时修饰成员函数吗

- static

- static对于变量

- a. 局部变量

在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。

内存中的位置：data段

初始化：局部的静态变量只能被初始化一次

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，作用域随之结束。

当static用来修饰局部变量的时候，它就**改变了局部变量的存储位置（从原来的栈中存放改为静态存储区）及其生命周期（局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问），但未改变其作用域。**

- b. 全局变量

在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。

内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

初始化：未经初始化的全局静态变量会被程序自动初始化为0

作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。（只能在本文件中存在和使用）

全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的（在其他源文件中使用加上extern关键字重新声明即可）。而**静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。**

- static对于函数

修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使

用。在多人开发项目时，为了防止与他人命名空间里的函数重名，可以将函数定位为 static。（和全局变量一样限制了作用域而已）

- static对于类

- a. 成员变量

用static修饰类的数据成员实际使其成为类的全局变量，会被类的所有对象共享，包括派生类的对象。

因此，**static成员必须在类外进行初始化，而不能在构造函数内进行初始化。不过也可以用const修饰static数据成员在类内初始化。**

- b. 成员函数

用static修饰成员函数，使这个类只存在这一份函数，所有对象共享该函数，不含this指针。

静态成员是可以独立访问的，也就是说，无须创建任何对象实例就可以访问。

不可以同时用const和static修饰成员函数。

- const

- i. const修饰变量：限定变量为不可修改。

- ii. const修饰指针：指针常量和指向常量的指针

- iii. const和函数：有以下几种形式

```
const int& fun(int& a); //修饰返回值
int& fun(const int& a); //修饰形参
int& fun(int& a) const{} //const成员函数
```

- iv. const和类：①const修饰成员变量，在某个对象的声明周期内是常量，但是对于整个类而言是可以改变的。因为类可以创建多个对象，不同的对象其const成员变量的值是不同的。切记，**不能在类内初始化const成员变量**，因为类的对象没创建前，编译器并不知道const成员变量是什么，因此const数据成员只能在初始化列表中初始化。②const修饰成员函数，主要目的是防止成员函数修改成员变量的值，即该成员函数并不能修改成员变量。③const对象，常对象，常对象只能调用常函数。

- v. 限定成员函数不可以修改任何数据成员

- static和const可以同时修饰成员函数吗？

答：不可以。C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。两

者的语意是矛盾的。**static**的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而**const**的作用是确保函数不能修改类的实例的状态，与类型的静态变量没有关系。因此不能同时用它们。

static初始化时机和线程安全问题

先说在C语言中：

静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，在编译阶段分配好了内存之后就进行初始化，在程序运行结束时变量所处的全局内存会被回收。所以在c语言中无法使用变量对静态局部变量进行初始化。

再说C++和C语言的区别：

c++主要引入了类这种东西，要进行初始化必须考虑到相应的构造函数和析构函数，而且很多时候构造或者析构函数中会指定我们定义的操作，并非简单的分配内存。因此为了造成不必要的影响（一些我不需要的东西被提前构造出来）所以c++规定全局或者静态对象在首次用到的时候才会初始化。

所以c++整了两种初始化的情况，我理解就是编译初始化和运行初始化。

编译初始化也叫静态初始化。对全局变量和const类型的初始化主要是，叫做zero initialization 和 const initialization，静态初始化在程序加载的过程中完成。从具体实现上看，zero initialization 的变量会被保存在 bss 段，const initialization 的变量则放在 data 段内，程序加载即可完成初始化，这和 c 语言里的全局变量静态变量初始化基本是一致的。其次全局类对象也是在编译器初始化。

动态初始化也叫运行时初始化。主要是指需要经过函数调用才能完成的初始化或者是类的初始化，一般来说是局部静态类对象的初始化和局部静态变量的初始化。

```

8
9 class A
10 {
11 public:
12
13     A() { cout << "construct A" << endl; }
14 };
15
16 static A aa;
17
18 void func()
19 {
20     static A aaa;
21 }
22 int main(int argc, char *argv[])
23 {
24     cout << "main-----" << endl;
25     static A a;
26     cout << "======" << endl;
27     func();
28     return 0;
29 }

```

C:\Windows\system32\cmd.exe

```

construct A
main-----
construct A
=====
construct A
请按任意键继续. . .

```

<https://blog.csdn.net/hueru>

下面是我自己的实验的一段代码：

```

#include<stdio.h>
static int a;
int main()
{
    int b = 5;
    {
        //g++编译报错
        static int c = b;
        //gcc编译不报错
        static int c = b;
    }
    return 0;
}

```

如果在c语言中有局部静态变量赋值操作的话会报

错：undefined reference to ' __cxa_guard_acquire'和 ' __cxa_guard_release'。而这两个API接口恰恰是c++中保证局部静态变量运行时初始化的关键，具体参考[c++局部静态变量和线程安全具体实现参考](#)

线程安全问题：

C语言中非局部静态变量一般在main执行之前的静态初始化过程中分配内存并初始化，可以认为是线程安全的；C++11标准针规定了局部静态变量初始化是线程安全的。这里的线程安全指的是：一个线程在初始化 m 的时候，其他线程执行到 m 的初始化这一行的时候，就会挂起而不是跳过。

具体实现如下：局部静态变量在编译时，编译器的实现是和全局变量类似的，均存储在bss段中。然后编译器会生成一个 `guard_for_bar` 用来保证线程安全和一次性初始化的整型变量，是编译器生成的，存储在 bss 段。它的最低的一个字节被用作相应静态变量是否已被初始化的标志，若为 0 表示还未被初始化，则表示已被初始化(`if ((guard_for_bar & 0xff) == 0)` 判断)。 `__cxa_guard_acquire` 实际上是一个加锁的过程，相应的 `__cxa_guard_abort` 和 `__cxa_guard_release` 释放锁。

```
void foo() {
    static Bar bar;
}

//gcc 4.8.3 编译器生成的汇编代码
void foo() {
    if ((guard_for_bar & 0xff) == 0) {
        if (__cxa_guard_acquire(&guard_for_bar)) {
            try {
                Bar::Bar(&bar);
            } catch (...) {
                __cxa_guard_abort(&guard_for_bar);
                throw;
            }
            __cxa_guard_release(&guard_for_bar);
            __cxa_atexit(Bar::~~Bar, &bar, &__dso_handle);
        }
    }
    // ...
}
```

C++中局部静态变量的问题

首先说一下局部静态变量的理解

局部静态变量位于内存中的静态存储区，未经初始化的局部静态变量会自动初始化为0。但是局部静态变量的作用域还是局部作用域，定义它的函数或者代码块结束的时候，作用域也随之结束。但是该变量值不会被销毁，而是在内存中驻留下来，知道程序全部结束，这个驻留的值我们不能访问她。

静态局部变量的构造和析构

对于全局变量的构造和析构，肯定是排在首位的。

而对于局部静态变量，程序首次执行到局部静态变量的定义处时才发出构造，其构造和析构都取决于程序的执行顺序。很显然，对于分布在程序各处的静态局部变量，其构造顺序取决于它们在程序的实际执行路径上的先后顺序，而析构顺序则正好与之相反。这就有两个问题：

1. 一方面是因为程序的实际执行路径有多个决定因素（例如基于消息驱动模型的程序和多线程程序），有时是不可预知的；
2. 另一方面是因为局部静态变量分布在程序代码各处，彼此直接没有明显的关联，很容易让开发者忽略它们之间的这种关系（这是最坑的地方）。

所以我们应该尽量少使用静态变量。

函数局部静态变量的返回

```
int tmp(){
    static int b = 5;
    return b;
}
int main(){
    static int a = 0;
    int c = tmp();
    std::cout<<"a:"<<a<<std::endl;
    std::cout<<"c:"<<c<<std::endl;
}
```

对于g++编译器来说，可以返回哦

Const与指针

只有两种情况：**指向常量的指针**和**常量指针**

指向常量的指针表明不能通过解除引用运算符去改变其值，指向的变量是常量

常量指针表明初始化后的指针指向的地址是不能改变的，但这块地址上的存储的值可以改变，地址跟随一生。所以 `p2= &a`是错误的，而`*p2 = a` 是正确的。

```
const int p      //p为常量，初始化后不能更改
const int *p     // *p为常量，不能通过*p改变其内容
int const *p     //同上
int *const p     //常量指针
```

指针和引用的区别

[关于引用的本质可以看这个](#)

引用的底层本质：

从高级语言层面的概念来说：引用是变量的别名，它不能脱离被引用对象独立存在。接下来看一下底层引用到底是什么？


```

//一段引用的代码
int i=5;
int &ri=i;
ri=8;

//上述代码对应的汇编代码
int i=5;
00A013DE  mov         dword ptr [i],5           //将文字常量5送入变量i

//这一步是引用的初始化过程，对于引用十分重要
int &ri=i;
00A013E5  lea         eax,[i]                   //将变量i的地址送入寄存器eax
00A013E8  mov         dword ptr [ri],eax       //将寄存器的内容（也就是变量i的地址）送入变量ri

ri=8;
00A013EB  mov         eax,dword ptr [ri]       //将变量ri的值送入寄存器eax
00A013EE  mov         dword ptr [eax],8       //将数值8送入以eax的内容为地址的单元中
return 0;
00A013F4  xor         eax,eax

```

可以看到在汇编代码中ri这个引用的数据类型是dword(double word-4字节)。所以ri确实是一个变量，存放的是被引用的对象的地址。指针和引用的区别要看下一段代码：

```

//一段常量指针的代码
int i=5;
int* const pi=&i;
*pi=8;

//上述代码对应的汇编代码
int i=5;
011F13DE  mov             dword ptr [i],5

int * const pi=&i;
011F13E5  lea             eax,[i]
011F13E8  mov             dword ptr [pi],eax

*pi=8;
011F13EB  mov             eax,dword ptr [pi]
011F13EE  mov             dword ptr [eax],8

```

可以看到引用和常量指针的汇编代码是一模一样的，所以可以得出在底层，引用就是一个常量指针。

高级语言层面引用与指针常量的关系

1. 相同点：指针和引用在内存中都占用4个或者8个字节的存储空间，都必须在定义的时候给初始化。
2. 指针常量本身（以p为例）允许寻址，即&p返回指针常量本身的地址，*p表示被指向的对象
引用变量本身（以r为例）不允许寻址，&r返回的是被引用对象的地址，而不是变量r的地址(r的地址由编译器掌握，程序员无法直接对它进行存取)
3. 引用不能为空，指针可以为空；
4. 指针数组这一块。数组元素允许是指针但不允许是引用，主要是为了避免二义性。假如定义一个“引用的数组”，那么array[0]=8;这条语句该如何理解？是将数组元素array[0]本身的值变成8呢，还是将array[0]所引用的对象的值变成8呢？
5. 在C++中，指针和引用经常用于函数的参数传递，然而，指针传递参数和引用传递参数是有本质上的不同的：**指针传递**参数本质上是**值传递**的方式，它所传递的是一个地址值。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。而在**引用传递**过程中，被调函数的形参虽然也作为局部变量在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（指针放的是实参变量地址的副本）。

6. "sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；

用const和#define定义常量哪个更好？

#define宏常量和const常量的区别

类型和安全检查不同

宏定义是字符替换，没有数据类型的区别，同时这种替换没有类型安全检查，可能产生边际效应等错误；

const常量是常量的声明，有类型区别，需要在编译阶段进行类型检查

编译器处理不同

宏定义是一个"编译时"概念，在预处理阶段展开，不能对宏定义进行调试，生命周期结束与编译时期；

const常量是一个"运行时"概念，在程序运行使用，放在内存中的data段中。

存储方式不同

宏定义是直接替换，不会分配内存，存储于程序的代码段中；

const常量需要进行内存分配，存储于程序的数据段中

是否可以做函数参数

宏定义和const常量可以在函数的参数列表中出现

typedef与#define的区别

typedef

typedef故名思意就是类型定义的意思，但是它并不是定义一个新的类型而是给已有的类型起一个别名。主要有两个作用，第一个是给一些复杂的变量类型起别名，起到简化的作用。第二个是

定义与平台无关的类型，屏蔽不同平台之间的差异。在跨平台或操作系统的时候，只需要改typedef本身就可以

#define

define为一宏定义语句，本质就是文本替换

区别

1. 关键字typedef在编译阶段有效，由于是在编译阶段，因此typedef有类型检查的功能。#define则是宏定义，发生在预处理阶段，也就是编译之前，它只进行简单而机械的字符串替换，而不进行任何检查。
2. #define没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而typedef有自己的作用域。
3. 对指针操作不同。见下面代码

```
typedef int * pint;
#define PINT int *

int i1 = 1, i2 = 2;

const pint p1 = &i1;           //p不可更改，p指向的内容可以更改，相当于 int * const p;
const PINT p2 = &i2;           //p可以更改，p指向的内容不能更改，相当于 const int *p; 或 int const *p;
```

#define<>和#define“ ”的区别

#include<>

一般用来查找标准库文件所在目录，在编译器设置的include路径内搜索；

#include""

#include "" 的查找位置是当前源文件所在目录

要注意的一点就是，如果我们自己写的头文件，而不是标准库函数中的，那么引用这个头文件要使用 `#include""`，而不能使用 `#include<>`，因为我们自己写的头文件并不在编译器设置的路径内，使用 `#include<>` 会提示无法找到。

若 `#include ""` 查找成功，则遮蔽 `#include <>` 所能找到的同名文件；否则再按照 `#include <>` 的方式查找文件。

左值和右值

[对于左值和右值写的很详细](#)

历史

C语言中的表达式被分为 左值 和其它(函数和非对象值)，其中左值被定义为标识一个对象的表达式，在C语言中lvalue是 `locator value` 的简写，因此lvalue对应了一块内存地址。

C++出来后，C++11之前，左值遵循了C语言的分类法，但与C不同的是，其将非左值表达式统称为右值，函数为左值，因为c++有引用这个东西。并添加了引用能绑定到左值但唯有const的引用能绑定到右值的规则。

自C++11开始，对值类别又进行了详细分类，在原有左值的基础上增加了纯右值和消亡值，并对以上三种类型通过是否具名(identity)和可移动(moveable)，又增加了glvalue和rvalue两种组合类型。

c++11后的值类别

自C++11开始，表达式的值分

为 左值(lvalue, left value)、将亡值(xvalue, expiring value)、纯右值(pvalue, pure rvalue)以及两种混合类别 泛左值(glvalue, generalized lvalue) 和 右值(rvalue, right value) 五种。我们只需要关注左值，纯右值和将亡值这三种即可。

左值

lvalue 是“**loactor value**”的缩写，可意为存储在内存中、有明确存储地址（可寻址）的数据

左值是可以取地址、位于**赋值符号左边**的值，就记住，左值是表达式结束（不一定是赋值表达式）后依然存在的对象。

左值也是一个关联了名称的内存位置，允许程序的其他部分来访问它的值。

有以下特征：

1. 可通过取地址运算符获取其地址
2. 可修改的左值可用来赋值
3. 可以用来初始化左值引用

那些是左值？

1. 变量名、函数名以及数据成员名
2. 返回左值引用的函数调用
3. 由赋值运算符或复合赋值运算符连接的表达式，如(a=b, a-=b等)
4. 解引用表达式*ptr
5. 前置自增和自减表达式(++a, ++b)
6. 成员访问（点）运算符的结果
7. 由指针访问成员（->）运算符的结果
8. 下标运算符的结果([])
9. 字符串字面值("abc")

右值

rvalue 译为 "**read value**"，指那些可以提供数据值的表达式（不一定可以寻址，例如存储于寄存器中的数据）。右值有可能在内存中也有可能在寄存器中。一般来说就是活不过一行就会消失的值。

那些是右值？

1. 字面值(字符串字面值除外)，例如1, 'a', true等
2. 返回值为非引用的函数调用或操作符重载，例如：str.substr(1, 2), str1 + str2, or it++
3. 后置自增和自减表达式(a++, a--)
4. 算术表达式 (x + y;)
5. 逻辑表达式
6. 比较表达式
7. 取地址表达式
8. lambda表达式 `auto f = []{return 5;};`

左值和右值区分的一个点

从本质上理解，右值的创建和销毁由编译器幕后控制，程序员只能确保在本行代码有效的，就是右值(包括立即数)；而用户创建的，通过作用域规则可知其生存期的，就是左值(包括函数返回的局部变量的引用以及const对象)。

右值又有纯右值和将亡值的说法。非引用返回的临时变量、运算表达式产生的临时变量、原始字面量和lambda表达式等都是纯右值。而将亡值是与右值引用相关的表达式，比如，将要被移动的对象、T&&函数返回值、std::move返回值和转换为T&&的类型的转换函数的返回值等。

左值引用和右值引用

左值引用

左值引用分为：左值引用和常量左值引用(不希望被修改)

观察下列代码：

```
//左值引用
int a = 10;
int &b = a; // 定义一个左值引用变量
b = 20;     // 通过左值引用修改引用内存的值

//常量左值引用
const int temp = 10;
const int &var = temp;
```

右值引用

[贴一个超级详细的连接](#)

右值引用是C++11中新增加的一个很重要的特性，他主要用来解决C++98/03中遇到的两个问题，第一个问题就是临时对象非必要的昂贵的拷贝操作，第二个问题是在模板函数中如何按照参数的实际类型进行转发。通过引入右值引用，很好的解决了这两个问题，改进了程序性能。我从以下四行代码来理解右值引用：

第一行代码：

```
int i = getVar();  
auto f = []{return 5;;};
```

对于第一个代码，会产生两种类型的值。等号左边是一个左值，可以取地址，我们可以管控i的生命周期。等号右边会得到一个临时值，这个临时值在表达式结束之后就销毁了，这个临时值就是右值，或者叫做纯右值。

第二行代码：

```
T&& k = getVar();
```

此时的k就是右值引用，由于右值是匿名变量，我们也只能通过引用的方式来获取右值。在这里getVar()产生的临时值不会像第一行代码那样，在表达式结束之后就销毁了，而是会被“续命”，他的生命周期将会通过右值引用得以延续，和变量k的声明周期一样长。

第三行代码：

```
T(T&& a){  
    m_val = a.m_val;  
    a.m_val=nullptr;  
}
```

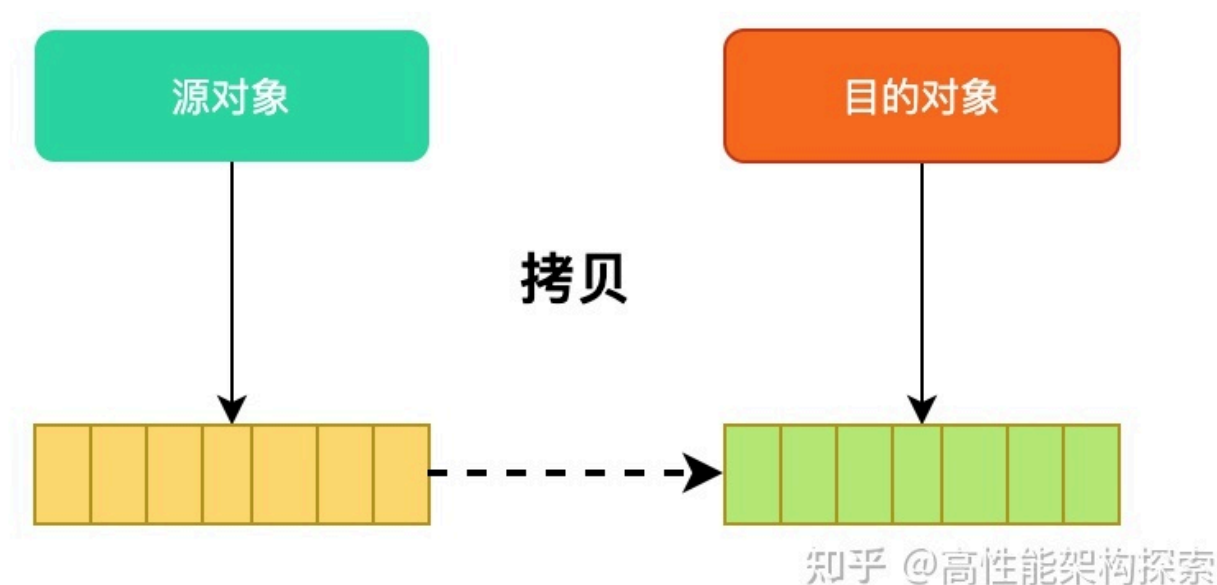
从类的移动构造开始说。由于一个带有堆内存的类必须提供一个深拷贝拷贝构造函数，因为默认的拷贝构造函数是浅拷贝，会发生“指针悬挂”的问题。如果不提供深拷贝的拷贝构造函数，上面的测试代码将会发生错误。内部的m_ptr将会被删除两次，一次是临时右值析构的时候删除一次，第二次外面构造的a对象释放时删除一次，而这两个对象的m_ptr是同一个指针，这就是所谓的指针悬挂问题。提供深拷贝的拷贝构造函数虽然可以保证正确，但是在有些时候会造成额外的性能损耗，因为有时候这种深拷贝是不必要的。这个构造函数并没有做深拷贝，仅仅是将指针的所有者转移到了另外一个对象，同时，将参数对象a的指针置为空，这里仅仅是做了浅拷贝，因此，这个构造函数避免了临时变量的深拷贝问题。

进而引出了移动语义的概念(std::move)。

当编译器看到&&的时候会判定为右值引用，对编译器来说这是一个临时变量的标识，对于临时变量来说我们仅仅做一次浅拷贝就行，不需要深拷贝，从而解决了前面提到的临时变量拷贝构造产生的性能损失的问题。这就是所谓的移动语义，右值引用的一个重要作用是用来支持移动语义

的。那我们知道了移动语义是通过右值来匹配临时值的，那么很自然会想到，普通的左值是否也能借助移动语义来优化性能呢？C++11为了解决这个问题，提供了`std::move`方法来将左值转换为右值，从而方便应用移动语义。`move`是将对象资源的所有权从一个对象转移到另一个对象，只是转移，没有内存的拷贝，这就是所谓的move语义。

`move`实际上它并不能移动任何东西，它唯一的功能是将一个左值强制转换为一个右值引用。如果是一些基本类型比如`int`和`char[10]`定长数组等类型，使用`move`的话仍然会发生拷贝（因为没有对应的移动构造函数）。所以，`move`对于含资源（堆内存或句柄）的对象来说更有意义。



第四行代码：

```

template <typename T>
void f(T&& val){
    foo(std::forward<T>(val));
}

//模板类会出现的问题
template <typename T>
void forwardValue(T& val)
{
    processValue(val); //右值参数会变成左值
}
template <typename T>
void forwardValue(const T& val)
{
    processValue(val); //参数都变成常量左值引用了
}

```

在模板函数中传入的是有个右值，但是第一个函数中变成了左值，第二个函数中变成了常量左值引用。

引入完美转发的概念。

C++11引入了完美转发解决这个问题：在函数模板中，完全依照模板的参数的类型（即保持参数的左值、右值特征），将参数传递给函数模板中调用的另外一个函数。C++11中的std::forward正是做这个事情的，他会按照参数的实际类型进行转发。

深拷贝和浅拷贝

在未定义显示拷贝构造函数的情况下，系统会调用默认的拷贝函数——即浅拷贝，它能够完成成员的一一复制。

当数据成员中没有指针时，浅拷贝是可行的；

****但当数据成员中有指针时，会出问题。如果没有自定义拷贝构造函数，会调用默认拷贝构造函数，这样就会调用两次析构函数。**第一次析构函数delete了内存，第二次的就指针悬挂了。所以，此时，必须采用深拷贝。**

深拷贝与浅拷贝的区别就在于深拷贝会在堆内存中另外申请空间来储存数据，从而也就解决了指

针悬挂的问题。简而言之，当数据成员中有指针时，必须要用深拷贝。

O0、O1、O2、O3优化

-O0

不做任何优化，这是默认的编译选项。

-O1

主要对代码的分支、常量以及表达式进行优化。会减小代码的尺寸，缩短执行周期啥的

-floop-optimize：执行循环优化,将常量表达式从循环中移除，简化判断循环的条件，并且 optionally do strength-reduction，或者将循环打开等。在大型复杂的循环中，这种优化比较显著。

-O2

O2优化再打开O1优化的前提下，尝试更多的寄存器级的优化以及指令级的优化，它会在编译期间占用更多的内存和编译时间。

所以一般来说可以直接开-O2的优化等级，

-O3

在O2的基础上进行更多的优化，例如使用伪寄存器网络，普通函数的内联，以及针对循环的更多优化。

这个好像不推荐，因为会增加编译失败和程序不可预知的一些行为，不建议使用

c++中四种变量存储类型总结

在C++语言中，变量的存储类共有如下四种：

- (1) auto存储类（自动存储类）
- (2) static存储类（静态存储类）
- (3) extern存储类（外部存储类）

(4) register存储类（寄存器存储类）

- **自动存储类**

auto存储类，即自动存储类。在函数内部定义的变量，如果不指定其存储类，那么它就是auto类变量。这个是最常见的，所以我们不加关键字auto

这是我们经常见到的一种变量存储类型。见如下代码：

```
void func( ) { int a; auto int b; ... }  
//a和b都是auto存储类变量
```

自动存储类在在进入代码块（函数）之前生成，在函数体内部存活，出了函数体（函数返回）后就消失。

自动变量默认初始值是不确定的、

自动存储类每调用一次函数时都要赋一次初始值

- **静态存储类**

static关键字在c和c++中是不同的，这个在上面说过了，具体的话可以去看上面。

- **extern存储类**

如果在一个文件中要引用另一个文件中定义的外部变量，则在此文件中应用extern关键字把此变量说明为外部的。例如：

```
extern int a; //a为别的文件中定义的外部变量  
int mydata; //外部变量的定义  
extern int mydata; //外部变量的说明
```

大型程序为了易于维护和理解，通常需要把程序划分为多个文件来保存，每个文件都可以单独编译，最后再把多个文件的编译结果（即目标文件）连接到一起来生成一个可执行程序。这种情况下，如果在一个文件中需要引用另一个文件中的外部变量，就需要利用extern说明。

- **register存储类**

为了提高某些自动类变量或函数参数的处理速度，可以在定义这些变量的类型说明符的前面加上register关键字，以通知编译系统为这些变量分配寄存器来存放其值。若使用register（而非auto）存储类标识代码块内的变量，编译器就会将变量缓存于处理器内的寄存器中，此种情况下不能对该变量或其成员变量使用引用操作符&以获取其地址，因为&只能获取内存空间中的地址

C++中this指针相关问题

This指针的来源

通过转化成c语言好理解一点。早期还没有针对特定c++的编译器，因此编译c++的时候都先翻译成c语言，再进行编译。

对于class结构来说，c语言中与之对应的就是结构体。

类中的成员变量可以翻译成结构体域中的变量，但是结构体中没有成员函数这个概念，因此成员函数就翻译成为全局函数。

那么如果函数内部想使用成员数据，可以使用该对象指针的方式，指向成员变量。

其作用就是指向非静态成员函数所作用的对象。

1. `this` 指针是一个隐含于每一个非静态成员函数中的特殊指针。它指向**调用该非静态成员函数的那个对象**。
2. 当对一个对象调用成员函数时，编译程序先将对象的地址赋给 `this` 指针，然后调用成员函数，每次成员函数存取数据成员时，都隐式使用 `this` 指针。
3. `this` 并不是一个常规变量，而是个右值，所以不能取得 `this` 的地址（不能 `&this`）。

this指针是什么时候创建的？

this在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果class里面没有方法的话，它们是没有构造函数的，只能当做C的struct使用。采用 `TYPE xx` 的方式定义的话，在栈里分配内存，这时候this指针的值就是这块内存的地址。采用 `new` 的方式创建对象的话，在堆里分配内存，`new`操作符通过`eax`（累加寄存器）返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给`ecx`

this指针存放在何处？堆、栈、全局变量，还是其他？

this指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级别里面，一个值只会以3种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的。

如果我们知道一个对象this指针的位置，可以直接使用吗？

this指针只有在成员函数中才有定义。

因此，你获得一个对象后，也不能通过对象使用this指针。所以，我们无法知道一个对象的this指针的位置（只有在成员函数里才有this指针的位置）。当然，在成员函数里，你是可以知道this指针的位置的（可以通过&this获得），也可以直接使用它。

在成员函数中调用delete this会出现什么问题？

在类对象的内存空间中，只有数据成员和虚函数表指针，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个this指针，让成员函数知道当前是哪个对象在调用它。

当调用delete this时，类对象的内存空间被释放。在delete this之后进行的其他任何函数调用，只要不涉及到this指针的内容，都能够正常运行。一旦涉及到this指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

如果在类的析构函数中调用delete this，会发生什么？

会导致堆栈溢出。原因很简单，delete的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存”。显然，delete this会去调用本对象的析构函数，而析构函数中又调用delete this，形成无限递归，造成堆栈溢出，系统崩溃。

inline 内联函数与宏定义

首先分析一下C宏定义的好处：首先C语言是一个效率很高的语言，使用预处理器实现，没有参数压栈，函数返回等操作，效率很高。

1. 相当于把内联函数里面的代码写在调用内联函数处。不用执行进入函数的步骤，直接执行函数体。
2. 从上面那一条的角度说，内联函数更像是宏，但却比宏多了类型检查，真正具有函数特性。
3. 在类声明中定义的函数，除了虚函数的其他函数都会自动隐式地当成内联函数。
4. 编译器会为所用 inline 函数中的局部变量分配内存空间
5. 会将 inline 函数的的输入参数和返回值映射到调用方法的局部变量空间中；

- 优点

- 内联函数同宏函数一样将在被调用处进行代码展开，省去了参数压栈、栈帧开辟与回收，结果返回等，从而提高程序运行速度。
- 内联函数相比宏函数来说，在代码展开时，会做安全检查或自动类型转换（同普通函数），而宏定义则不会。
- 内联函数在运行时可调试，而宏定义不可以。
- 可以说inline函数不仅吸收了C宏定义的，同时消除宏定义的缺点。
- 缺点
 - 代码膨胀。内联是以代码膨胀（复制）为代价，消除函数调用带来的开销。
 - inline 函数无法随着函数库升级而升级。inline函数的改变需要重新编译。
 - 万一又递归调用，代码量很大。
- 虚函数（virtual）可以是内联函数（inline）吗？
- 首先要明白，内联函数是编译器做出的选择，是否内联决定权在编译器，程序员不可控。同时，虚函数是多态性的一种体现，多态性表现在函数的运行阶段而不是函数的编译阶段。因此，**虚函数表现为多态性时（运行期）不可以内联。**
- 唯一可以内联的时候是：编译器知道所调用的对象是哪个类。只有在**编译器具有实际对象而不是对象的指针或引用时才会发生。**

inline说明对编译器来说只是一种建议，编译器可以忽略这个建议的，比如，你将一个长达100多行的函数指定为inline，编译器就会自动忽略这个inline，将这个函数还原成普通函数。在调用内联函数时，要保证内联函数的定义让编译器看到，也就是说，内联函数inline必须要定义在头文件中，这与通常的函数定义是不一样的。

struct 和 typedef struct

```
//代码1 (c语言)
typedef struct test3{
    int a;
    int b;
    int c;
}test4;
//代码2 (c++)
struct test3{
    int a;
    int b;
    int c;
}test4;
```

- 在c语言中

对于代码1。**test3**相当于标识符，而**test4**相当于变量类型.定义一个结构体变量为 `test4 t`，即 `struct test3 = test4`

对于代码2。要定义结构体变量必须写成 `struct test3 t`

- 在c++中

对于代码1。不要这样写，基本上是代码二的类型。`test4`变成了变量名

对于代码2。这是c++的写法，`test4`相当于一个结构体变量。

explicit 关键字

`explicit`关键字主要是用来修饰类中的构造函数的，对于仅有一个参数或除第一个参数外其余参数均有默认值的类构造函数，尽可能使用`explicit`关键字修饰。因为只有一个参数或者出了第一个参数其他参数是默认参数的构造函数来说，他还有另一个名字叫做转换构造函数。

****所以explicit主要用来防止隐式转换。****因为仅含一个参数的构造函数和除了第一个参数外其余参数都有默认值的多参构造函数承担了两个角色。第一个是成为带参数的构造函数，第二个是一个默认且隐含的类型转换操作符（就是单参数的构造函数是一种隐含的类型转换符）

额外说一下隐式类型转换：

c++隐式类型转换是指c++自动将一种类型转换成另一种类型，是编译器的一种自主行为。

举一些类型转换的例子：

```
int i=3;
double j = 3.1;
i+j;//i会被转换成double类型，然后才做加法运算。

class A{};
class B: public A
{};//B是子类
void Fun(A& a);
B b;
Fun(b);//使用子类对象代替父类对象是可以的，也是因为隐式类型转换。

class Test
{
    public:
        Test(int i);
};

Test t1 = 1;//正确，由于强制类型转换，1先被Test构造函数构造造成Test对象，然后才被赋值给t1
Test t2(1);//正确
```

friend友元类和友元函数

[看这里](#)

采用类的机制后实现了数据的隐藏与封装，类的数据成员一般定义为私有成员，成员函数一般定义为公有的，依此提供类与外界间的通信接口。有时需要定义一些函数，这些函数不是类的一部分，但又需要频繁地访问类的数据成员，这时可以将这些函数定义为该函数的友元函数。如果不用友元你可以使用成员函数来set这些数据成员。

除了友元函数外，还有友元类，两者统称为友元。

友元函数能够使得普通函数直接访问类的保护数据和私有数据成员，避免了类成员函数的频繁调用，可以节约处理器开销，提高程序的效率，但所矛盾的是，即使是最大限度大保护，同样也破坏了类的封装特性，这即是友元的缺点，在现在cpu速度越来越快的今天我们并不推荐使用它。

友元类友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息（包

括私有成员和保护成员）。

友元的作用是提高了程序的运行效率（即减少了类型检查 and 安全性检查等都需要时间开销），但它破坏了类的封装性和隐藏性，使得非成员函数可以访问类的私有成员。

我碰到了一种必须使用友元的情况。如下图



```
C++  
#include <bits/stdc++.h>  
using namespace std;  
struct fruit{  
    string name;  
    int price;  
    //友元+重载  
    friend bool operator < (fruit f1, fruit f2){  
        return f1.price < f2.price; //把价格高的排前面  
    }  
};  
priority_queue<fruit> q;  
int main(){  
    fruit f1, f2, f3;  
    f1.name = "桃子";  
    f1.price = 3;  
    f2.name = "梨";  
    f2.price = 4;  
    f3.name = "苹果";  
    f3.price = 1;  
    q.push(f1);  
    q.push(f2);  
    q.push(f3);  
    cout << q.top().name << " " << q.top().price << endl;  
    return 0;  
}
```

上述代码由于使用了priority_queue，而且每个成员是一个结构体，所以需要重新定义std::less的比较方式，所以在结构体重重载了<号，这个时候重载运算符需要用到友元。因为std::less类中的比较运算符需要调用struct中的，因此需要把结构体中的设置为友元。

C++ 虚函数

前瞻

虚函数是实现多态的一个技术之一。在C++中，动态联编通过指针和引用来实现。但是，想一想，正常来说我们不允许程序将一种类型的指针指向另外一种类型的地址。但是在类中却可以这样做，派生类的指针指向基类对象的地址（派生类对象的地址赋给基类指针），而且不用进行显式转换，如下：

```
BrassPlus dilly;  
Brass *pb = &dilly;  
Brass &pb = dilly;
```

派生类的指针指向基类对象的地址（派生类对象的地址赋给基类指针）称为向上转型，C++允许隐式向上转型。将子类指向父类，向下转换则必须强制类型转换。

然后我们就可以用父类的指针指向其子类的对象，然后通过父类的指针调用实际子类的成员函数。如果子类重写了该成员函数就会调用子类的成员函数，没有声明重写就调用基类的成员函数。这种技术可以让父类的指针有“多种形态”。

用到的情形举例：比如你有个游戏，游戏里有个虚基类叫「怪物」，有纯虚函数「攻击」。然后派生出了三个子类「狼」「蜘蛛」「蟒蛇」，都实现了自己不同的「攻击」函数，比如狼是咬人，蜘蛛是吐丝，蟒蛇把你缠起来～～。然后出现好多怪物的时候就可以定义一个虚基类指针数组，把各种怪物的指针给它，然后迭代循环的时候直接 `monster[i]->attack()` 攻击玩家就行了，如下图：

虚函数工作原理

C++没有强制规定虚函数的实现方式。编译器中主要用虚表指针（vptr）和虚函数表（vtbl）来实现的

先直接上图，然后再看一下虚函数的执行过程：

```

int main(int argc, char const *argv[])
{
    Monster *pMonster[3];
    pMonster[0] = new Wolf;
    pMonster[1] = new Spider;
    pMonster[2] = new Snake;

    for (int i = 0; i < 3; ++i)
    {
        pMonster[i]->attack();
    }

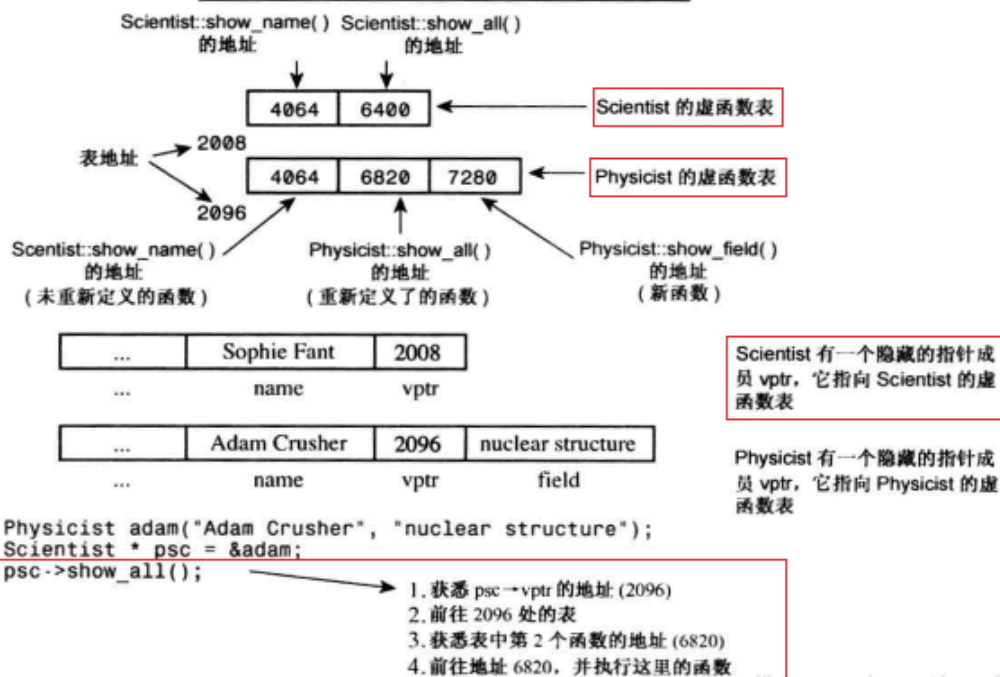
    return 0;
}

```

```

class Scientist{
{
    ...
    char name[40];
public:
    virtual void show_name();
    virtual void show_all();
    ...
};
class Physicist : public Scientist
{
    ...
    char field[40];
public:
    void show_all(); // redefined
    virtual void show_field(); // new
    ...
};

```



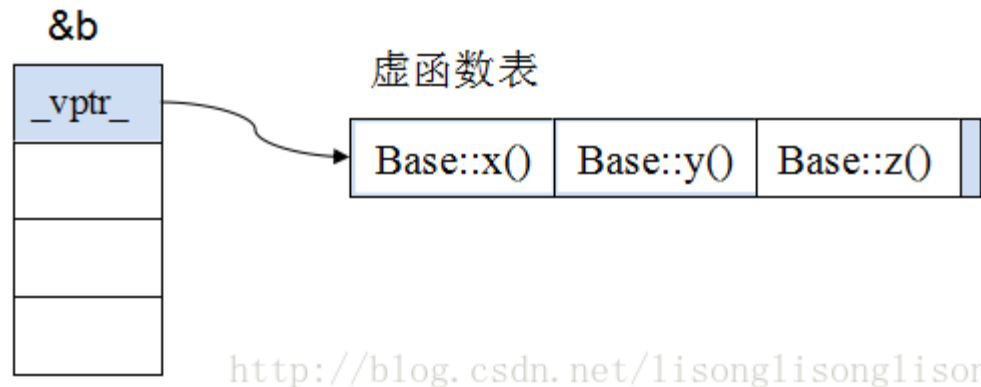
一种虚函数机制

当调用一个对象对应的函数时，通过对象内存中的vptr找到一个虚函数表（注意这虚函数表既不在堆上，也不再栈上）。虚函数表内部是一个函数指针数组，记录的是该类各个虚函数的首地

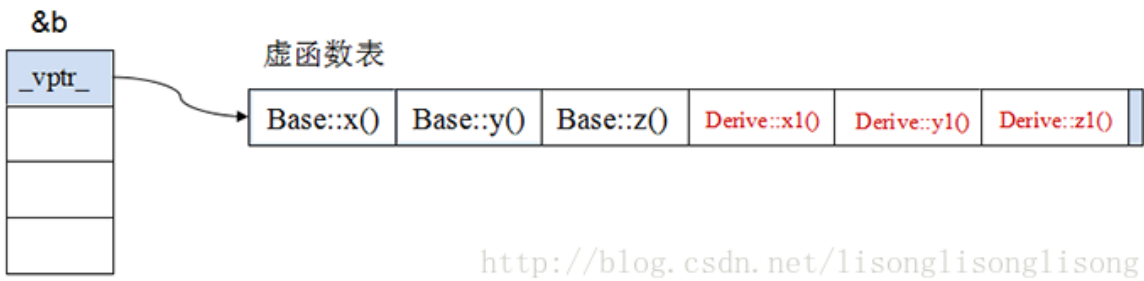
址。然后调用对象所拥有的函数。

继承情况下的虚函数表

- 原始基类的虚函数表



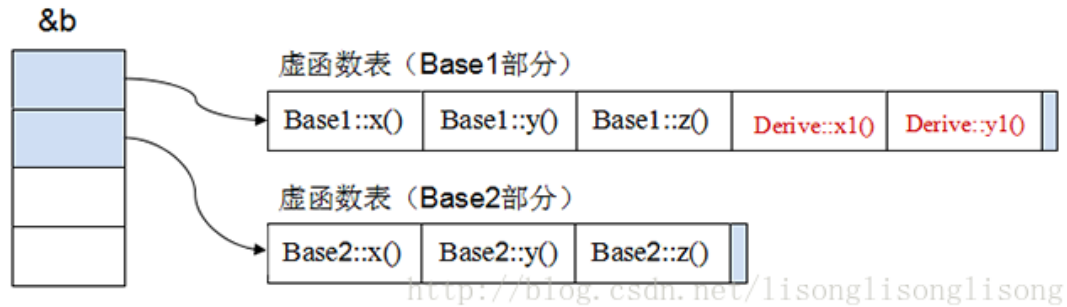
- 单继承时的虚函数（无重写基类虚函数）



- 单继承时的虚函数（重写基类虚函数）



- 多重继承时的虚函数（Derived::public Base1,public Base2）



这样就会有内存和执行速度方面的成本：

1. 每个对象都会增大，因为在对象最前面的位置加入了指针

2. 对于每个类，编译器都会创建虚函数地址表
3. 对于每个函数调用，都要查找表中地址

虚函数的性能分析

第一步是通过对象的vptr找到该类的vtbl，因为虚函数表指针是编译器加上去的，通过vptr找到vtbl就是指针的寻址而已。

第二部就是找到对应vtbl中虚函数的指针，因为vtbl大部分是指针数组的形式实现的

在单继承的情况下调用虚函数所需的代价基本上和非虚函数效率一样，在大多数计算机上它多执行了很少的一些指令

在多继承的情况由于会根据多个父类生成多个vptr，在对象里为寻找 vptr 而进行的偏移量计算会变得复杂一些

空间层面为了实现运行时多态机制，编译器会给每一个包含虚函数或继承了虚函数的类自动建立一个虚函数表，所以虚函数的一个代价就是会增加类的体积。在虚函数接口较少的类中这个代价并不明显，虚函数表vtbl的体积相当于几个函数指针的体积，如果你有大量的类或者在每个类中有大量的虚函数,你会发现 vtbl 会占用大量的地址空间。但这并不是最主要的代价，主要的代价是发生在类的继承过程中，在上面的分析中，可以看到，当子类继承父类的虚函数时，子类会有自己的vtbl，如果子类只覆盖父类的一两个虚函数接口，子类vtbl的其余部分内容会与父类重复。如果存在大量的子类继承，且重写父类的虚函数接口只占总数的一小部分的情况下，会造成**大量地址空间浪费**。同时由于虚函数指针vptr的存在，虚函数也会增加该类的每个对象的体积。在单继承或没有继承的情况下，类的每个对象只会多一个vptr指针的体积，也就是4个字节；在多继承的情况下，类的每个对象会多N个（N=包含虚函数的父类个数）vptr的体积，也就是4N个字节。当一个类的对象体积较大时，这个代价不是很明显，但当一个类的对象很轻量时，如成员变量只有4个字节，那么再加上4（或4N）个字节的vptr，对象的体积相当于翻了1（或N）倍，这个代价是非常大的。

虚函数的一些问题

- 构造函数可以设置为虚的吗？

答：不能。因为虚函数的调用是需要通过“虚函数表”来进行的，而虚函数表也需要在对象实例化之后才能够进行调用。在构造对象的过程中，还没有为“虚函数表”分配内存。所以，这个调用也是违背先实例化后调用的准则。

子类的默认构造函数总要执行的操作：执行基类的代码后调用父类的构造函数。

- **c++虚析构函数**

如果类是父类，则必须声明为虚析构函数。基类声明一个虚析构函数，为了确保释放派生对象时，按照正确的顺序调用析构函数。

如果析构函数不是虚的，那么编译器只会调用对应指针类型的虚析构函数。切记，是指针类型的，不是指针指向类型的！而其他类的析构函数就不会被调用。例如如下代码：

```
Employee* pe = new Singer;
delete pe;
```

只会调用Employee的析构函数而不会调用Singer类的析构函数。如果这个类不是父类也可以定义虚析构函数，只是效率方面问题罢了

- **那些函数不能是虚函数？**

除了上面说的构造和析构函数往外。

①**友元函数**不是虚函数，因为友元函数不是类成员，只有类成员才能使虚函数。

②**静态成员函数**不能是虚。在C++中，静态成员函数不能被声明为virtual函数。首先会编译失败，也就是不能通过编译。

原因如下：

- i. static成员不属于任何类对象或类实例，所以即使给此函数加上virtual也是没有任何意义的。
- ii. 静态与非静态成员函数之间有一个主要的区别。那就是静态成员函数没有隐藏的this指针。对于虚函数，它的调用恰恰需要this指针。在有虚函数的类实例中，this指针调用vptr指针，vptr找到vtable(虚函数列表)，通过虚函数列表找到需要调用的虚函数的地址。总体来说虚函数的调用关系是：this指针->vptr->vtable->virtual虚函数。所以说，static静态函数没有this指针，也就无法找到虚函数了

③**内联函数**也不能是虚的，因为要在编译的时候展开，而虚函数要求动态绑定。另外就是虚函数的类对象必须包含vptr，但是内联函数是没有地址的，编译的时候直接展开了所以不行。

④**构造函数**也不行。

⑤**成员函数模板不能是虚函数**。因为c++编译器在解析一个类的时候就要确定虚函数表的大小，如果允许一个虚函数是模板函数，那么compiler就需要在parse这个类之前扫描所有的代码，找出这个模板成员函数的调用（实例化），然后才能确定vtable的大小，而显然这是不可行的，除非改变当前compiler的工作机制。因为类模板中的成员函数在调用的时候才会创建

- **虚函数表是共享还是独有的？**

答：虚函数表是针对类的，一个类的所有对象的虚函数表都一样。在gcc编译器的实现中虚函数表vtable存放在可执行文件的只读数据段.rodata中。是编译器在编译器为我们处理好的。

- **虚函数表和虚函数指针的位置？**

答：既不在堆上，也不在栈上。虚函数表（vtable）的表项在编译期已经确定，也就是一组常量函数指针。跟代码一样，在程序编译好的时候就保存在**可执行文件里面**。程序运行前直接加载到内存中即可。而堆和栈都是在运行时分配的。而跟虚函数表对应的，是虚函数表指针（vptr），作为对象的一个（隐藏的）成员，总是跟对象的其他成员一起。如果对象分配在堆上，vptr也跟着在堆上；如果对象分配在栈上，vptr也在栈上……

- **编译器如何处理虚函数表**

对于派生类来说，编译器简历虚表的过程有三部：

- i. 拷贝基类的虚函数表，如果是多继承，就拷贝每个基类的虚函数表
- ii. 查看派生类中是否有重写基类的虚函数，如果有，就替换成已经重写后的虚函数地址
- iii. 查看派生类中是否有新添加的虚函数，如果有，就加入到自身的虚函数表中

- **构造函数或析构函数中调用虚函数会怎样？**

首先不应该在构造函数和析构函数中调用虚函数。

在构造函数中调用虚函数。假如有一个动物基类，这个基类定义了一个虚函数来表示动物的行为，叫做action。我们在基类的构造函数中调用这个虚函数。然后有一个派生类重写了该虚函数。当我们创建一个派生类对象的时候，首先会执行基类部分，因此执行基类的构造函数，然后才会执行子类的构造函数。编译器在执行基类构造函数中的虚函数时，会认为这是一个基类的对象，因为派生类还没有构造出来。因此达不到动态绑定的效果，父类构造函数中调用的仍然是父类版本的函数，子类中调用的仍然是子类版本的函数

在析构函数中调用虚函数。析构函数也是一样，派生类先进行析构，如果有父类的析构函数中有virtual函数的话，派生类的内容已经被析构了，C++会视其基类，执行基类的virtual函数。

- **如何获取 虚表地址和虚函数地址？**


```

//该类如下:
class Base {
public:
    virtual void f() { cout << "Base::f" << endl; }
    virtual void g() { cout << "Base::g" << endl; }
    void h() { cout << "Base::h" << endl; }
};

Base b;
// 1.&b代表对象b的起始地址
// 2.(int *)&b 强转成int *类型,为了后面取b对象的前4个字节,前四个字节是虚表指针
// 3.*(int *)&b 取前四个字节,即vptr虚表地址
printf("虚表地址:%p\n", *(int *)&b);
// 根据上面的解析我们知道*(int *)&b是vptr,即虚表指针.并且虚表是存放虚函数指针的
// 所以虚表中每个元素(虚函数指针)在32位编译器下是4个字节,因此(int *)*(int *)&b
// 这样强转后为了后面的取四个字节.所以*(int *)*(int *)&b就是虚表的第一个元素.
printf("第一个虚函数地址:%p\n", *(int *)*(int *)&b);
printf("第二个虚函数地址:%p\n", *((int *)*(int *)&b + 1));
//始终记着vptr指向的是一块内存,
// 这块内存存放着虚函数地址,这块内存就是我们所说的虚表.
//64位下把int换成longlong就好了

```

C语言怎么实现多态

c++的多态分为两种：

1. 编译时多态：重载
2. 运行时多态：重写即虚函数。虚函数本身其实就是回调函数

思路就是使用函数指针来实现多态。

编译时多态：

先说一个c中的宏，`__V_ARGS__`，是c99引入进来的可变参宏，一般是用来输出debug信息。可以用这个宏实现一个简单的多态机制。代码举例：

```

#define Check(...) printf(__VA_ARGS__);
int main(int argc, char *argv[])
{
    int i = 1, j = 2;
    char *Error = "error!!";
    Check("i = %d\n", j);
    Check("i = %d, j = %d\n", i, j);
    Check("i = %d, j = %d, Error:%s\n", i, j, Error);
    return 0;
}

/*****关于结构体的*****/
#define func(...) myfunc((struct mystru) { __VA_ARGS__})

struct mystru
{
    const char *name;
    double d;
    int number;
};

void myfunc(struct mystru ms)
{
    printf("%s, %lf, %d\n", ms.name, ms.d, ms.number);
}

int main(int argc, char *argv[])
{
    func();
    func("hello", 1.1);
    func("hello");
    func("hello", 1.1, 100);
    func(.name = "hello");
    func(.d = 1.1);
    func(.number = 100);
    func(.d = 1.1, .name = "hello");
    return 0;
}

```

注意：一般我们在使用结构体的时候都是先赋值在传参数，这里实际上是用"..."可变参数替换为

了 `__VA_ARGS__` 这个宏，然后转化为 `(struct mystru){ __VA_ARGS__ }`，其实就是 `(struct mystru){ ... }`。通过 `(struct mystru)` 把参数转化结构体，这也是为什么如果我们不指定 `.name` 或者是 `.number` 时，必须按顺序传递参数，否则报错，因为在内存中结构体布局是确定的。不明确指定哪个参数只能按顺序传递。

运行时多态：

在c中 `sizeof(struct)` 是0,从内存上来看，c++的class和struct不仅仅有数据还有成员函数，这些成员函数如果是non-inline的，那么只会在内存中产生一份实例供所有对象使用，如果是inline的，会为每一个使用着产生一个实例。而c的struct就简单多了，它不占用内存空间，每一个实例按照struct分配一份就好，但这也是问题所在，它在内存中没实例啊。那么用c的struct实现时，我们就要想办法让它在内存中存在一份，大家都能找到它。

所以主要关键点就是在于函数指针的使用

```

//虚函数表结构
struct base_vtbl
{
    void(*dance)(void *);
    void(*jump)(void *);
};

//基类
struct base
{
    /*virtual table*/
    struct base_vtbl *vptr;
};

//基类的构造函数
struct base * new_base()
{
    struct base *temp = (struct base *)malloc(sizeof(struct base));
    //基类虚表结构中函数指针具体关联的函数名
    temp->vptr->dance = base_dance;
    temp->vptr->jump = base_jump;
    return temp;
}

//基类的成员函数
void base_dance()
{
    printf("base dance\n");
}

void base_jump()
{
    printf("base jump\n");
}

//派生类
struct derived1
{
    struct base super;
    int high;
};

```

```

//派生类的构造函数
struct derived1 * new_derived1(int h)
{
    struct derived1 * temp= (struct derived1 *)malloc(sizeof(struct derived1));
    //派生类虚表结构中函数指针具体关联的函数名
    temp->super->vptr->dance = derived1_table;
    temp->super->vptr->jump = derived1_jump;
    temp->high = h;
    return temp;
}
//派生类对象成员函数
void derived1_dance()
{
    printf("derived1 dance\n");
}
void derived1_jump(void * this)
{
    struct derived1* temp = (struct derived1 *)this;
    printf("derived1 jump:%d\n", temp->high);
}

/*****实际调用*****/
struct base * bas = new_base();
//这里调用的是基类的成员函数
bas->vptr->dance();
bas->vptr->jump();

struct derived1 * child = new_derived1(100);
//基类指针指向派生类
bas = (struct base *)child;

//这里调用的其实是派生类的成员函数
bas->vptr->dance();
bas->vptr->jump((void *)bas);

```

抽象基类-纯虚函数

定义纯虚函数是为了实现一个接口，起到一个规范的作用，规范继承这个类的程序员必须实现这

个函数。

有纯虚函数的类叫做抽象基类，对于抽象类来说，C++是不允许它去实例化对象的。也就是说，抽象类无法实例化对象

C++ 纯虚基类函数=0

“=0”这个操作在虚函数中有2层意思，即告诉编译器：

1. 有的朋友误解这是返回值为0的意思，但是它并不是，它仅表示的是这个是个纯虚函数，是个抽象函数，没有实现
2. 这个类的继承类里面必须要实现这个函数。

如何定义一个只能在堆上（栈上）生成对象的类？

综述：在c++中，建立对象有两种方式，分为静态建立和动态建立，如代码所示：

```
//静态建立
A a;
//动态建立
A *a = new A;
```

静态建立对象时，由编译器自动为对象在栈中分配内存，是直接移动栈顶指针，腾出适当的空间，然后再这片空间上调用构造函数生成对象，这种静态建立是直接调用类的构造函数。

动态建立对象时，程序员使用new 运算符在堆中建立。这个过程分为两步，第一部是调用operator new()函数在堆空间中搜索出来适当的内存并进行分配，第二步是调用构造函数生成对象，初始化这片内存空间，间接调用构造函数。

- 只能在堆上生成对象的类。

只能在堆上也就意味着不能再栈上，在栈上是编译器分配内存空间，构造函数来构造栈对象。在栈上当对象周期完成，编译器会调用析构函数来释放栈对象所占的空间，也就是说编译器管理了对象的整个生命周期。**编译器在调用构造函数为类的对象分配空间时，会先检查析构函数的访问性**，不光是析构函数，编译器会检查所有非静态函数的访问性。因此，如果类的析构函数是私有的，编译器不会为对象在栈上分配内存空间。

扩展一下，如果把析构函数写在private中的话，不能用A a这种静态方式建立对象。但也会有其他问题，如果这个类是父类的话，通常要将析构函数加上virtual关键字，然后再子类重写，实现多态性。但是子类不能访问private成员，可以使用protected，子类可以访问父类的protected成员，但不能访问private。另一个问题，当用new建立后，无法delete。因为delete会调用对象的析构函数，但是析构函数在类外无法访问。可以这样写：

```
class A
{
protected :
    A(){}
    ~A(){}
public :
    static A* create()
    {
        return new A();
    }
    void destory()
    {
        delete this ;
    }
};
```

- 只能在栈上生成对象的类。

只有使用new运算符才会在堆上创建对象。设为私有即可。

动态建立类对象，是使用new运算符将对象建立在堆空间中。这个过程分为两步，第一步是执行operator new()函数，在堆空间中搜索合适的内存并进行分配；第二步是调用构造函数构造对象，初始化这片内存空间（这种方法，间接调用类的构造函数），但是operator new()函数用于分配内存，无法提供构造功能，**所以不能将构造函数设为私有**；

```
class A
{
private :
    void * operator new ( size_t t){}      // 注意函数的第一个参数和返回值都是固定的
    void operator delete ( void * ptr){}  // 重载了new就需要重载delete
public :
    A(){}
    ~A(){}
};
```

C++如何阻止类被实例化？

为什么要组织实例化？

一个类不想被实例化通常有两种情况：一种是抽象类，一种是工具类。

抽象类

比如现在需要计算图形的面积，可以是正方形、长方形、圆形等等。于是抽象出了基类，叫图形。这部分抽象的没办法实例化，如：

```
class Sharp{};

class Circle : public Sharp{};

class Rectangle : public Sharp{};
```

工具类

我们需要一个类来封装加、减、乘、除。这个类就是一个典型的工具类，用它创建对象没有意义，可以直接通过类名调用静态成员函数。


```
class Calculate
{
public:
    static int add(int x, int y);
    static int sub(int x, int y);
    static int mul(int x, int y);
    static int div(int x, int y);
};
```

如何阻止？

方法一：类中包含纯虚函数。

```
class Sharp
{
public:
    virtual void get_s() = 0;    //纯虚函数
};
```

纯虚函数没有函数体。含有纯虚函数的类叫抽象类。抽象类不好创建对象，因为就算是创建了对对象，调用纯虚函数的时候，也不知道如何执行。

方法二：构造函数私有

把构造函数设置成私有，就不能在类的外部创建对象，相当于间接的阻止了该类实例化对象。

```
class Calculate
{
private:
    Calculate();
public:
    static int add(int x, int y);
    static int sub(int x, int y);
    static int mul(int x, int y);
    static int div(int x, int y);
};
```

C++所有构造函数

类对象被创建时，编译器为对象分配内存空间，并自动调用构造函数，由构造函数完成成员的初始化工作。

因此构造函数的作用是初始化对象的成员函数。

****默认构造函数：****如果没有人为构造函数，则编译器会自动默认生成一个无参构造函数。

****一般构造函数：****包含各种参数，一个类可以有多个一般构造函数，前提是参数的个数和类型和传入参数的顺序都不相同，根据传入参数调用对应的构造函数。

****拷贝构造函数：****拷贝构造函数的函数参数为对象本身的引用，用于根据一个已存在的对象复制出一个新的该类的对象，一般在函数中会将已存在的对象的数据成员的值一一复制到新创建的对象中。如果没有显示的写拷贝构造函数，则系统会默认创建一个拷贝构造函数，但当类中有指针成员时，最好不要使用编译器提供的默认的拷贝构造函数，最好自己定义并且在函数中执行深拷贝。

****移动构造函数：****有时候我们会遇到这样一种情况，我们用对象a初始化对象b后对象a我们就不在使用了，但是对象a的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把a对象的内容复制一份到b中，那么为什么我们不能直接使用a的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷。拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制。

但是指针的浅层复制是非常危险的。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若第一个指针将其释放，另一个指针的指向就不合法了（pointer dangling）。所以我们只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如a->value）置为NULL，这样在调用析构函数的时候，由于有判断是否为NULL的语句，所以析构a的时候并不会回收a->value指向的空间（同时也是b->value指向的空间）

赋值构造函数：=运算符的重载，类似拷贝构造函数，将=右边的类对象赋值给类对象左边的对象，不属于构造函数，=两边的对象必须都要被创建。

****类型转换构造函数：****有时候不想要隐式转换，用explicit关键字修饰。一般来说带一个参数的构造函数，或者其他参数是默认的构造函数

什么情况下会调用拷贝构造函数？

1. 对象以值传递的方式进入函数体
2. 对象以值传递的方式从函数返回
3. 一个对象需要另外一个对象初始化

为什么拷贝构造函数必须是引用？

为了防止递归调用。

如果不用引用，就会是值传递的方式，但是值传递会调用拷贝构造函数生成临时对象，从而又调用一次拷贝构造函数。就这样无穷的递归下去。

如果是指针类型

就变成了一个带参数的构造函数了。。。

比如 `A(A* test)`

构造函数析构函数是否能抛出异常

构造函数可以抛出异常

对象只有在构造函数执行完成之后才算构造妥当，c++只会析构已经完成的对象。因此如果构造函数中发生异常，控制权就需要转移出构造函数，执行异常处理函数。在这个过程中系统会认为对象没有构造成功，导致不会调用析构函数。在构造函数中抛出异常会导致当前函数执行流程终止，在构造函数流程前构造的成员对象会被释放，但是如果在构造函数中申请了内存操作，则会造成内存泄漏。另外，如果有继承关系，派生类中的构造函数抛出异常，那么基类的构造函数和析构函数可以照常执行的。

解决办法：用智能指针来管理内存就可以

C++标准指明析构函数不能、也不应该抛出异常

C++异常处理模型最大的特点和优势就是对C++中的面向对象提供了最强大的无缝支持。那么如果对象在运行期间出现了异常，C++异常处理模型有责任清除那些由于出现异常所导致的已经失效了的对象(也即对象超出了它原来的作用域)，并释放对象原来所分配的资源，这就是调用这些

对象的析构函数来完成释放资源的任务，所以从这个意义上说，析构函数已经变成了异常处理的一部分。

析构函数不能抛出异常原因有两个：

1. 如果析构函数抛出异常，则异常点之后的程序不会执行，如果析构函数在异常点之后执行了某些必要的动作比如释放某些资源，则这些动作不会执行，会造成诸如资源泄漏的问题。
2. 异常发生时，c++的异常处理机制在异常的传播过程中会进行栈展开（stack-unwinding）。在栈展开的过程中就会调用已经在栈构造好的对象的析构函数来释放资源，此时若其他析构函数本身也抛出异常，则前一个异常尚未处理，又有新的异常，会造成程序崩溃。

解决办法：把异常完全封装在析构函数内部，决不让异常抛出函数之外，代码如下：

```
DBConn::~~DBConn()  
{  
    try  
    {  
        db.close();  
    }  
    catch(...)  
    {  
        abort();  
    }  
}  
//如果close抛出异常就结束程序，通常调用abort完成：
```

创建派生类对象，构造函数的执行顺序是什么？析构函数的执行顺序？

首先要知道类的构造函数不能被继承，构造函数不能被继承是有道理的，因为即使继承了，它的名字和派生类的名字也不一样（构造函数名字和类名一样），不能成为派生类的构造函数，当然更不能成为普通的成员函数。在设计派生类时，对继承过来的成员变量的初始化工作也要由派生类的构造函数完成，但是大部分基类都有private属性的成员变量，它们在派生类中无法访问，更不能使用派生类的构造函数来初始化。这种矛盾在C++继承中是普遍存在的，这个问题的

思路是：在派生类的构造函数中调用基类的构造函数。

对象创建时候执行顺序是：**静态代码 --> 非静态代码 --> 构造函数。**静态代码包括（静态方法，静态变量，静态代码块等），非静态代码即（成员方法，成员变量，成员代码块等）。代码块中或者成员变量中如果有类对象的话，肯定要先将类对象创建出来。所以说先执行代码块再执行构造函数，而且如果代码块中有多个成员变量，则按照成员变量的声明顺序进行构造。初始化列表的顺序，不影响成员变量构造顺序

因为静态变量等这些在内存中属于全局变量，所以要先创建。然后类会查看成员函数的相关信息，为该类的对象中的每个成员变量分配相应的存储空间，然后再执行构造函数创建对象，如果构造函数有初始化，则将值放到准备好的内存空间中。

构造函数执行顺序

1. 基类构造函数。如果有多个基类，则构造函数的调用顺序是该基类在派生类中出现的顺序，而不是他们在成员初始化列表中的顺序。
2. 成员类对象构造函数。如果有多个成员类构造函数调用顺序是对象在类中被声明的顺序，而不是他们在成员初始化列表中的顺序
3. 派生类构造函数

析构函数顺序

1. 派生类析构函数
2. 成员类对象的析构函数
3. 调用基类的析构函数

C++成员变量的初始化顺序问题

```
class A
{
private:
    int n1;
    int n2;
public:
    A():n2(0),n1(n2+2){}
    void Print(){
        cout << "n1:" << n1 << ", n2: " << n2 <<endl;
    }
};

int main()
{
    A a;
    a.Print();
    return 1;
}

//输出: n1: 是一个随机数; n2: 0
//有的编译器是n1:2, n2=0
```

1. 成员变量在使用初始化列表初始化时，只与定义成员变量的顺序有关，与构造函数中初始化成员列表的顺序无关。因为成员变量的初始化次序是根据变量在内存中次序有关，而内存中的排列顺序早在编译期就根据变量的定义次序决定了。
2. 如果不使用初始化列表初始化，在构造函数内初始化时，此时与成员变量在构造函数中的位置有关。
3. 类成员在定义时，是不能初始化的
4. 类中const成员常量必须在构造函数初始化列表中初始化。
5. 类中static成员变量，必须在类外初始化。

变量的初始化顺序：

1. 初始化基类中的静态成员变量和静态代码块，按照在程序中出现的顺序初始化；
2. 初始化派生类中的静态成员变量和静态代码块，按照在程序中出现的顺序初始化；
3. 初始化基类的普通成员变量和代码块，再执行父类的构造方法；

4. 初始化派生的普通成员变量和代码块，在执行子类的构造方法；

c++的public、private、protected继承特点

c++默认是什么继承？

c++类中默认是私有继承

c++结构体中默认是public继承。

什么不能被继承？

2. 什么不能被继承

构造函数是不能继承的，也就是说，创建派生类对象时，必须调用派生类的构造函数。然而，派生类构造函数通常使用成员初始化列表语法来调用基类构造函数，以创建派生对象的基类部分。如果派生类构造函数没有使用成员初始化列表语法显式调用基类构造函数，将使用基类的默认构造函数。在继承链中，每个类都可以使用成员初始化列表将信息传递给相邻的基类。C++11 新增了一种让您能够继承构造函数的机制，但默认仍不继承构造函数。

析构函数也是不能继承的。然而，在释放对象时，程序将首先调用派生类的析构函数，然后调用基类的析构函数。如果基类有默认析构函数，编译器将为派生类生成默认析构函数。通常，对于基类，其析构函数应设置为虚的。

赋值运算符是不能继承的，原因很简单。派生类继承的方法的特征标与基类完全相同，但赋值运算符的特征标随类而异，这是因为它包含一个类型为其所属类的形参。赋值运算符确实有一些有趣的特征，下面介绍它们。

https://blog.csdn.net/qq_36607894

1、构造函数

构造函数不能被继承第一个原因：在创建派生类对象时必须调用派生类的构造函数。派生类构造函数通常使用成员列表初始化来调用基类构造函数以创建派生类中的基类部分。如果派生类没有使用成员列表初始化语法，则将使用默认的基类构造函数，如果基类没有默认的构造函数就会报错。第二个原因：因为即使继承了，它的名字和派生类的名字也不一样（构造函数名字和类名一样），不能成为派生类的构造函数，当然更不能成为普通的成员函数。在设计派生类时，对继承过来的成员变量的初始化工作也要由派生类的构造函数完成，但是大部分基类都有private属性的成员变量，它们在派生类中无法访问，更不能使用派生类的构造函数来初始化。这种矛盾在C++继承中是普遍存在的，解决问题的思路是：在派生类的构造函数中调用基类的构造函数。

2、析构函数

析构函数不能被继承，释放对象的时候需要先调用派生类析构函数然后调用基类析构函数。

3、赋值运算符=

这里面说一下为什么赋值运算符不能被继承，其实不应该说不能被继承，而是被覆盖了。第一点是在c++的编译器中如果没有定义赋值运算符，会自动加上一个默认的。所以如果赋值运算符可以被继承，则会导致派生类也会有这个赋值运算符。第二就是在如果派生类的成员和基类的成员有相同的声明，那么派生类会覆盖基类的成员，哪怕参数和数据类型都不相同，也会被覆盖。显然，B1中的赋值运算符函数名operator =和基类A1中的operator =同名，所以，A1中的赋值运算符函数int perator=(int a);被B1中的隐含的赋值运算符函数B1& perator =(const B1& robj);所覆盖。所以赋值运算符会被覆盖，不能被继承，那么派生类就不能使用基类中的赋值运算符做一些事情，就像图上说的特征标随类而异。举个例子就是基类A中赋值运算符参数是int，但是派生类赋值运算符参数是类类型，导致编译出错。

```
class A1
{
public:
    int perator=(int a)
    {
        return 8;
    }
};

class B1 : public A1
{
public:
    B1& operator =(const B1& robj); // 注意这一行是编译器添加的，派生类和基类参数都不一样，肯定会报错。
};

B1 b;
int a= 8
b=a//报错。
```

除了如上面的三个函数不能被继承外还有一些，如：

c++11的final关键字

将自身的构造函数与析构函数放在private作用域内

友元+虚继承

解释一下这个，首先要明白什么是友元和虚继承。

友元：友元的本质就是想在类外不通过成员函数访问类的私有成员，分为友元函数和友元类。友元函数内部可以直接访问私有成员。一个类A可以将另一个类B声明为自己的友元，类B的所有成员函数就都可以访问类 A 对象的私有成员。

虚继承：虚拟继承是多重继承中特有的概念。虚拟基类是为解决多重继承而出现的。如:类D继承自类B1、B2，而类B1、B2都继承自类A，因此在类D中两次出现类A中的变量和函数。为了节省内存空间，可以将B1、B2对A的继承定义为虚拟继承，而A就成了虚拟基类。虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的必要因为这样只会降低效率和占用更多的空间。

```
class CFinalClassMixin { //从这个类中继承的类都不能再被继承
    friend class Cparent;
private:
    CFinalClassMixin() {}
    ~CFinalClassMixin(){}
};
class Cparent: virtual public CFinalClassMixin, public CXXX {
public:
    Cparent() {}
    ~Cparent(){}
};
class CChild : public Cparent {
public:
    CChild() {};//编译错误
    ~CChild() {};//编译错误
};
```

如果不是虚继承，那么CChild直接调用Cparent的构造函数，这是成立的，而且CChild是不需要调用CFinalClassMixin的构造函数。若把它声明为虚继承（派生类对象一定会调用基类的构造函数），那么CChild就必须负责CFinalClassMixin构造函数的调用，这样又因为不能继承friend类，所以不能调用，造成了错误。

成员初始化列表

c++11新特性里面说的有，很详细

谈一谈new/delete和malloc/free的区别和联系（c++管理内存的方式）

- 相同点
 - 都是用来申请动态内存和释放动态内存的
 - 区别（主要是new和malloc的却别）
 - 概念上的区别
 - malloc/free是C++/C语言的标准库函数，而new/delete是C++的运算符。因此malloc仅仅只分配内存，而不会进行初始化类成员的工作，new不止分配内存，而且还是调用类的构造函数。
 - 返回类型的安全性
 - new操作符内存分配成功时候，返回的是对象类型的指针，不需要进行类型转换，从这个角度来说比较安全
 - malloc内存分配成功则返回 `void*` 类型（泛型指针），必须通过强制类型转换将 `void*` 转换成需要的类型。
 - 分配失败后返回值也不同
 - malloc失败，会返回空指针。
 - new失败，默认是抛出异常，要捕获异常 `bad_alloc`
- ```
try
{
 int *a = new int();
}
catch (bad_alloc)
{
 ...
}
```
- 是否需要指定内存大小
    - new操作符申请内存的时候不需要指定内存块的大小，编译器会自动根据类型信息来计算
    - malloc需要显示的指出内存的大小

- new的内部

调用new 操作符分配对象时会经历三个步骤：

- a. 调用operator new函数分配一块足够大的，原始的空间
- b. 编译器运行相应的构造函数以构造对象，并传入初值
- c. 构造完对象后，就返回一个指向该对象的指针。

- 后续内存的分配

当malloc分配内存后，发现后续内存分配不足的时候，可以使用realloc函数进行内存重载实现内存的扩充。realloc会先判断当前指针所指向的内存是否有足够的连续空间，如果有则可以原地扩大内存地址，并返回原来地址空间指针。如果空间不够，就从新分配一个空间，将原来的数据拷贝到新分配的内存区域，释放原来的内存区域。

new没有这样的配套设施

## 有了malloc/free为什么还要new/delete？

对于一些非内部数据类型(eg:类对象)来说，光用maloc/free无法满足要求。对象在创建的同时要自动执行构造函数，对象在消亡的时候要自动执行析构函数，而由于malloc/free是库函数而不是运算符，不在编译器的控制权限内（库函数是编译好的代码由链接器链接到为我们的代码中），也就不能自动执行构造函数和析构函数。所以，在c++中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和释放内存工作的运算符delete。

比如有一个类，你用malloc后，可以看到该对象的成员变量并没有被初始化，因此自定义类型的用malloc不合适。

既然new/delete 的功能完全覆盖了malloc/free，为什么C++不把malloc/free 淘汰出局呢？这是因为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 必须配对使用，malloc/free 也一样。

## new delete, new[] delete[]一定要配对使用吗？为什么？

首先可以明确一点：配套使用是肯定没有错的。

- **new和delete**

new和delete主要是为了为那些自定义类型的对象开辟空间，因为这些对象在创建的时候要自动执行构造函数，消亡的时候要执行析构函数，对于自定义类型对象如果不配对使用的话，可能会出现没有析构干净的情况

- **new[]和delete[]**

我们再用new[]创建数组的时候，比如一个对象大小为N，则K个数组需要K\*N个空间来构造。那当delete的时候如何知道这个数组空间的大小呢？我们会在new出来的这个空间的头部申请一个int类型的字节，4字节用来存储数组的长度，这样调用delete[]的时候就知道数组的大小，才会调用K次析构函数，释放K \* N + 4字节大小的内存。

下面针对自定义类型变量：

如果new和delete[]使用，delete的时候会找前4个字节，看要释放的内存有多大。但是使用new导致没有设备区部分4个字节用来存放数组长度，这样会导致这4个字节是未定义的，因此会调用不定次的delete。同时比如说其实地址为A，应该从A开始释放，现在会从A-4开始释放。

如果new[]和delete一起使用，程序会认为这是一个对象占用的空间，而不是数组，因此就析构一次。同时释放的是new[]中表示长度的前4个字节的地址，应该从A-4开始释放，如果不从头释放的话回出问题。

## free释放内存的理解

其实malloc算法有很多，除了glibc使用的ptmalloc以外，还有tcmalloc, jemalloc等等。但总体上个人认为主要分两个流派。第一个流派是ptmalloc为代表的隐藏头风格，第二个流派就是以tcmalloc和jemalloc为代表的位图风格。且就目前的情况看，后者在性能上要占据上风。其中一个重要原因在于，隐藏头风格的算法，元数据（即chunk size）之间间隔太远，不利于CPU cache命中。

- 如何知道要释放的空间大小？

对于glibc的malloc算法，**空间的大小记录在参数指针指向地址的前面，free的时候通过这个记录即可知道要释放的内存有多大。**

你把房子卖了，你就算还拿着钥匙，能进门，这也不是你的房子了。你照样可以把你的钱放在在这个房子里，但是不保证过一会还在。

p，你的房子所在地址，根据p这个地址，你可以找到你的房子，在有效范围内生活（如做饭、大便等），但记得别越界到邻居家了；如memcpy(p, xxx, xx);需要指定长度

free(p)后你把房子卖了，但p这个地址还存在你的记忆中，除非你把你房子的地址从你记忆中抹去，即p = NULL；

如果你没有抹去对你房子的记忆，即p != NULL；你仍然可以找到那所房子，但是那房子已经不属于你了；

如果你还要进房子生活（如做饭、大便等），即对p进行写入操作，那么你已经违法了

## 参考链接

### 被free回收的内存是立即返还给操作系统吗？

这个需要看源码。

被free回收的内存会首先被ptmalloc使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时ptmalloc也会尝试对小块内存进行合并，避免过多的内存碎片

## 怎么在栈上栈上分配内存？

### alloca函数

alloca函数分配的内存不需要手动释放，和普通的栈上对象的处理一样：超出作用域自动回收内存。

alloca函数并不被推荐使用，因为它不安全。对于一个很大的数据结构，如果采用默认的内存分配模式，会引起爆栈。但是你用alloca去分配，虽然能够在栈上得到一个地址，但是这个地址可能超出了栈的边界，可能对其他的内存空间造成了破坏。并且alloca函数本身不知道是否踩了其他的内存空间。

使用场景：有时候我们只是想暂时打印日志信息或者错误信息，并不需要长期保存其内容，随着程序运行超出数组作用域，[内存](#)自动释放。下次需要的时候再动态的申请，不用管理释放，这样很方便

```
if(!result){
 int length;
 length = getLogLength(); //凭空捏造的函数
 char* message = (char*)alloca(sizeof(char) * length);
 message = getLogInformation(); //凭空捏造的函数
 std::cout << message << std::endl;
}
```

## 带返回值的函数如果不return会怎么样？

首先不是BUG，这种情况属于**未定义的行为**，而未定义的行为不会导致编程失败。

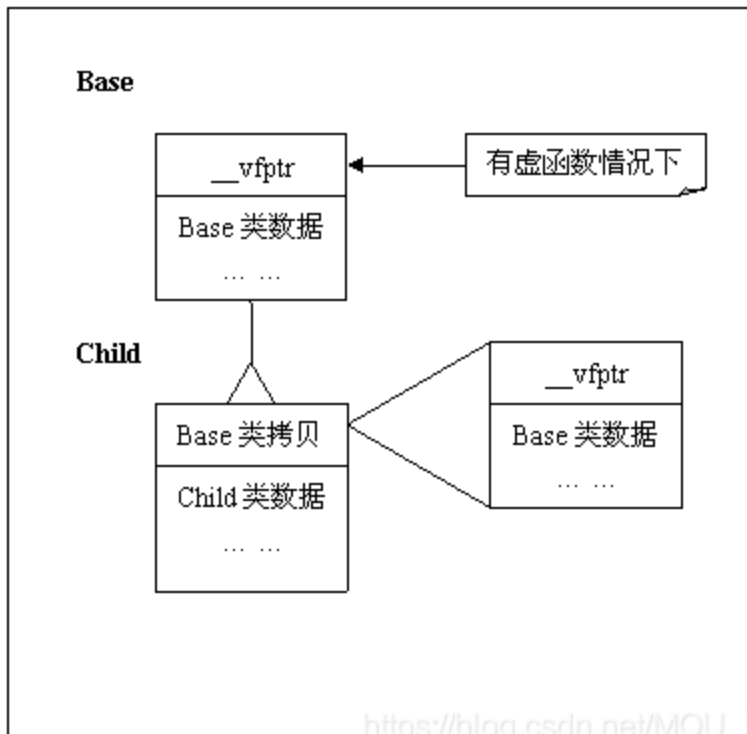
当main函数没有return结尾的时候会在生成的目标文件中自动加入return 0

## C++ 四种强制类型转换

[参考链接](#)

### 向上类型转换和向下类型转换

说这个之前要说一下类对象的内存结构：



**\*\*向上类型转换：\*\*upcast**，把派生类的指针或引用转换成基类的指针或者引用是安全的（或者说基类指针指向派生类）；因为用转换后的指针只能访问基类部分的函数时候，都可以访问，肯定安全。这个是隐式转换，c++认为是安全的。

**\*\*向下类型转换：\*\*downcast**，把基类指针或引用转换成派生类表示（或者说把派生类指针指向基类）时，由于没有动态类型检查，所以是不安全的。如果转换后的指针访问派生类新增加的函数，这个时候基类中本来没有这个函数，那就会出错。

很形象，向上就是派生类指针变成基类指针，向下就是基类指针变成派生类指针。

## 四种强制类型转换

### 1. **static\_cast**

用于基本数据类型之间的转换

void\*和其他类型指针之间的转换

子类对象的指针转换成父类对象指针

以上三个都是隐式转换，最好吧所有隐式转换都用static\_cast代替

### 2. **dynamic\_cast**

只用于对象的指针和引用，主要用于执行“安全的向下转型”，因为downcast是不安全的

dynamic\_cast是唯一一个在运行时处理的，因为我们转换后的指针如果请求一块无效的内存的话是会报错的，但是用该强转后会返回null，即转换成功会返回引用或者指针，失败返回null。如果一个引用类型执行了类型转换并且这个转换是不可能的，运行时一个 `bad_cast` 的异常类型会被抛出：

### 3. **const\_cast**

const\_cast转换符是用来移除const或volatile属性。一般用于强制消除对象的常量性。而C不提供消除const的机制（已验证）。

但是不能用于去除变量的常量性，而是去除指向对象的引用或指针的常量性，因此

去除对象必须是指针或者引用。

指向常量的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。

#### 4. `reinterpret_cast`

```
reinterpret_cast<type-id> (expression)
```

`reinterpret`的意思是重新解释，可以将任意类型指针转换为其他类型的指针。所以他的`type-id`必须是一个指针、引用、算术类型。

这个操作符能够在非相关的类型之间转换。它可以把一个指针转换成一个整数，也可以把一个整数转换成一个指针。

底层实现是从一个指针到别的指针的值的二进制拷贝。

## 为什么要引入四种强制类型转换

C语言可以在任意类型之间进行转换，但是有一点就是不安全。可能会不经意间将指向`const`对象的指针转换成非`const`对象的指针，也有可能将基类对象指针转换成派生类对象的指针。因此这四种强制类型转换的代码更加严谨规范

同时C++风格的强转很清晰的知道在干什么，只要扫一眼就知道这样转换的目的

## 为什么说不要使用 `dynamic_cast`

没有说一定不能用，而是需要在恰当的场合使用恰当的特性。比如：能在编译时解决掉的问题没必要留到运行时、能用多态搞定的事情也没必要使用 `dynamic_cast` 和 `typeid` 等。

## RAII基本理解与使用

**\*\*基本概念：**RAII是C++中的一个惯用法，即“Resource Acquisition Is Initialization”，翻译为“资源获取就初始化”。是一种资源管理技术。C++之父说这种技术是依赖于类中构造函数和析构函数的性质以及与异常处理的交互性质来管理资源。

**\*\*提出原因：**首先要明确在计算机系统中，资源的数量是有限的，一定要合理管理和使用有限的资源。比如内存，文件，套接字等等吧。因此在使用资源的时候要遵循三个步骤：

1. 获取资源
2. 使用资源
3. 释放资源



正常情况下没什么问题，大家都非常规规矩矩的这样做。但是我们都知程序员很懒，一般不喜欢干重复性很高的工作，因此有两种情况下会出现一些问题。第一种比如说对文件操作，考虑一种极端情况，各种if判断后都要释放资源，也就是说释放资源的语句要写很多次，这样是非常麻烦的。第二种情况就是在使用资源的过程中程序会抛出异常，我们必须用catch来捕获所有异常然后关闭文件，当控制流程特别复杂的时候就很烦，代码很臃肿。而且有时候释放资源的语句就不会被执行。那么有些资源就是放不掉，因此Bjarne Stroustrup就想到不管啥情况都能释放资源的就是析构函数，因为stack unwinding（栈展开）导致析构函数能被执行。因此将资源的初始化和释放都放到一个类中就能解决上面遇到的问题。

stack winding:When program run, each function(data, registers, program counter, etc) is mapped onto the stack as it is called. Because the function calls other functions, they too are mapped onto the stack. This is stack winding.

stack unwinding:Unwinding is the removal of the functions from the stack in the reverse order.(展开是以相反的顺序从堆栈中删除函数)。在c++中，系统必须确保调用所有创建起来的局部对象的析构函数。

**\*\*总结：**\*\*在构造函数中申请分配资源，在析构函数中释放资源。因为C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超出作用域的时候会自动调用析构函数。所以，在RAII的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。**RAII的核心思想是将资源或者状态与对象的生命周期绑定，通过C++的语言机制，实现资源和安全的管理,智能指针是RAII最好的例子。**

RAII是c++区别于其他所有编程语言的重要特性。最初学习c++时的教条是new和delete一定要配套使用，但是使用RAII思想后，改成每一个资源配置动作都应该在单一语句执行，获得资源后立刻交给对象去管理，一般不要出现delete，用智能指针。

## C++11的enum class 、enum struct 和 enum

枚举类型(enumeration)使我们可以将一组整型常量组织在一起。每个枚举类型定义了一种新的类型。枚举属于字面值常量类型。

- 旧版本enum存在的问题
  - i. 没有非常完全的类型安全。即不同枚举之间不能赋值，同时整形不能向枚举类型转换，但是枚举可以向整形转。
  - ii. 无法确定数据的类型，导致无法明确枚举类型所占用的内存大小。
  - iii. 枚举中的成员可以在括号外部直接访问，而不需要使用域运算符。

- enum class 、 enum struct

有更好的类型安全和封装的特性。

- i. 与整形之间不会发生隐式类型转换，除非用static\_cast强制转换
- ii. 可以指定底层的数据类型，默认是int
- iii. 需要通过域运算符来访问枚举成员

## 内存泄漏？出现内存泄漏如何调试？

内存泄露一般指的是堆内存的泄露，即用户自己开辟的内存空间。应用程序使用malloc、realloc、new等函数从堆中分配到一块内存后，必须调用free或delete进行回收，否则这块内存不能继续被使用。内存泄漏会因为减少可用内存的数量从而降低计算机的性能。最终，在最糟糕的情况下，过多的可用内存被分配掉导致全部或部分设备停止正常工作，或者应用程序崩溃。内存泄漏可能不严重，甚至能够被常规的手段检测出来。在现代操作系统中，一个应用程序使用的常规内存存在程序终止时被释放。这表示一个短暂运行的应用程序中的内存泄漏不会导致严重后果。在C++中出现内存泄露的主要原因就是程序猿在申请了内存后( malloc(), new )，没有及时释放没用的内存空间，甚至消灭了指针导致该区域内存空间根本无法释放。

### 内存泄漏的原因

1. malloc/new和delete/free没有匹配
2. new[] 和 delete[]也没有匹配
3. 没有将父类的析构函数定义为虚函数，当父类的指针指向子类对象时，delete该对象不会调用子类的析构函数

### 检测手段

1. 良好的程序设计能力，把new和delete全部都封装到构造函数和析构函数中，保证任何资源的释放都在析构函数中进行
2. 智能指针（万一被问道，这也是一个问题）
3. valgrind，这个可以打印出发生内存泄露的部分代码
4. linux使用swap命令观察还有多少可以用的交换空间，两分钟内执行三四次，肉眼看交换区是不是变小了
5. 使用/usr/bin/stat工具如netstat、vmstat等。如果发现波段有内存被分配且没有释放，有可能进程出现了内存泄漏。

### valgrind

## 🍉C++11新特性

### auto和decltype

auto和decltype都是c++11新增的关键字，都用于自动类型推导，但是语法格式有区别。一句话概括，当你需要某个表达式的返回值类型而又不想实际执行它时用decltype。

- **auto**

auto的出现有对我的代码来说两个作用：

- i. 为了避免太长的类型描述影响代码可读性。比

如 `std::vector<MyType>::iterator it = myArray.begin();`

- ii. 这个类型不是我们所关注的，也会用。比如遍历容器中的值，用到auto自动帮我们获得定义变量的类型所以auto定义的变量必须有初始值。

auto可以推断基本类型，也可以推断引用类型，当推断引用类型时候，将引用对象的类型作为推断类型（重要）

auto的自动类型推断发生在编译期，所以使用auto并不会造成程序运行时效率的降低。

编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数组类型的推导

- **decltype**

一句话概括，当你需要某个表达式的返回值类型而又不想实际执行它时用decltype。

有时候我们想从表达式中推断出要定义的类型，不会真的求表达式的值。

decltype 是为了解决复杂的类型声明而使用的关键字

- i. auto忽略顶层const，decltype保留顶层const；
- ii. 对引用操作，auto推断出原有类型，decltype推断出引用；（重要）
- iii. 对解引用操作，auto推断出原有类型，decltype推断出引用；
- iv. auto推断时会实际执行，decltype不会执行，只做分析。

- **有了auto为什么还需要decltype？**

decltype 可以让你获得编译期的类型。auto 不能。所以当你需要某个表达式的返回值类型而又不想实际执行它时用decltype。

```
int a=8, b=3;
auto c=a+b; //运行时需要实际执行a+b, 哪怕编译时就能推导出类型
decltype(a+b) d; //编译期类型推导
//不可以用auto c; 直接声明变量, 必须同时初始化。
```

## C++智能指针

<https://aijishu.com/a/1060000000286819>

智能指针是一个 RAII (Resource Acquisition is initialization) 类模型, 用来动态的分配内存。

把指针用类封装然后实例化成对象, 在对象过期的时候, 让析构函数删除指向的内存

它提供所有普通指针提供的接口, 却很少发生异常。在构造中, 它分配内存, 当离开作用域时, 它会自动释放已分配的内存。这样的话, 程序员就从手动管理动态内存的繁杂任务中解放出来了。

| 指针类别                    | 支持        | 备注                                       |
|-------------------------|-----------|------------------------------------------|
| <code>unique_ptr</code> | C++ 11    | 拥有独有对象所有权语义的智能指针                         |
| <code>shared_ptr</code> | C++ 11    | 拥有共享对象所有权语义的智能指针                         |
| <code>weak_ptr</code>   | C++ 11    | 到 <code>std::shared_ptr</code> 所管理对象的弱引用 |
| <code>auto_ptr</code>   | C++ 17中移除 | 拥有严格对象所有权语义的智能指针                         |

## 为什么要用智能指针？

原因1：内存泄露，即new和delete不匹配

原因2：多线程下对象析构问题，造成这个问题本质的原因是类对象自己销毁(析构)的时候无法对自己加锁,所以要独立出来,采用这个中间层(shared\_ptr).

## auto\_ptr

最早的智能指针, c++11之后就删除了, 有以下问题：

1. `auto_ptr`不能指向一组对象, 不能和`new[]`一起使用

2. auto\_ptr不能和标准容器一起用
3. 容易野指针。有两个auto\_ptr比如说 p1和p2，当在函数参数传递指针的时候，所有权也会发生转移。具体来说比如p1指向一块内存，然后p2是p1的拷贝，因此p2也指向了这块内存。当函数调用完后，p2指向的这块内存释放掉了，p1不就成了一个野指针。

所以这个东西很烂，没什么人用。

## shared\_ptr

### 概述

共享所有权，也就是说多个指针可以指向一个相同的对象，当最后一个shared\_ptr离开作用域的时候才会释放掉内存。

实现原理：在shared\_ptr内部有一个共享引用计数器来自动管理，计数器实际上就是指向该资源指针的个数，每当复制一个 shared\_ptr，引用计数会 + 1。当一个 shared\_ptr 离开作用域时，引用计数会 - 1，当引用计数为 0 的时候，则delete 内存。这样相比auto来说就好很多，当计数器为0的时候指针才会彻底释放掉这个资源。

### 线程安全问题？

参考，有时间总结一下

Boost 文档对于 shared\_ptr 的线程安全有一段专门的记述，内容如下：

shared\_ptr objects offer the same level of thread safety as built-in types.

A shared\_ptr instance can be "read" (accessed using only const operations) simultaneously by multiple threads. 一个 shared\_ptr 实例可以同时被多个线程“读”（仅使用不变操作进行访问）

Different shared\_ptr instances can be "written to" (accessed using mutable operations such as operator= or reset) simultaneously by multiple threads (even when these instances are copies, and share the same reference count underneath.) Any other simultaneous accesses result in undefined behavior. 不同的 shared\_ptr 实例可以同时被多个线程“写入”（使用类似 operator= 或 reset 这样的可变操作进行访问）（即使这些实例是拷贝，而且共享下层的引用计数）。

任何其它的同时访问的结果会导致未定义行为。”

总结：1、同一个shared\_ptr被多个线程“读”是安全的。2、同一个shared\_ptr被多个线程“写”是不安全的。3、共享引用计数的不同的shared\_ptr被多个线程“写”是安全的。

所以说我们可以借助shared\_ptr实现线程安全的对象释放，但是shared\_ptr本身不是100%线程安全的，不考虑其管理对象的安全级别

看线程安全问题之前最好还是要看一下源码解析。

shared\_ptr 可能的线程安全隐患大概有如下几种，一是引用计数的加减操作是否线程安全，二是shared\_ptr修改指向时，是否线程安全。

1. shared\_ptr 的引用计数是原子操作的，所以引用计数的加减是线程安全的。
2. shared\_ptr修改指针指向的时候会不安全。 同一个shared\_ptr被多个线程“读”是安全的。同一个shared\_ptr被多个线程“写”是不安全的(多个线程操作同一个shared\_ptr对象)。如下面的代码：

```
void fn(shared_ptr<A>& sp) {
 ...
 if (..) {
 sp = other_sp;
 } else if (...) {
 sp = other_sp2;
 }
}
```

当你在多线程回调中修改shared\_ptr指向的时候。shared\_ptr内数据指针要修改指向，sp原先指向的引用计数的值要减去1，other\_sp指向的引用计数值要加1。然而这几步操作加起来并不是一个原子操作，如果多少线程都在修改sp的指向的时候，那么有可能会出问题。比如在导致计数在操作减一的时候，其内部的指向，已经被其他线程修改过了。引用计数的异常会导致某个管理的对象被提前析构，后续在使用到该数据的时候触发core dump。当然如果你没有修改指向的时候，是没有问题的。

测试：在多个线程中同时对一个shared\_ptr循环执行两遍swap。

shared\_ptr的swap函数的作用就是和另外一个shared\_ptr交换引用对象和引用计数，是写操作。执行两遍swap之后，shared\_ptr引用的对象的值应该不变。

```

#include <stdio.h>
#include <tr1/memory>
#include <pthread.h>

using std::tr1::shared_ptr;

shared_ptr<int> gp(new int(2000));

//多线程操作不同的shared_ptr对象, 安全
//该函数拷贝了一个p1, 用p1进行操作
shared_ptr<int> CostaSwapSharedPtr1(shared_ptr<int> & p)
{
 shared_ptr<int> p1(p);
 shared_ptr<int> p2(new int(1000));
 p1.swap(p2);
 p2.swap(p1);
 return p1;
}

//多线程操作指向同一个shared_ptr对象, 不安全
//直接对全局变量gp进行操作
shared_ptr<int> CostaSwapSharedPtr2(shared_ptr<int> & p)
{
 shared_ptr<int> p2(new int(1000));
 p.swap(p2);
 p2.swap(p);
 return p;
}

//线程执行函数
void* thread_start(void * arg)
{
 int i =0;
 for(;i<100000;i++)
 {
 shared_ptr<int> p= CostaSwapSharedPtr2(gp);
 if(*p!=2000)
 {
 printf("Thread error. *gp=%d \n", *gp);

```

```

 break;
 }
}
printf("Thread quit \n");
return 0;
}

int main()
{
 pthread_t thread;
 int thread_num = 10, i=0;
 pthread_t* threads = new pthread_t[thread_num];
 for(;i<thread_num;i++)
 pthread_create(&threads[i], 0 , thread_start , &i);
 for(i=0;i<thread_num;i++)
 pthread_join(threads[i],0);
 delete[] threads;
 return 0;
}

```

解决方案之一就是加锁。所以甭管安全不安全，加锁就完事儿了。

## 所管理数据的线程安全性

我们上面说的是针对shared\_ptr本身的线程安全问题。但是用shared\_ptr管理对象的线程安全问题又是另一会儿事。

如果shared\_ptr管理的数据是STL容器，那么多线程如果存在同时修改的情况，是极有可能触发core dump的。比如多个线程中对同一个vector进行push\_back，或者对同一个map进行了insert。甚至是对STL容器中并发的做clear操作，都有可能出发core dump，当然这里的线程不安全性，其实是其所指向数据的类型的线程不安全导致的，并非是shared\_ptr本身的线程安全性导致的。尽管如此，由于shared\_ptr使用上的特殊性，所以我们有时也要将其纳入到shared\_ptr相关的线程安全问题的讨论范围内。

## 拥有的一些方法

### 1. reset方法

reset() 释放并销毁原生指针。如果参数为一个新指针，将管理这个新指针

”当智能指针调用了reset函数的时候,就不会再指向这个对象了,所以如果还有其它智



能指针指向这个对象,那么其他的智能指针的引用计数会减1

[cppreference](#)参考

## 2. make\_shared方法

返回一个指定类型的 `std::shared_ptr`, 和`shared_ptr`的构造函数一样都是用来初始化一个智能指针对象的。但是效率上有所不同：

`make_shared`执行一次堆分配, 而`shared_ptr`构造函数执行两次

读过源码的应该都知道, `shared_ptr`里面维护了两个部分, 或者叫两个控制块：

- 引用计数相关控制块, 添加删除等等
- 被管理的对象, 原生指针

如果使用`new`即自身构造函数来分配内存的话, 就会对于上面两部分执行`heap-allocation`, 即两次堆分配, 如下图：

image但是如果使用`make_shared`的话, 只用执行一次`heap-allocation`, 如下图：  
image**其次使用`make_shared`还是异常安全的**

在`c++17`之后就不是问题了, 因为函数的求值顺序发生了变化, 函数的每个参数都需要在计算其他参数之前完全执行

比如下面代码：

```
//潜在的资源泄露
processWidget(std::shared_ptr<Widget>(new Widget), computePriority());
```

在运行期, 函数的参数必须在函数被调用前被估值, 所以在调用`processWidget`时, 下面的事情肯定发生在`processWidget`能开始执行之前：

- 1、表达式 `new Widget` 必须被估值即一个`Widget`必须被创建在堆上。
- 2、`std::shared_ptr`（负责管理由`new`创建的指针）的构造函数必须被执行。
- 3、`computePriority`必须跑完。

编译器不需要必须产生这样顺序的代码。但 `new Widget` 必须在`std::shared_ptr`的构造函数被调用前执行, 因为`new`的结构被用为构造函数的参数, 但是`computePriority`可能在这两个调用前（后, 或很奇怪地, 中间）被执行。也就是, 编译器可能产生出这样顺序的代码：

```
执行“new Widget”。
执行computePriority。
执行std::shared_ptr的构造函数。
```

如果`computePriority`产生了一个异常, 则在第一步动态分配的`Widget`就会泄露了, 因为它永远不会被存放到在第三步才开始管理它的`std::shared_ptr`中。

使用std::make\_shared可以避免这样的问题。

### 3. swap方法

swap 交换两个 shared\_ptr 对象(即交换所拥有的对象)

### 4. shared\_from\_this

我们往往会需要在类内部使用自身的 shared\_ptr，如下代码：

```
class Widget
{
public:
 void do_something(A& a)
 {
 a.widget = 该对象的 shared_ptr;
 }
}
```

上述代码是说我们将当前对象的sp交由对象a管理，那就意味着当前对象的生命周期的结束不能早于对象 a。因为对象 a 在析构之前还是有可能使用到 a.widget。如果我们直接 a.widget = this; 那肯定不行，因为这样并没有增加当前 shared\_ptr 的引用计数。shared\_ptr 还是有可能早于对象 a 释放。如果我们使用 a.widget = std::make\_shared<Widget>(this);，肯定也不行，因为这个新创建的 shared\_ptr，跟当前对象的 shared\_ptr 毫无关系。当前对象的 shared\_ptr 生命周期结束后，依然会释放掉当前内存，那么之后 a.widget 依然是不合法的。对于这种，需要在对象内部获取该对象自身的 shared\_ptr，那么该类必须继承

std::enable\_shared\_from\_this<T>。

**\*\*总结：**\*\*智能指针的优势在于一旦某个对象不再被引用，系统会立刻回收内存，通常发生在关键任务完成后的清理时期。同时，内存中所有的对象都是有用的，绝对没有垃圾占内存的现象出现。

## weak\_ptr

weak\_ptr 比较特殊，它主要是为了配合 shared\_ptr 而存在的。就像它的名字一样，它本身是一个弱指针，因为它本身是不能直接调用原生指针的方法的。如果想要使用原生指针的方法，需要将其先转换为一个 shared\_ptr。那 weak\_ptr 存在的意义到底是什么呢？

weak指针的出现是为了解决shared指针循环引用造成的内存泄漏的问题。由于 shared\_ptr 是通过引用计数来管理原生指针的，那么最大的问题就是循环引用（比如 a 对象持有 b 对象，b 对象持有 a 对象），这样必然会导致内存泄露(无法删除)。而 weak\_ptr 不会增加引用计数，因此

将循环引用的一方修改为弱引用，可以避免内存泄露。

如下述代码：

```
struct A{
 shared_ptr b;
};
struct B{
 shared_ptr<A> a;
};
shared_ptr<A> pa = make_shared<A>();
shared_ptr pb = make_shared();
pa->b = pb;
pb->a = pa;
```

pa 和 pb 存在着循环引用，根据 shared\_ptr 引用计数的原理，pa 和 pb 都无法被正常的释放，因为我们需要对方先释放。对于这种情况，我们可以使weak\_ptr：

```
struct A{
 shared_ptr b;
};
struct B{
 weak_ptr<A> a;
};
shared_ptr<A> pa = make_shared<A>();
shared_ptr pb = make_shared();
pa->b = pb;
pb->a = pa;
```

weak\_ptr 不会增加引用计数，因此可以打破 shared\_ptr 的循环引用。

当创建一个shared指针对象时候，该指针所指向的资源数为1，当用shared对象指针创建一个weak对象时候，资源计数器没有变化。weak\_ptr的构造和析构并不会改变引用计数的大小，同时由于weak\_ptr没有重载运算符\*，->，因此他不操作资源，只是观测

## 方法

1. expired() 判断所指向的原生指针是否被释放，如果被释放了返回 true，否则返回 false

2. `use_count()` 返回原生指针的引用计数
3. `lock()` 返回 `shared_ptr`，如果原生指针没有被释放，则返回一个非空的 `shared_ptr`，否则返回一个空的 `shared_ptr`
4. `reset()` 将本身置空

有以下问题：

1. 要是用weak指针对象如何判断该指针指向的对象是否销毁？  
答：weak\_ptr类中有一个成员函数`lock()`，这个函数可以返回一个指向共享对象的`shared_ptr`，如果weak指针所指向的资源不存在，那么lock函数返回一个空`shared_ptr`，通过这个可以判断
2. weak\_ptr类中没有重载`operator *`和`operator ->`，因此不能使用weak\_ptr类对象直接访问指针所指向的资源，因此如果想要访问weak\_ptr指向的资源的时候，必须首先使用lock成员函数获取到该weak\_ptr所指向资源的`shared_ptr`的对象，然后再去访问。这样做也为了避免我们在写程序时，忘记考虑weak\_ptr所指向的资源被释放的情况。

弱指针的使用有两个：第一当 parent 类持有 child 的 `shared_ptr`，child 持有指向 parent 的 `weak_ptr`。第二是定义对象时，用强智能指针`shared_ptr`，在其它地方引用对象时，使用弱智能指针`weak_ptr`。

## unique\_ptr

`unique_ptr` 的核心特点就如它的名字一样，它拥有对持有对象的唯一所有权。即两个 `unique_ptr` 不能同时指向同一个对象。

那具体这个唯一所有权如何体现呢？

1. `unique_ptr` 不能被复制到另外一个 `unique_ptr`
2. `unique_ptr` 所持有的对象只能通过转移语义将所有权转移到另外一个 `unique_ptr`

```
std::unique_ptr<A> a1(new A());
std::unique_ptr<A> a2 = a1; //编译报错，不允许复制
std::unique_ptr<A> a3 = std::move(a1); //可以转移所有权，所有权转义后a1不再拥有任何指针
```

`unique_ptr` 本身拥有的方法主要包括：

1. `get()` 获取其保存的原生指针，尽量不要使用

2. `bool()` 判断是否拥有指针
3. `release()` 释放所管理指针的所有权，返回原生指针。但并不销毁原生指针。
4. `reset()` 释放并销毁原生指针。如果参数为一个新指针，将管理这个新指针

```
std::unique_ptr<A> a1(new A());
A *origin_a = a1.get();//尽量不要暴露原生指针
std::unique_ptr<A> a2(a1.release());//常见用法，转义拥有权
a2.reset(new A());//释放并销毁原有对象，持有一个新对象
a2.reset();//释放并销毁原有对象，等同于下面的写法
a2 = nullptr;//释放并销毁原有对象
```

## lambda表达式

我觉得lambda表达式的出现很简洁，也使得代码变得没那么膨胀吧

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

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

- 捕获列表 [capture]

捕获一定范围内的变量，有以下几种方式：

- i. `[]` - 表示不捕捉任何变量
- ii. `[&]` - 捕获外部作用域中所有变量，并作为引用在函数体内使用 (按引用捕获)
- iii. `[=]` - 捕获外部作用域中所有变量，并作为副本在函数体内使用 (按值捕获)
- iv. `[=, &foo]` - 按值捕获外部作用域中所有变量，并按照引用捕获外部变量 `foo`
- v. `[this]` - 捕获当前类中的 `this` 指针，让 lambda 表达式拥有和当前类成员函数同样的访问权限

```

void output(int x, int y)
{
 auto x1 = [] {return m_number; }; // error
 auto x2 = [=] {return m_number + x + y; }; // ok
 auto x3 = [&] {return m_number + x + y; }; // ok
 auto x4 = [this] {return m_number; }; // ok
 auto x5 = [this] {return m_number + x + y; }; // error
 auto x6 = [this, x, y] {return m_number + x + y; }; // ok
 auto x7 = [this] {return m_number++; }; // ok
}

```

- 参数列表 (params): 和普通函数的参数列表一样, 如果没有参数参数列表可以省略不写
- opt 选项, 不需要可以省略, 一般有两个
  - i. mutable: 可以修改按值传递进来的拷贝 (注意是能修改拷贝, 而不是值本身)
  - ii. exception: 指定函数抛出的异常, 如抛出整数类型的异常, 可以使用 throw ();
- 返回值类型  
很多时候, lambda 表达式的返回值是非常明显的, 因此在 C++11 中允许省略 lambda 表达式的返回值。
- 函数体: 函数的实现, 这部分不能省略, 但函数体可以为空。

## nullptr和null的区别

[非常详细的参考链接0](#)

- 首先给出NULL在C和C++中的定义

```

#ifdef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif

```

要明白一点儿, NULL是一个无类型的东西, 而且是一个宏。而宏这个东西, 从C++诞生开始, 就是C++之父嗤之以鼻的东西, 他推崇尽量避免宏。

下面看一段代码：

```
void f(void*){}

void f(int){}

int main()
{
 f(NULL); // what function will be called?
}
```

我们本来是想用NULL来代替空指针，但是在将NULL输入到函数中时，它却选择了int形参这个函数版本，所以是有问题的，这就是用NULL代替空指针在C++程序中的二义性。而引入了 *nullptr*，这个问题就得到了真正解决，会很顺利的调到void f(void)这个版本。\*

**nullptr，可以保证在任何情况下都代表空指针**

nullptr并非整型类别，甚至也不是指针类型，但是能转换成任意指针类型。nullptr的实际类型是std::nullptr\_t。

## 成员列表初始化

首先要说明成员列表初始化的格式

其次说明必须被初始化的几个情况：

1. const修饰的成员变量。因为const只能被初始化不能被赋值
2. 类中有对象或者继承关系时（没有默认构造函数的情况下）。首先要明确一点就是如果一个类有自定义的构造函数时后，编译器就不会创建默认构造函数了。  
所以类中引用了其他的类时候（成员类），那么这个被引用的类如过有自定义的构造函数没有默认构造函数时候，就会报错  
还有继承的时候也是，创建派生类对象时会先创建基类对象，调用基类的构造函数，如果基类有自定义的构造函数的话你有没有列表初始化，编译器会调用默认构造函数，但是你没有就会出错
3. 引用的数据成员。因为const和引用都必须初始化而不能被赋值，都需要成员列表初始化。

成员初始化的一些特点：

1. 成员的初始化顺序与成员的声明顺序相同
2. 成员的初始化顺序与初始化列表中的位置无关
3. 初始化列表先于构造函数的函数体执行

**为什么用成员初始化列表会快一些？或者类中初始化数据成员与对数据成员赋值有什么区别？**

（区分基本类型和非基本类型）

答：构造函数执行有两个阶段，首先是初始化阶段，这个阶段所有类中的成员都会被初始化，即使成员没有出现在初始化列表中。第二个阶段是计算阶段，一般是执行构造函数体内部的赋值操作。对于基本内置类型，没啥区别，所以不讨论。主要讨论的是类类型相关的问题。举个例子：

```
class classA {...};
class classB
{
public:
 //赋值
 classB(classA a) {mA = a;}
 //成员列表初始化
 classB(classA a): mA(a) {}
private:
 classA mA;
};
```

对上述两个构造函数的实现来说，结果上是一样的，但在内部实现上有很大的区别。使用成员列表初始化的时候，直接调用的是A的拷贝构造函数完成。但是当赋值的时候，首先调用一次A的构造函数生成对象a，然后在调用一次A的构造函数生成对象mA，然后执行赋值构造函数。可以看到成员列表初始化的情况下少一次构造函数的调用，效率上肯定会提高。

就好比：

```
//拷贝构造
classA a; //1次默认构造函数
classA b = a;
//赋值构造
classA a; //1次默认构造函数
classA b; //2次默认构造函数
b = a;
```



## 右值引用

## for循环

## 可变参数模板

可变参数模板和普通模板的语义是一样的，只是写法上稍有区别，声明可变参数模板时需要在typename或class后面带上省略号“...”。比如我们常常这样声明一个可变模版参数：

template<typename...>或者template<class...>，一个典型的可变模版参数的定义是这样的：

```
template <class... T>
void f(T... args);
```

省略号的作用如下：

1. 声明一个参数包T... args，这个参数包中可以包含0到任意个模板参数；
2. 在模板定义的右边，可以将参数包展开成一个一个独立的参数。

## C++11中的原子操作（atomic operation）

### 参考链接

所谓的原子操作，取的就是“原子是最小的、不可分割的最小个体”的意义，它表示在多个线程访问同一个全局资源的时候，能够确保所有其他的线程都不在同一时间内访问相同的资源。也就是他确保了在同一时刻只有唯一的线程对这个资源进行访问。这有点类似互斥对象对共享资源的访问的保护，但是原子操作更加接近底层，因而效率更高。

互斥对象的使用，保证了同一时刻只有唯一的一个线程对这个共享进行访问，从执行的结果来看，互斥对象保证了结果的正确性，但是也有非常大的性能损失

在以往的C++标准中并没有对原子操作进行规定，我们往往是使用汇编语言，或者是借助第三方的线程库，例如intel的pthread来实现。在新标准C++11，引入了原子操作的概念，并通过这个新的头文件提供了多种原子操作数据类型，例如，atomic\_bool,atomic\_int等等，如果我们在多个线程中对这些类型的共享资源进行操作，编译器将保证这些操作都是原子性的，也就是说，确保任意时刻只有一个线程对这个资源进行访问，编译器将保证，多个线程访问这个共享资源的正确性。从而避免了锁的使用，提高了效率。

## C++11的std::function和std::bind

C++11的std::function和std::bind作用的对象叫做可调用对象，先解释一下什么是可调用对象c++中有几种可调用对象，比如函数、函数指针、lambda表达式、bind对象、函数对象。

**\*\*函数：**\*\*这个刚学编程的时候大家都知道了，所以说这就不说了

**\*\*函数指针：**\*\*函数指针就是指向函数的指针，把函数当做变量来处理，大部分用作回调函数

**lambda表达式：**（匿名函数对象，一个C++的语法糖） 一个lambda表达式表示一个可调用的代码单元，我们可以将其理解为有一个未命名的内联函数，与任何函数类似，一个lambda具有一个返回类型，一个参数列表和一个函数体，并且这个函数体是可以定义在其他函数内部的。

**\*\*重载了函数调用符的类：**\*\*C++的类厉害的地方之一就是可以重载函数运算，就是retType operator( )(parameter...){ }的形式，这种重载了函数调用符的类可以直接让我们用类名来实现函数的功能，比如：

```
class FunctionalClass
{
public:
 bool operator()(const int &a, const int &value)
 {
 return a >= value;
 }
};
```

调用运算符就是 ()，重载了调用运算符的类就叫做函数对象，在泛型编程中大量使用函数对象作为实参，比如头文件functiona中定义了算术运算符，关系运算符，逻辑运算符的模板类作为函数对象来调用，比如greater源码如下：

```
template <class T>
struct greater
{
 bool operator()(const T& x, const T& y) const{return x > y;}
};
```

**std::bind** : \*\*用来生产一个可调用对象来适应原对象的参数列表。

## std::function

从上面看由于可调用对象的定义方式比较多，但是函数的调用方式比较类似，因此我们可以使用一个统一的方式保存可调用对象或者传递可调用对象，因此就有了std::function。

std::function是一个可调用对象包装器，是一个类模板，可以容纳除了类成员函数指针之外的所有可调用对象，它可以用统一的方式处理函数、函数对象、函数指针，lambda表达式并允许保存和延迟它们的执行。下面是代码示例：

```
typedef std::function<int(int, int)> comfun;
// 普通函数
int add(int a, int b) { return a + b; }
// lambda表达式
auto mod = [](int a, int b){ return a % b; };
// 函数对象类
struct divide{
 int operator()(int denominator, int divisor){
 return denominator/divisor;
 }
};

int main(){
 comfun a = add;
 comfun b = mod;
 comfun c = divide();
 std::cout << a(5, 3) << std::endl;
 std::cout << b(5, 3) << std::endl;
 std::cout << c(5, 3) << std::endl;
}
```

总结：std::function对C++中各种可调用实体(普通函数、Lambda表达式、函数指针、以及其它函数对象等)的封装，形成一个新的可调用的std::function对象，简化调用。std::function对象是对C++中现有的可调用实体的一种类型安全的包裹(如：函数指针这类可调用实体，是类型不安全的)。

## std::bind

std::bind可以看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来适应原对象的参数列表。std::bind将可调用对象与其参数一起进行绑定，绑定后的结果可以使用std::function保存。std::bind主要有以下两个作用：

1. 将可调用对象和其参数绑定成一个仿函数；
2. 只绑定部分参数，减少可调用对象传入的参数。

bind的调用形式如下：

```
auto newCallable = bind(callable, arg_list);
```

该形式表达的意思是：当调用 newCallable 时，会调用 callable，并传给它 arg\_list 中的参数。需要注意的是：arg\_list中的参数可能包含形如 \_n 的名字。其中n是一个整数，这些参数是占位符，表示newCallable的第n个参数，它们占据了传递给newCallable的参数的位置。数值n表示生成的可调用对象中参数的位置：\_1 为newCallable的第一个参数，\_2 为第二个参数，以此类推。

```
//表示绑定函数 fun 的第一，二，三个参数值为： 1 2 3
```

```
std::function<void(int, int, int)> f1 = std::bind(fun_1, 1, 2, 3);
f1(); //print: x=1,y=2,z=3
```

```
//表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别由调用 f2 的第一，二个参数指定
```

```
std::function<void(int, int, int)> f2 = std::bind(fun_1, std::placeholders::_1, std::placeholders::_2, 3);
f2(1, 2); //print: x=1,y=2,z=3
```

```
//表示绑定函数 fun 的第三个参数为 3，而fun 的第一，二个参数分别由调用 f3 的第二，一个参数指定
```

```
std::function<void(int, int, int)> f3 = std::bind(fun_1, std::placeholders::_2, std::placeholders::_1, 3);
//注意： f2 和 f3 的区别。
f3(1, 2); //print: x=2,y=1,z=3
```

从上可以看到，std::bind的返回值是可调用实体，可以直接赋给std::function。

## c++11关键字

### noexcept

noexcept告诉编译器指定某个函数不抛异常

C++中的异常处理是在运行时而不是编译时检测的。为了实现运行时检测，编译器创建额外的代码，然而这会妨碍程序优化。

一个操作或者函数不可能抛出任何异常，在以往的C++版本中常用throw()表示，在C++ 11中已经被noexcept代替。

如果在运行时，noexcept函数向外抛出了异常（如果函数内部捕捉了异常并完成处理，这种情况不算抛出异常），程序会直接终止，调用std::terminate()函数，该函数内部会调用std::abort()终止程序。

成员函数声明后面跟上throw()，表示告诉类的使用者：我的这个方法不会抛出异常，所以，在使用该方法的时候，不必把它至于 try/catch 异常处理块中。

```
void swap(Type& x, Type& y) throw() //C++11之前
{
 x.swap(y);
}
void swap(Type& x, Type& y) noexcept //C++11
{
 x.swap(y);
}
```

## override

告诉编译器要重写父类的方法（函数参数、返回类型必须相同）

## final关键字

该关键字用来修饰类，当用final修饰后，该类不允许被继承，在 C++ 11 中 final 关键字要写在类名的后面

## =default

如果一个 C++ 类没有显式地给出构造函数、析构函数、拷贝构造函数、operator = 这几类函数的实现，在需要它们时，编译器会自动生成；或者，在给出这些函数的声明时，如果没有给出其实现，编译器在链接时就会报错。=default 如果标记这类函数，编译器会给出默认实现。

=default 笔者觉得最大的作用就是，在开发中简化了那些构造函数中没有实际的初始化代码的写

法，尤其是声明和实现分别属于一个 .h 和 .cpp 文件。

## **=delete**

禁止编译器生成这些函数

## **using**

一般的using关键字我们都是用来声明当前文件的命名空间，比如标准库的命名空间std-> using namespace std;

但在c++11中，它的用处还有几个：

1. 取代typedef
2. 让父类同名函数在子类中以重载方式使用

# 实现一个引用计数功能？c++中共享指针是怎样计数的？

## 什么是引用计数？

使用一个计数器来标识当前对象被多少指针所指或者被引用的次数。当引用对象被创建或被拷贝时，引用计数要加1；当引用对象被销毁或被覆盖时，引用计数减1；当引用计数为0时，数据对象被销毁。（也是核心原理）。通俗的来讲即这块地址上每多一个指针指向他，计数加一；

因为有多指针指向了同一个内存，如果提前释放了这个内存，那其他指向这个地址的指针则会成为野指针，就是指针悬挂现象。同时如果所有的指针都不指向这个地址，这块地址还没被释放，就会被造成内存泄漏。所以采用计数的方式来判断这个地址上还有多少个指针，当最后一个指针不再指向这块内存的时候，就释放掉这块内存。

## 实现引用计数的目的

1. **右值引用**一旦一个对象通过调用new被分配出来，记录谁拥有这个对象是很重要的，因为其所有者要负责对它进行delete。但是对象所有者可以有多个，且所有权能够被传递，这就使得内存跟踪变得困难。引用计数可以跟踪对象所有权，并能够自动销毁对象。
2. 节省内存，提高程序运行效率。如何很多对象有相同的值，为这多个相同的值

存储多个副本是很浪费空间的，所以最好做法是让所有对象都共享同一个值的实现。

```

template <class T>
class Ref_count{
private:
 T* ptr; //数据对象指针
 int* count; //引用计数器指针
public:
 /*
 普通指针构造共享指针,注意这样有问题,造成二龙治水
 因为同一块内存的普通指针构建的共享指针也指的是同一块内存,所以不应该是1,应该++
 比如shared_ptr<int> s_ptr(p); s_ptr指向了这块地址, pCount = 1
 shared_ptr<int> s_ptr1 = s_ptr; s_ptr1也指向了这块地址, pCount = 2
 shared_ptr<int> s_ptr2(p); s_ptr2也指向了这块地址, 不过重新创建了引用计数, pCount1 = 1, 这样
 //所以要避免一个原生指针多次使用这个函数
 Ref_count(T* t):ptr(t),count(new int(1)){}

 ~Ref_count(){
 decrease();
 }

 //拷贝构造
 Ref_count(const Ref_count<T>& tmp){
 count = tmp->count;
 ptr = tmp->ptr
 increase();
 }

 //注意=在指针里面是指向的意思,因此说明=左边的共享指针指向了=右边的
 //因此=左边的共享指针-1, =右边的共享指针+1
 Ref_count<T>& operator=(const Ref_count& tmp){
 if(tmp != this){
 decrease();
 ptr = tmp->ptr;
 count = tmp->count;
 increase();
 }
 return *this
 }

```



```

T* operator ->() const{
 return ptr;
}

T& operator *() const{
 return *ptr;
}

void increase(){
 if(count){
 *(count)++;
 }
}

void decrease(){
 if(count){
 *(count)--;
 if(*count == 0){
 //引用计数为0的时候就删除数据对象指针和引用对象指针
 delete ptr;
 ptr = nullptr;
 delete count;
 count = nullptr;
 }
 }
}

T* get() const{
 return ptr;
}

int get_count() const{
 if(!count){
 return 0;
 }
 return *count;
}

};

```

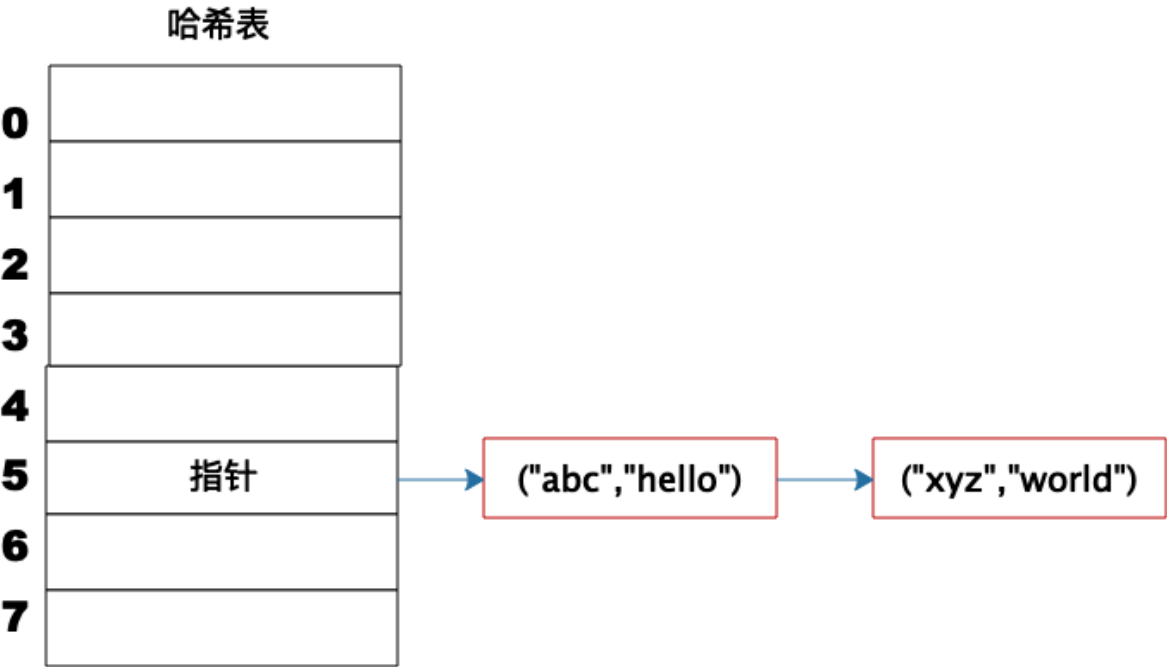
注意千万不要吧一个原生指针给多个共享指针管理，会造成错误，具体看我的注释。

# 哈希表时间复杂度为什么是O(1)?

首先要参考数组的查找，数组在内存中是一块连续的地址空间，只要知道查找数据的下标就可以快速定位到数据的内存。

那么哈希表就是利用了数组的这种特性，即hash表的物理存储其实是数组。

事实上，(“abc”,“hello”) 这样的 Key、Value 数据并不会直接存储在 Hash 表的数组中，因为数组要求存储固定数据类型，主要目的是每个数组元素中要存放固定长度的数据。所以，数组中存储的是 Key、Value 数据元素的地址指针。一旦发生 Hash 冲突，只需要将相同下标，不同 Key 的数据元素添加到这个链表就可以了。查找的时候再遍历这个链表，匹配正确的 Key。



@51CTO博客

## sizeof相关面试题

### sizeof的定义

sizeof是C语言的一种单目操作符，它并不是函数。\*\*sizeof操作符以字节形式给出了其操作数所占存储空间的大小。\*\*操作数可以是一个表达式或括在括号内的类型名。操作数所占存储空间的大小由操作数的类型决定。作用就是返回一个对象或者类型所占的内存字节数。

sizeof在头文件中typedef为unsigned int，其值在编译时即计算好了，参数可以是数组、指针、类型、对象、函数等。它的功能是：获得保证能容纳实现所建立的最大对象的字节大小。由于在编译时计算，因此sizeof不能用来返回动态分配的内存空间的大小。

数组——编译时分配的数组空间大小；

指针——存储该指针所用的空间大小（存储该指针的地址的长度，是长整型，应该为4）；

类型——该类型所占的空间大小；

对象——对象的实际占用空间大小；

函数——函数的返回类型所占的空间大小。函数的返回类型不能是void。

对了，还要注意结构体的内存对齐原则！！！这个也是考点

问：定义一个不包含任何成员变量和成员函数的空的类，对该类型sizeof，得到的结果是多少？

答：1。因为该类的实例不包含任何信息，按常理来说应该sizeof=0的。但是当我们声明该类实例时，必须在内存中占用一定的空间才行，否则你无法使用这些实例。至于占用多少内存是由编译器决定的。用了一个char为了保证空类和空类之间在内存中不会有相同的地址。c++中的struct和class本质其实没有区别，区别仅仅是默认的“权限不同”，class是private，struct是public，sizeof(class)或者sizeof(struct)是1。

问：如果在空类里面添加一个构造函数和析构函数，sizeof是多少？

答：还是一样的。因为调用构造函数和析构函数或者其他函数只需要知道函数的地址就行，函数的地址和类的实例无关。

问：如果吧析构函数或者其他函数编程虚函数呢？

答：如果类中有虚函数，会生成虚函数表，并且每个对象都会在头部添加一个指向虚函数表的指针。所以sizeof就是指针的大小，计算机内部地址总线的宽度，32位机器上是4，64位机器上是8。

问：不用sizeof如何获得int所占的字节数？

思路：设初始值为1，则循环将值左移，直到值为0，记录循环的次数，即总共的位数，再除以8（一个字节=8位），即该类型的字节长度。

```
int main()
{
 int i = 1;
 int count = 0;
 while(i)
 i = i << 1; // 一个循环，每次左移一位
 count++;
 cout << count/8 << endl; // 因为一个字节8位
 return 0;
}
```

## Volatile的作用？是否具有原子性？对编译器有什么影响？

对于volatile这个单词，权威词典有三个意思：

1. likely：可能的。这意味着被 volatile 形容的对象「有可能也有可能不」发生改变，因此我们不能对这样的对象的状态做出任何假设。
2. suddenly：突然地。这意味着被 volatile 形容的对象可能发生瞬时改变。
3. unexpectedly：不可预期地。这与 likely 相互呼应，意味着被 volatile 形容的对象可能以各种不可预期的方式和时间发生更改。

因此，volatile 其实就是告诉我们，被它修饰的对象出现任何情况都不要奇怪，我们不能对它们做任何假设。

volatile是一种类型修饰符

### 在常规程序中的volatile

```
volatile int *p = /* ... */;
int a, b;
a = *p;
b = *p;
```

如果忽略volatile关键字，上述代码只需要从内存读取一次就够了。因为从内存中读取一次之后，CPU 的寄存器中就已经有了这个值；把这个值直接复用就可以了。这样一来，编译器就会

做优化，把两次访存的操作优化成一次。但是如果加上 `volatile` 关键字后，编译器对访问该变量的代码就不再进行优化，从而可以提供稳定访问，

总结：当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后，再取变量值时，就直接从寄存器中取值。这个时候在寄存器和内存中都有我们的值，按道理来说应该是一致的，但有几种情况：

1. 当变量值在本线程里改变时，会同时把变量的新值copy到该寄存器中，以便保持一致
2. 当变量在因别的线程而改变了值，该寄存器的值不会相应改变，从而造成应用程序读取的值和实际的变量值不一致
3. 当该寄存器在因别的线程而改变了值，原变量的值不会改变，从而造成应用程序读取的值和实际的变量值不一致

`volatile`的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值

但是要知道`volatile`并不能真正解决多线程之间的问题。对于临界区的资源我们可以用锁机制来保护，对于非临界区的资源，如果使用 `volatile` 会禁止编译器优化相关变量，从而降低性能，所以也不建议依赖 `volatile` 在这种情况下做线程同步。

使用 `std::atomic<type>` 操作是原子的，同时构建了良好的内存屏障

### 多线程编程中什么情况下需要加 `volatile`？

C/C++多线程编程中不要使用`volatile`。C++11标准中明确指出解决多线程的数据竞争问题应该使用原子操作或者互斥锁。

因为C和C++中的`volatile`并不是用来解决多线程竞争问题的，而是用来修饰一些因为程序不可控因素导致变化的变量，比如访问底层硬件设备的变量，以提醒编译器不要对该变量的访问擅自进行优化。

### 是否可以和`const`一起使用？

没问题，这俩又不冲突

## extern 关键字

①为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言的进行编译，而不是C++的。主要原因是C++和C程序编译完成后在目标代码中的命名规则不同。比如C++语言在编译的时候为了解决函数的多态问题，会将函数名和参数联合起来生成一个中间的函数名称，而C语言则不会，因此会造成链接时找不到对应函数的情况，此时C函数就需要用extern "C"进行链接指定，这告诉编译器，请保持我的名称，不要给我生成用于链接的中间函数名。

②extern是C/C++语言中表明函数和全局变量作用范围（可见性）的关键字，该关键字告诉编译器，其声明的函数和变量可以在本模块或其它模块中使用。

## auto类型推断的原理

### [参考链接](#)

auto使用的是**模板实参推断**（Template Argument Deduction）的机制。

从函数实参来确定模板实参的过程被称为模板实参推断(template argument deduction)；

在使用类模板创建对象时，程序员需要显式的指明实参（也就是具体的类型），而对于函数模板，调用函数时可以不显式地指明实参，编译器会根据传入的实参类型来自动推导出模板实参类型。

看代码：

```

template<typename Container>
void useContainer(const Container& container)
{
 auto pos = container.begin();
 while (pos != container.end())
 {
 auto& element = *pos++;
 ... // 对元素进行操作
 }
}

// auto pos = container.begin()的推断等价于如下调用模板的推断
template<typename T>
void deducePos(T pos);
deducePos(container.begin());

```

auto被一个虚构的模板类型参数T替代，然后进行推断，即相当于把变量设为一个函数参数，将其传递给模板并推断为实参，auto相当于利用了其中进行的实参推断，承担了模板参数T的作用

## C++程序优化的方法

- 空间足够时，可以将经常需要读取的资源，缓存在内存中。
- 尽量减少大内存对象的构造与析构，考虑缓存暂时不用的对象，等待后续继续使用。
- 尽量使用C++11的右值语义，减少临时对象的构造。
- 简单的功能函数可以使用内联。少用继承，多用组合，尽量减少继承层级。
- 在循环遍历时，优化判断条件，减少循环次数。
- 优化线程或进程的同步方式，能用原子操作的就不用锁。能应用层同步的就不用内核对象同步。
- 优化堆内存的使用，如果有内存频繁的申请与释放，可以考虑内存池。
- 优化线程的使用，节省系统资源与切换造成的性能损耗，线程使用频繁的可以考虑线程池。
- 尽量使用事件通知，谨慎使用轮循或者sleep函数。
- 界面开发中，耗时的业务代码不要放在UI线程中执行，使用单独的线程去异步处理耗时业务，提高界面响应速度。
- 经常重构、优化代码结构。优化算法或者架构，从设计层面进行性能的优化。

## void\*（泛型指针）

指针是对内存区域的抽象。指针变量中存放着目标对象的内存地址，而与指针相复合的类型，则说明了相应内存区域中的内容具有哪些属性，以及能做什么事情。也就是说，在内存空间某块区域中的内容，原本可以是不可解读的；但是，如果有一个描述这块内存区域的指针存在，我们就能找到它（地址的作用），并且合理地使用它（类型的作用）。void\* 只有其中一半的作用。因为没有明确与指针相复合的类型，所以不能解引用，也不能使用基于类型之上（sizeof(T)）的指针运算。

## C++11 RVO/NRVO机制

RVO (return value optimization) 和NRVO (named return value optimization) 是C++在处理“返回一个class object的函数”时常用的优化技术，主要作用就是消除临时对象的构造和析构成本。

RVO就是我经常自己写的那种，就是如果函数返回值有返回值，编译器会优化成传入一个引用，然后直接将值放在引用中：

```
int get_max(){
 int a;
 return a;
}
//改成
int get_max(int& a){
 int a = max;
}
```

[参考链接](#)

## C++从源代码到可执行程序的流程是怎样的

源程序（source code）→预处理器（preprocessor）→编译器（compiler）→汇编程序（assembler）→目标程序（object code）→连接器（链接器，Linker）→可执行程序（executables）

- **预处理**

生成test.i文件，做一下处理：



- i. 展开宏定义
- ii. 处理`#if`, `#end`, `#ifndef`
- iii. 处理`#include`指令, 把.h文件插入对应位置
- iv. 删除注释

- **编译**

编译过程所进行的是对预处理后的文件进行语法分析, 词法分析, 语义分析, 符号汇总, 然后生成汇编代码文件`test.s`

词法分析阶段是编译过程的第一个阶段。这个阶段的任务是从左到右一个字符一个字符地读入源程序, 即对构成源程序的字符流进行扫描然后根据构词规则识别单词(也称单词符号或符号)。词法分析程序实现这个任务。词法分析程序可以使用lex等工具自动生成。

语法分析是编译过程的一个逻辑阶段。语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语, 如“程序”, “语句”, “表达式”等等。语法分析程序判断源程序在结构上是否正确。源程序的结构由上下文无关文法描述。

语义分析是编译过程的一个逻辑阶段。语义分析的任务是对结构上正确的源程序进行上下文有关性质的审查, 进行类型审查

生成`test.s`文件

- **汇编**

将汇编代码转成二进制文件, 二进制文件就可以让机器来读取。每一条汇编语句都会产生一句机器语言。生成`test.o`文件

- **链接**

将二进制文件变成一个可执行文件。链接会涉及到动态链接和静态链接。

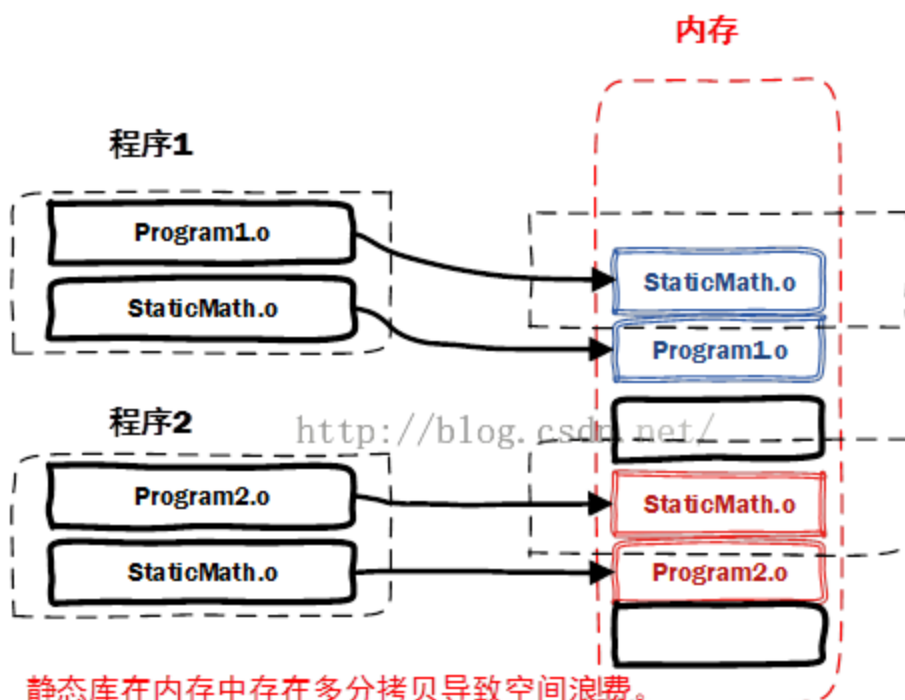
## C++的静态链接和动态链接讲一下

### 静态链接

在程序的链接阶段, 将上一阶段生成的`test.o`文件与库函数合并生成可执行文件。

这样做好处就是以后的代码和库函数无关了, 可以随便移植。

坏处是①静态链接的文件体积太大。②如果库函数更新的话需要重新链接。③在内存中会存在多分拷贝, 因为每个程序都会存在一份库函数, 如下图:



静态库在内存中存在多份拷贝导致空间浪费。  
假如，静态库占用1M内存，有2000个这样的程序，将占用近2GB的空间~~~~~

52php.cnblogs.com

静态库的生成使用如下命令：

```
ar -crv libadd.a add.o
```

## 动态链接

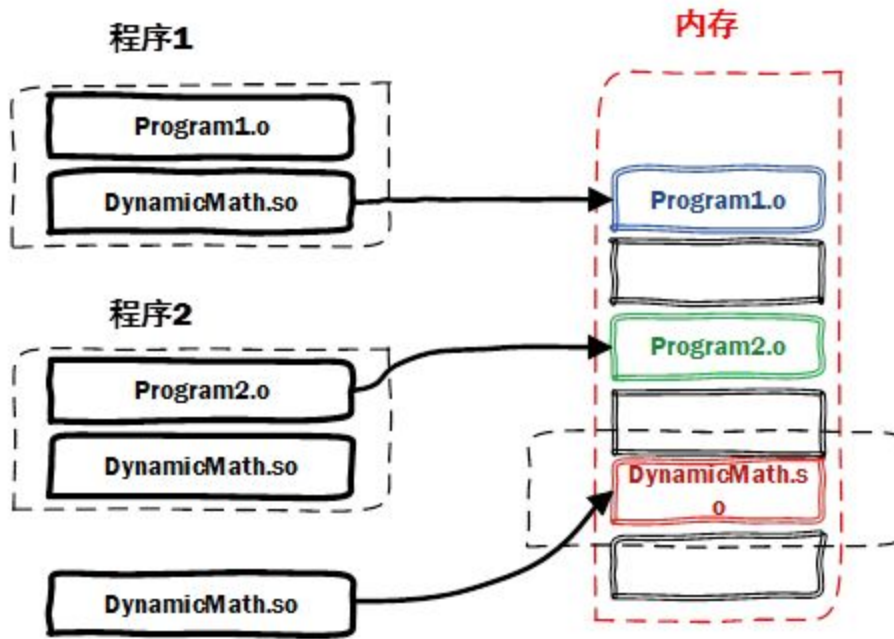
显然程序员都比较懒，比如我只用一个功能，我把所有静态库链接进去肯定是不科学的，而且每次库省级都要重新编译链接

动态库在程序编译时并不会被连接到目标代码中，而是在程序运行是才被载入。不同的应用程序如果

调用相同的库，那么在内存里只需要有一份该共享库的实例，规避了空间浪费问题。动态库在程序运行是才被载入，也解决了静态库对程序的更新、部署和发布页会带来麻烦。用户只需要更新动态库即可，增量更新。

我们把静态链接与动态链接做一个这样子的比喻：把链接过程看做我们平时学习时做笔记的过程。我们平时学习时准备一本笔记本专门记录我们的学习笔记，比如在某本书的某一页上看到一个很好很有用的知识，这时候我们有两种方法记录在我们的笔记本上。一种是直接把那一页的内容全部抄写一遍到笔记本上（静态链接）；另一种是我们在笔记本上做个简单的记录（动态链接），比如写上：xxx知识点在《xxx》的xxx页。从这两种方法中我们可以很清楚地知道两种方式的特点。第一种方式的优点就是我们在复习的时候就很方便，不用翻阅其它书籍了，但是缺点也很明显，就是占用笔记本的空间很多，这种方法很快就把我们的笔记本给写满了。第二种方式的优点就是很省空间，缺点就是每当我们复习的时候，手头上必须备着相关的参考书籍，比如我们去教室复习的时候，就得背着一大摞书去复习，这样我们复习的效率可能就没有那么高了。

## 动态编译和静态编译



动态库在内存中只存在一份拷贝，避免了静态库浪费空间的问题。

静态编译，编译器在编译可执行文件时，把需要用到的对应链接库中的部分提出来，连接到可执行文件中，使可执行文件在运行时不需要依赖于动态链接库。

动态编译，可执行文件需要附带一个动态链接库，在执行的时候，需要调用其对应动态链接库的命令。优点是缩小了执行文件本身的体积，另一方面加快了编译速度，节省系统资源。

缺点是需附着一个动态链接库，当这个库很大的时候很麻烦；其次是如果该环境没有安装对应的库，动态编译的可执行文件不能运行。

## 动态联编和静态联编

在c++中联编指的是计算机程序不同部分彼此关联起来的过程。

静态联编指联编工作在编译期完成，这种联编过程是在程序运行之前完成的，又叫做早期联编。如果要实现静态联编，就必须在编译阶段确定程序中的操作调用与执行该操作代码的关系。静态联编对函数的选择是基于执行对象的指针或引用的类型，优点是效率高，但灵活性太差。

动态联编指的是程序运行的时候动态的进行。实际上是虚函数的实现过程，又叫做晚期联编。动态联编对成员函数的选择是基于对象类型的，针对不同对象类型做出不同编译结果。

c++一般情况下的联编是静态联编，当涉及到虚函数的时候就会使用动态联编。

## 指针没有初始化会怎么样

没有初始化的指针叫做野指针

指针未初始化能通过编译，但是运行的时候可能报错也可能不报错。因为你如果没使用这个指针，那倒是无所谓。但是如果你使用了，可能会出问题。因为这个指针由于没有初始化，可能会指向任何内存空间，完全随机的，有两种情况：

1. 指向的地址是系统使用的内存，用户程序不能使用，如果用户程序使用则会报错
2. 指向的不是系统的内存，不报错。但是如果这个指针指向了你之前程序使用过的内存，则你在个指针赋值，就会修改之前的内存上的数据，也会出问题。

指针变量设置为nullptr表明它不指向任何内容,这样引用她也不会出现上面的问题。

## 空指针到底是什么？

由系统保证空指针不指向任何实际的对象或函数，也就是说，任何对象或者函数的地址都不可能是空指针，空指针与任何对象或函数的指针值都不相等。空指针表示“未分配”或者“尚未指向任何地方”。它与未初始化的指针有所不同，空指针可以确保不指向任何对象或函数，而未初始化指针可能指向任何地方。

## 野指针和指针悬挂

野指针(wild pointer)指的是未经初始化的指针

悬挂指针(dangling pointer)指的是已经销毁的对象或已经回收的地址

## 基础变量没有初始化怎么样

不怎么样，不使用他就行。

建议变量在构造函数时均初始化完成，避免不必要的问题。比如变量char buf[2000]，把字符串"hello world"拷贝进入buf，恰好EOF("\0")没有被复制，由于没有使用memset初始化变量，buf空间时随机值,那么调用print，strcmp之类的操作时，有可能会因为没有EOF,导致访问到非法地址，然后程序异常。

## c/c++参数入栈顺序和参数计算顺序

c/c++中规定了函数参数的压栈顺序是从右至左

举个例子：

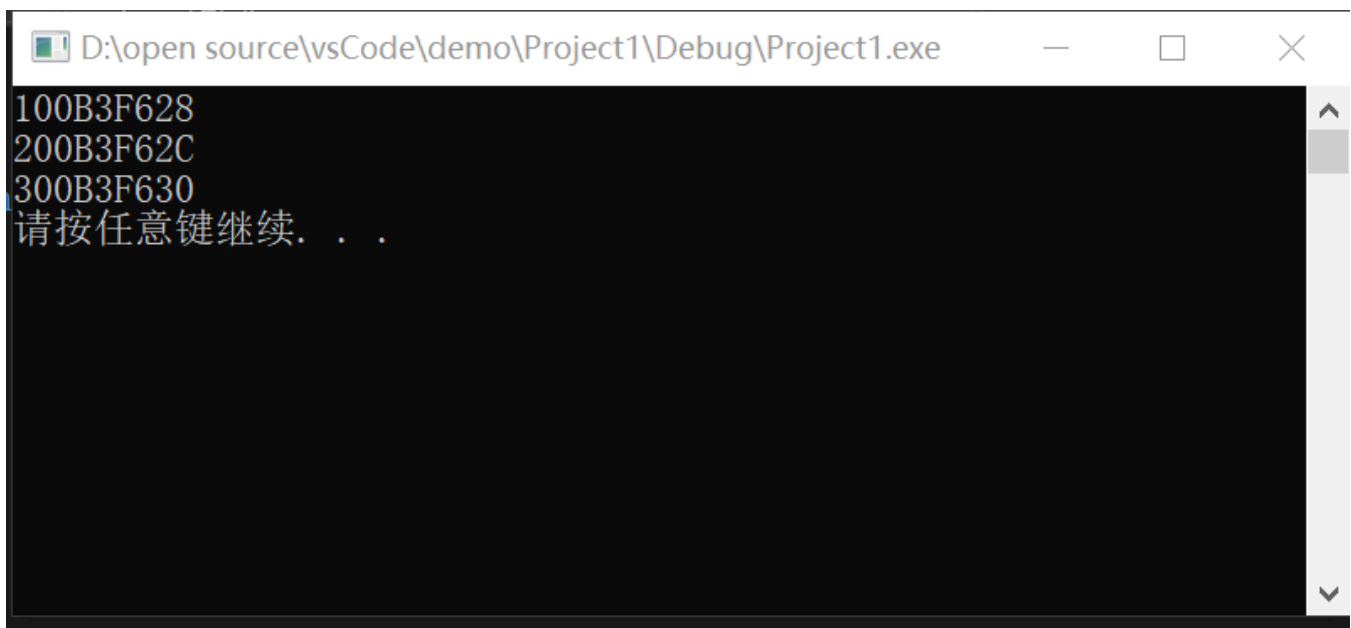
```

void fun(int x, int y, int z)
{
 cout << x << &x << endl;
 cout << y << &y << endl;
 cout << z << &z << endl;
}

int main(int argc, char *argv[])
{
 fun(1, 2, 3);
 system("pause");
 return 0;
}

```

输出如下：



```

D:\open source\vsCode\demo\Project1\Debug\Project1.exe
100B3F628
200B3F62C
300B3F630
请按任意键继续. . .

```

我们知道先入栈的占高地址，从结果看出入栈的顺序依次为 `z->y->x`，即压栈顺序从右至左。

**why?**

本质上是因为支持可变长参数导致的。如果是从左至右的入站方式，最左边的参数就会被压在栈底，栈顶指针指向的是最右边的长度参数，由于可变长度，编译器无法找到最左边的参数吧。但是如果从右至左入栈，最左边的参数就在栈顶。

**printf的原理**

C/C++的函数参数是通过压入堆栈的方式来给函数传参数的。最先压入的参数最后出来，在计算机的内存中，数据有2块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到。因为它就在堆栈指针的上方。printf的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可算出数据需要的堆栈指针的偏移了。

## 模板元编程

泛型编程是一种编程风格。

在c中不同参数相同功能函数名不同，现在函数名相同，但是可能参数类型不同也要重新写函数。模板出现就是提高了程序的复用性，提高效率。我觉得，编程能力本质上就是一个量变引起质变的过程，当刚上手的时候肯定是根据具体的数据类型来组织代码。但是随着越来越熟了，就会偷懒就会方便，所以你想用一种广泛的表达去取代具体数据类型，这在c++中就叫做模板编程。

c++模板主要分成两大类：函数模板和类模板。

模板的格式是 `template<template T>` 或者 `template<class T>`。template是一个声明模板的关键词，表示声明一个模板关键字class不能省略，如果类型形参多余一个，每个形参前都要加class <类型 形参表>可以包含基本数据类型可以包含类类型。

## 模板底层怎么实现？

编译器并不是把函数模板处理成能够处理任意类的函数；

编译器从函数模板通过具体类型产生不同的函数；

编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

# 函数模板

函数模板分为成员函数模板和普通函数模板。

**调用方式（重要）：**

1. 自动类型推导，隐式调用

```
mySwap(a, b)
```

2. 显式指定类型

```
mySwap<int>(a, b)
```

**模板注意事项：**

1. 自动类型推导，必须推导出一致的数据类型T才可以使用。也就是说参数类型和你模板定义的得一致才行。
2. 模板必须要确定出T的类型

**普通函数与模板函数的区别：**

1. 普通函数调用时可以发生自动类型转换（隐式类型转换）
2. 如果使用函数模板，自动类型推导的话，则不会发生隐式转换
3. 如果使用函数模板，显式指定类型，则可以发生隐式转换

**普通函数与函数模板的调用规则：**

1. 优先调用普通函数
2. 可以使用空模板参数来强制调用模板函数

```
myPrint<>(arg1, arg2,...)
```

3. 函数模板也可以重载
4. 如果函数模板可以产生更好的匹配，优先调用函数模板

## 类模板

为什么成员函数模板不能是虚函数(virtual)？

这是因为c++ 编译器在解析一个类的时候就要确定虚函数表的大小，如果允许一个虚函数是模板函数，那么compiler就需要在parse这个类之前扫描所有的代码，找出这个模板成员函数的调用（实例化），然后才能确定vtable的大小，而显然这是不可行的，除非改变当前compiler的工作机制。因为类模板中的成员函数在调用的时候才会创建

类模板没有自动类型推导的使用方式，只有显式指定参数类型

### 类模板中成员函数创建时间：

1. 普通类中的成员函数在编译的时候就创建
2. 类模板中的成员函数在调用的时候才会创建

### 类模板对象做函数参数：

就是类模板实例化出的对象，作为参数的形式传入函数

1. 指定传入的类型 —— 直接显示对象的数据类型

```
void print(Person<string, int>&p);
```

2. 参数模板化 —— 将对象中的参数变为模板进行传递

```
template<class T1, class T2>
void print(Person<T1, T2>& p);
```

3. 整个类模板化 —— 将整个对象类型模板化进行传递

```
template<class T>
void print(T& p);
```

### 类模板与继承：

1. 当派生类继承基类的一个类模板时，子类在声明时，要指定出分类中的T类型



```

template<class T>
class Father{
 T m;
};

//报错
class Son: public Father{

};

//正确
class Son: public Father<int>{

};

```

因为子类要继承父类中的成员变量，但是模板没有指定内存大小，所以是不确定的，而不确定性是c++所嗤之以鼻的。因此继承的时候得指定要继承模板的数据类型才行。

但是这样不灵活，父类中的类型就被定死了，有违背c++灵活编程，所以这种方法不太实用

2. 如果不指定，编译器无法给子类非配内存
3. 如果要灵活的话，子类也需变为类模板

```

template<class T1, class T2>
class Son: public Father<T2>{
 T1 obj;
};

```

## 类模板成员函数的类外实现

```
//构造函数类外实现
template<class T1, class T2>
Person<T1, T2>::Person(T1 name, T2 age){}

//成员函数类外实现
template<class T1, class T2>
void Person<T1, T2>::show(){}

```

## 模板和实现可不可以不写在一个文件里面？为什么？

不可以，实习的时候出错过。编译能通过，连接不能通过。如下代码：

```
// add.h
template <typename T>
T add(const T &a, const T &b);

// add.cpp
#include "add.h"
template <typename T>
T add(const T &a, const T &b)
{
 return a + b;
}

// main.cpp
#include "add.h"
int main()
{
 int i = add(1, 1);
 return 0;
}

$ gcc -c main.cpp # main.cpp编译通过
$ gcc -c add.cpp # add.cpp编译通过
$ gcc -o main main.o add.o # 链接失败!
main.o: In function `main':
main.cpp:(.text+0x34): undefined reference to `int add<int>(int const&, int const&)'
collect2: error: ld returned 1 exit status

```

编译器面对巨量代码的时候，也是以一个一个的.cpp/.c文件作为基本单元，根据代码的include包含找到声明，翻译代码产生.o文件。注意他们每个cpp/c文件都是相互独立完成自己工作的，对于缺少的部分，如果妥善声明，会留待链接过程的时候产生引用关系。由于我们在编译main.cpp的时候只是找到了.h文件，即模板的声明，但是没有模板函数的具体实现，因此就没有办法实例化add函数。在add.cpp中有模板函数的使用，可以实例化，但是cpp编译单元中并没有人使用该函数，因此该cpp不会产生任何可执行代码，编译add.cpp时没有生成可执行代码。因此在链接的时候会报错，因为没有可执行代码

《C++编程思想》第15章(第300页)说明了原因：模板定义很特殊。由template<...>处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

## 模板类和模板函数的区别是什么？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加 <T>，而函数模板不必

## 模板和继承，这个区别是什么？

第一点：

模板可以生成一组类或者函数，这些类或函数的实现都是一样的

继承是事物之间的一种关系，从父类到子类实际上就是从普遍到特殊、从共性到特性

第二点：

模板和继承都是多态性的体现，继承是运行时的多态性，模板是编译时的多态性。

第三点：

继承是数据的复制、模版是代码的复制。

模板函数在编译完成之后，会生成对应参数数类型的函数；

继承是对虚表、数据的复制

# mutable关键字

mutable : 可变的

首先想到的是在lambda表达式中有这个东西，表示如果是值传递的，可以修改，不加这个mutable属性不能修改，表示传递过来的是常量。虽然在匿名函数内部改变了变量的值，但是在外部还是原来的值

除了lambda表达式中的，就只有类中的了

mutable 在类中只能够修饰非静态数据成员，用来修饰一个 const 示例的部分可变的数据成员的。如下代码：

```
struct Test
{
 int a;
 mutable int b;
};

const struct Test test = {1,2};
test.a = 10; # 编译错误
test.b = 20; # 允许访问
```

## c++RITT机制

概念：

RTTI(Run Time Type Identification)即通过运行时类型识别，C++引入这个机制是为了让程序在运行时能根据基类的指针或引用来获得该指针或引用所指的对象的实际类型。但是现在RTTI的类型识别已经不限于此了，它还能通过typeid操作符识别出所有的基本类型（int，指针等）的变量对应的类型。

和很多其他语言一样，C++是一种静态类型语言。其数据类型是在编译期就确定的，不能在运行时更改。然而由于面向对象程序设计中多态性的要求，C++中的指针或引用(Reference)本身的类型，可能与它实际代表(指向或引用)的类型并不一致。有时我们需要将一个多态指针转换为其实际指向对象的类型，就需要知道运行时的类型信息，这就产生了运行时类型识别的要求。C++要想获得运行时类型信息，只能通过RTTI机制，并且C++最终生成的代码是直接和机器相关

的。

**如何实现：**

C++通过以下的两个操作提供RTTI：

(1) typeid运算符，该运算符返回其表达式或类型名的实际类型。返回指针和引用所指的 actual 类型；

(2) dynamic\_cast运算符，该运算符将基类的指针或引用安全地转换为派生类类型的指针或引用。

我们知道C++的多态性（运行时）是由虚函数实现的，对于多态性的对象，无法在程序编译阶段确定对象的类型。当类中含有虚函数时，其基类的指针就可以指向任何派生类的对象，这时就有可能不知道基类指针到底指向的是哪个对象的情况，类型的确定要在运行时利用运行时类型标识做出。为了获得一个对象的类型可以使用typeid函数，该函数返回一个对type\_info类对象的引用，要使用typeid必须使用头文件 `<typeinfo>`，因为typeid是一个返回类型为type\_info的引用的函数

## 虚继承

**为什么要引入虚拟继承**

虚拟继承是多重继承中特有的概念。虚拟基类是为解决多重继承而出现的。如:类D继承自类B1、B2，而类B1、B2都继承自类A，因此在类D中两次出现类A中的变量和函数。实现的代码如下：

```
class A
```

```
class B1:public virtual A;
```

```
class B2:public virtual A;
```

```
class D:public B1,public B2;
```

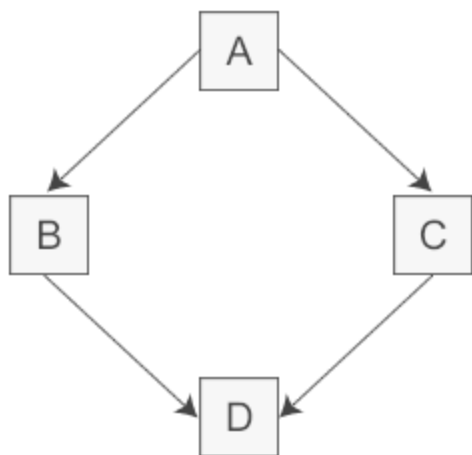


图1：菱形继承

上述就是菱形继承会产生两个问题。首先是空间问题，不节省空间。类B中有类A的数据，类C中也有。其次是二义性，歧义问题。当类D调用类A中的数据时候，到底是走类B的还是走类D的？如下：

```

void seta(int a){ B::m_a = a; } // B类中的m_a
void seta(int a){ C::m_a = a; } // C类中的m_a

```

虚拟继承在一般的应用中很少用到，所以也往往被忽视，这也主要是在C++中，多重继承是不推荐的，也并不常用，而一旦离开了多重继承，虚拟继承就完全失去了存在的

必要因为这样只会降低效率和占用更多的空间。

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

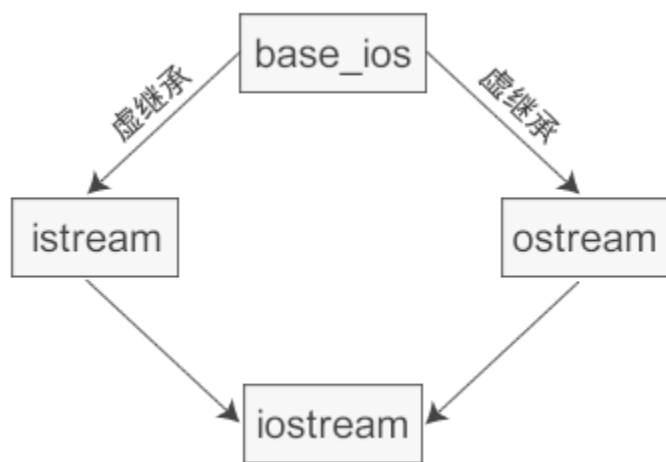


图3：虚继承在C++标准库中的实际应用

### 引入虚继承和直接继承会有什么区别呢

由于有了间接性和共享性两个特征，所以决定了虚继承体系下的对象在访问时必然会在时间和空间上与一般情况有较大不同。

时间：在通过继承类对象访问虚基类对象中的成员（包括数据成员和函数成员）时，都必须通过某种间接引用来完成，这样会增加引用寻址时间（就和虚函数一样），其实就是调整this指针以指向虚基类对象，只不过这个调整是运行时间接完成的。

空间：由于共享所以不必要在对象内存中保存多份虚基类子对象的拷贝，这样较之多继承节省空间。虚拟继承与普通继承不同的是，虚拟继承可以防止出现diamond继承时，一个派生类中同时出现了两个基类的子对象。也就是说，为了保证这一点，在虚拟继承情况下，基类子对象的布局是不同于普通继承的。因此，它需要多出一个指向基类子对象的指针。

在虚继承中，虚基类是由最终的派生类初始化的，换句话说，最终派生类的构造函数必须要调用

虚基类的构造函数。对最终的派生类来说，虚基类是间接基类，而不是直接基类。这跟普通继承不同，在普通继承中，派生类构造函数中只能调用直接基类的构造函数，不能调用间接基类的。比如 在最终派生类 D 的构造函数中，除了调用 B 和 C 的构造函数，还调用了 A 的构造函数，这说明 D 不但要负责初始化直接基类 B 和 C，还要负责初始化间接基类 A。而在以往普通继承中，派生类的构造函数只负责初始化它的直接基类，再由直接基类的构造函数初始化间接基类，用户尝试调用间接基类的构造函数将导致错误。

为什么这么做呢？因为现在采用了虚继承，虚基类 A 在最终派生类 D 中只保留了一份成员变量 m\_a，如果由 B 和 C 初始化 m\_a，那么 B 和 C 在调用 A 的构造函数时很有可能给出不同的实参，这个时候编译器就会犯迷糊，不知道使用哪个实参初始化 m\_a。为了避免出现这种矛盾的情况，C++ 干脆规定必须由最终的派生类 D 来初始化虚基类 A，直接派生类 B 和 C 对 A 的构造函数的调用是无效的。在第 50 行代码中，调用 B 的构造函数时试图将 m\_a 初始化为 90，调用 C 的构造函数时试图将 m\_a 初始化为 100，但是输出结果有力地证明了这些都是无效的，m\_a 最终被初始化为 50，这正是在 D 中直接调用 A 的构造函数的结果。

### 笔试，面试中常考的C++虚拟继承的知识点

| 第一种情况:                                                                                                 | 第二种情况:                                                                                          | 第三种情况                                                                                                              | 第四种情况                                                                                                  |
|--------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <pre>class a {     virtual void func(); }; class b:public virtual a {     virtual void foo(); };</pre> | <pre>class a {     virtual void func(); }; class b :public a {     virtual void foo(); };</pre> | <pre>class a {     virtual void func();     char x; }; class b:public virtual a {     virtual void foo(); };</pre> | <pre>class a {     virtual void func(); }; class b:public virtual a {     virtual void foo(); };</pre> |

如果对这四种情况分别求sizeof(a) , sizeof(b)。结果是什么样的呢？下面是输出结果：

第一种：4， 12 第二种：4， 4 第三种：8， 16 第四种：8， 8

每个存在虚函数的类都要有一个4字节的指针指向自己的虚函数表，所以每种情况的类a所占的字节数应该是没有什么问题的。同时虚继承要有这样的一个指针vptr\_b\_a，这个指针叫虚类指针，也是四个字节；还要包括类a的字节数，所以类b的字节数就求出来了。而“第二种”和“第四种”情况则不包括vptr\_b\_a这个指针，这回应该木有问题了吧。

# c++出现的内存问题

c++中内存问题大致有这么几个方面：

1. 缓冲区溢出。一般来说vector或者string都很智能帮我们管理缓冲区，所以很多时候不会出现
2. 指针悬挂/野指针，用shared\_ptr和weak\_ptr就可以解决
3. 重复释放，用boost的scoped\_ptr或者仔细检查，只在对象析构的时候释放一次
4. 内存泄漏，也可以用scoped\_ptr，这个比其他的来说还算正常，至少用一段时间内感觉不出来
5. new[]和delete不配对，moduo书中给的方案是把new[]统统换成vector就行
6. 内存碎片，这个就非常深邃了

## C++编译期多态与运行期多态

### 编译期多态

又叫做静态多态。指的是c++的泛型编程中（引申出泛型编程）模板的具现化与函数的重载解析。

编译期多态的类之间没有继承关系，约束他们的是相似的接口。在编译期间，编译器推断出模板参数，因此确定调用函数是哪个具体类型的接口。不同的推断结果调用不同的函数，这就是编译器多态。这类似于重载函数在编译器进行推导，以确定哪一个函数被调用。

### 运行期多态

又叫做动态多态。指的是c++面向对象编程中的三大特性之一。（引申出虚函数）

运行期多态要归结于类的继承思想。对于有相关功能的对象集合，我们总希望能够抽象出它们共有的功能集合，在基类中将这些功能声明为虚接口（虚函数），然后由子类继承基类去重写这些虚接口，以实现子类特有的具体功能。

运行期多态的实现依赖于虚函数机制。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。

### 运行期多态与编译期多态优缺点分析



- 运行期多态优点
  - i. 灵活
  - ii. 方便程序耦合同时减少内聚
  - iii. 能够处理异质对象集合（一个动物园有一堆动物，每一种动物都是一个对象）
- 运行期多态缺点
  - i. 运行期间进行虚函数绑定，提高了程序运行开销。
  - ii. 庞大的类继承层次，对接口的修改易影响类继承层次。
  - iii. 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。
- 编译期多态优点
  - i. 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。
  - ii. 在编译器完成多态，提高运行期效率。
  - iii. 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。
- 编译期多态缺点
  - i. 程序可读性降低，代码调试带来困难。
  - ii. 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。
  - iii. 无法处理异质对象集合。

## C++ 中的指针参数传递和引用参数传递

指针参数传递本质上是值传递，它所传递的是一个地址值。值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。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的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。引用参数传递过程中，被调函数的形式参数也作为局部变量在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（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问 主调函数中的实参变量（根据别名找到主调函数中的本体）。因此，被调函数对形参的任何操作都会影响主调函数 中的实参变量。

引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理 都会通过一个间接寻址的方式操作到主调函数中的相关变量。而对于指针传递的参数，如果改变被调函数中的指针 地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就 得使用指向指针的指针或者指针引

用。

从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

## 简单说一下函数指针

**\*\*定义：**函数指针首先是一个指针，其次这个指针指向函数的入口地址，函数指针本身就是一个指针变量。在编译的时候，每个函数都有一个入口地址，该入口地址就是函数指针所指向的地址，有了指向函数的指针后，可以用该指针变量调用函数。

**\*\*用途：**调用该函数，或者做函数的参数，比如回调函数。

# 如何让函数在 main 函数执行前执行？

```
//第一种：gcc扩展，标记这个函数应当在main函数之前执行。
//同样有一个__attribute__((destructor))，标记函数应当在程序结束之前（main结束之后，或者调用了exit后）执行；
__attribute__((constructor))void before() {
 printf("before main 1\n");
}
//第二种：全局和static变量的初始化在程序初始阶段，先于 main 函数的执行
int test1(){
 cout << "before main 2" << endl;
 return 1;
}
static int i = test1();

// 第三种：知乎大牛 Milo Yip 的回答利用 lambda 表达式
int a = []() {
 cout << "before main 3" << endl;
 return 0;
}();

int main(int argc, char** argv) {
 cout << "main function" <<endl;
 return 0;
}

//输出
before main 1
before main 2
before main 3
main function
```

## i++和++i的区别

前置加加不会产生临时对象，后置加加必须产生临时对象，临时对象会导致效率降低

```
//++i
int& int::operator++ (){
 *this +=1;
 return *this;
}

//i++
const int int::operator (int) {
 int oldValue = *this;
 ++ (*this) ;
 return oldValue;
}
```

## 变量声明和定义区别？

- 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间
- 定义要在定义的地方为其分配存储空间。

相同变量可以在多处声明（外部变量extern），但只能在一处定义

## C++异常处理的方法

### 1、try、throw和catch关键字

C++中的异常处理机制主要使用**try**、**throw**和**catch**三个关键字。就是程序中需要throw一个类型的异常，然后catch住这个类型的异常进行处理。

程序的执行流程是先执行try包裹的语句块，如果执行过程中没有异常发生，则不会进入任何catch包裹的语句块，如果发生异常，则使用throw进行异常抛出，再由catch进行捕获，throw可以抛出各种数据类型的信息，代码中使用的是数字，也可以自定义异常class。

### 2、函数的异常声明列表

有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：

```
int fun() throw(int,double,A,B,C){...};
```

这种写法表明函数可能会抛出int,double型或者A、B、C三种类型的异常，如果throw中为空，表明不会抛出任何异常，如果没有throw则可能抛出任何异常

### 3、C++标准异常类 exception

exception 类位于 `<exception>` 头文件中

下表是对层次结构中出现的每个异常的说明：

| 异常                    | 描述                                                    |
|-----------------------|-------------------------------------------------------|
| <b>std::exception</b> | 该异常是所有标准 C++ 异常的父类。                                   |
| std::bad_alloc        | 该异常可以通过 new 抛出。                                       |
| std::bad_cast         | 该异常可以通过 dynamic_cast 抛出。                              |
| std::bad_exception    | 这在处理 C++ 程序中无法预期的异常时非常有用。                             |
| std::bad_typeid       | 该异常可以通过 typeid 抛出。                                    |
| std::logic_error      | 理论上可以通过读取代码来检测到的异常。                                   |
| std::domain_error     | 当使用了一个无效的数学域时，会抛出该异常。                                 |
| std::invalid_argument | 当使用了无效的参数时，会抛出该异常。                                    |
| std::length_error     | 当创建了太长的 std::string 时，会抛出该异常。                         |
| std::out_of_range     | 该异常可以通过方法抛出，例如 std::vector 和 std::bitset<>::operator。 |
| std::runtime_error    | 理论上不可以通过读取代码来检测到的异常。                                  |
| std::overflow_error   | 当发生数学上溢时，会抛出该异常。                                      |
| std::range_error      | 当尝试存储超出范围的值时，会抛出该异常。                                  |
| std::underflow_error  | 当发生数学下溢时，会抛出该异常。                                      |

### 4、自定义异常类

c++网络库的源码中有

## C++函数调用的压栈过程

当函数从入口函数main函数开始执行时，编译器会将我们操作系统的运行状态，main函数的返回地址、main的参数、main函数中的变量、进行依次压栈；

当main函数开始调用func()函数时，编译器此时会将main函数的运行状态进行压栈，再将func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈；当func()调用f()的时候，编译器此时会将func()函数的运行状态进行压栈，再将其返回地址、f()函数的参数从右到左、f()定义变量依次压栈

## codedump

coredump是程序由于异常或者bug在运行时异常退出或者终止，在一定的条件下生成的一个叫做core的文件，这个core文件会记录程序在运行时的内存，寄存器状态，内存指针和函数堆栈信息等等。对这个文件进行分析可以定位到程序异常的时候对应的堆栈调用信息。

## 怎样判断两个浮点数是否相等？

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相等！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与0的比较也应该注意。与浮点数的表示方式有关

## C++为什么提供move函数？

我想说一下一个我的个人经历

有一段代码，作用是把数据库表保存到XML文件。这个转换的过程，有个中间容器，大概是这样：

```
std::map<string, std::vector<int>>> mapTable;
```

可以理解为map的key是数据表的列名，std::vector是那列数据（一行一行的）。

我之前是这么填充的：

```
std::vector<std::variant> vecRow;
for(){
 vecRow.push_back(...);
}
mapTable["列名1"] = vecRow;
```

codereview的时候我的mentor就和我讲了这个事情，本质上上述代码，把vecRow中的所有元素都复制了以便然后放到mapTable中，白白的重新创建了一遍所有行数据，又把不再需要的vecRow释放掉了。这样就很蠢。

改进：当我们知道vecRow生命（作用域后），我们可以利用这个vecRow，在std::move之前，还是有办法的，创建vecRow的时候就让它成为mapTable里某列的引用，如下：

```
std::vector<std::variant> &vecRow = mapTable["列名1"];
for(){
 vecRow.push_back(...);
}
```

但是考虑到这样的话会改动别的代码，所以用谁提的std::move是最好的

```
mapTable["列名1"] = std::move(vecRow);
```

就这么一点点改动，就能让vecRow里的东西放进mapTable里，又没避免大规模创建、析构对象。执行完上面的函数，应该会发现vecRow空了。

## 总结

其实编译器已经在力所能及的优化它能够优化的东西了，但是编译器的优化不是万能的。有时候某个变量的生命周期编译器不可预见，但是我们自己是可以知道的，因此对于这些生命周期很短的变量我们为了节省效率就可以使用move函数。举个例子：比如黄金交易，张三买了李四的黄金，就应该把黄金从李四家移动到张三家里。但如果黄金量很大，移动的成本就会非常高。另一种方式就是大家的黄金都存在银行里，张三买李四的黄金，无非就是账户里的黄金数发生个变化，实体黄金不移动，这样效率就高很多。至于“为什么管理机构（编译器）不优化全世界的黄金交易为纸上黄金交易？”，那是因为真的有人需要搬黄金回家用啊

+++

# STL

## STL介绍

Standard Template Library，标准模板库，是C++的标准库之一，一套基于模板的容器类库，还包括许多常用的算法，提高了程序开发效率和复用性。

**STL包含6大部件：容器、迭代器、算法、仿函数、适配器和空间配置器。**

- 容器：容纳一组元素的对象，提供各种数据结构。
- 迭代器：提供一种访问容器中每个元素的方法，从实现的角度来说，迭代器是一种将 `operator*`，`operator->`，`operator++` 等指针操作赋予 重载的类模板。
- 仿函数：一个行为类似函数的对象，调用它就像调用函数一样，重载了 `operator()` 的类或者类模板。
- 算法：包括查找算法、排序算法等等。
- 适配器：用来修饰容器等，比如queue和stack，底层借助了deque。
- 空间配置器：负责空间配置和管理，是一个实现了动态空间配置，空间管理，空间释放的类模板。

## 各个容器特点总结

容器分类：

### 1. 序列容器 sequence containers

- array
- vector
- deque
- list
- forward-list

### 2. 关联容器 associative containers

(红黑树实现)

- set
- multiset
- map



- multimap

(hash表实现)

- hash\_set
- hash\_multiset
- hash\_map
- hash\_multimap

### 3. 无序容器

- unordered\_map
- unordered\_multimap
- unordered\_set
- unordered\_multiset

1. vector:底层数据结构为数组，支持快速随机访问。
2. array：固定大小数组。支持快速随机访问，不能添加或者删除元素
3. list:底层数据结构为双向链表，支持快速增删。
4. forward\_list:单向链表，只支持单向顺序访问
5. deque:底层数据结构为一个中央控制器和多个缓冲区，支持首尾（中间不能）快速增删，也支持随机访问。
6. stack:底层一般用23实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时
7. queue:底层一般用23实现，封闭头部即可，不用vector的原因应该是容量大小有限制，扩容耗时（stack和queue其实是适配器,而不叫容器，因为是对容器的再封装）
8. priority\_queue:底层数据结构一般为vector为底层容器，堆heap为处理规则来管理底层容器实现
9. set:底层数据结构为红黑树，有序，不重复。
10. multiset:底层数据结构为红黑树，有序，可重复。
11. map:底层数据结构为红黑树，有序，不重复。
12. multimap:底层数据结构为红黑树，有序，可重复。
13. hash\_set:底层数据结构为hash表，无序，不重复。
14. hash\_multiset:底层数据结构为hash表，无序，可重复。
15. hash\_map :底层数据结构为hash表，无序，不重复。
16. hash\_multimap:底层数据结构为hash表，无序，可重复。
17. 4种无序容器

| 无序容器               | 功能                                                                                                    |
|--------------------|-------------------------------------------------------------------------------------------------------|
| unordered_map      | 存储键值对 <key, value> 类型的元素，其中各个键值对键的值不允许重复，且该容器中存储的元素                                                   |
| unordered_multimap | 和 unordered_map 唯一的区别在于，该容器允许存储多个键相同的元素                                                               |
| unordered_set      | 不再以键值对的形式存储数据，而是直接存储数据元素本身（当然也 key 和值 value 相等的键值对，正因为它们相等，因此只存储 value 即可）。另外，该容器存储的元素不能重复，且容器内部存储的元素 |
| unordered_multiset | 和 unordered_set 唯一的区别在于，该容器允许存储值相同的元素。                                                                |

18. 支持随机访问的容器：string,array,vector,deque  
支持在任意位置插入/删除的容器：list,forward\_list  
支持在尾部插入元素：vector,string,deque

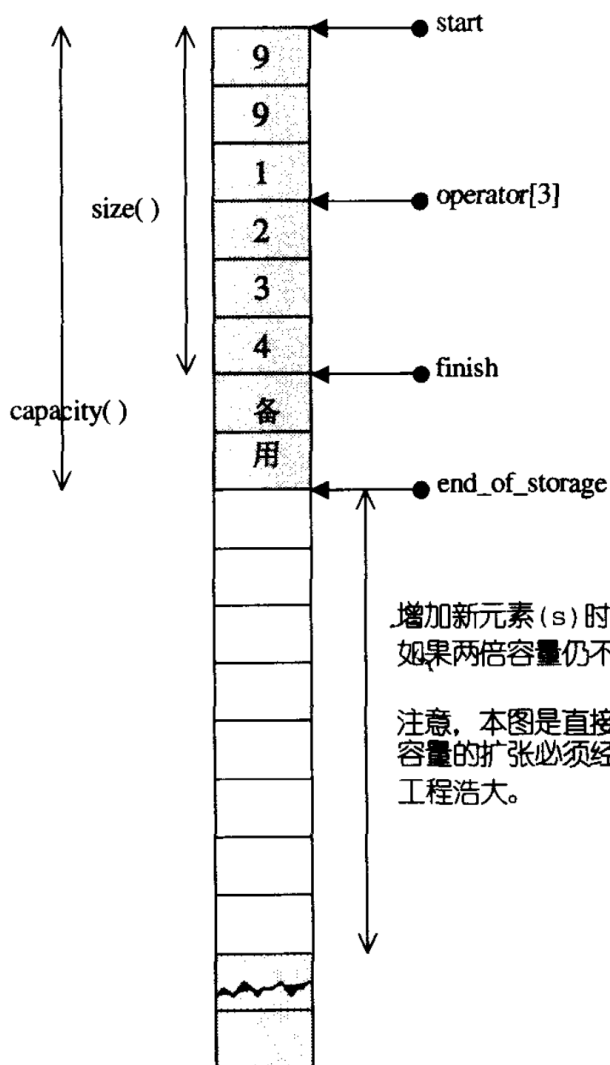
# vector

## vector描述

vector是动态空间，随着元素的加入它内部机制会自行空充空间以容纳新元素。vector维护了一个连续的线性空间，普通指针就可以满足要求作为vector的迭代器，随机访问迭代器。vector里面其实有三个迭代器，分别是指向空间头部的iterator，指向空间尾部的iterator和指向可用空间的iterator。当有新的元素插入时，如果当前容量够就直接插入，如果容量不够则扩容至两倍或1.5倍，如果两倍不足，则扩容至足够大的空间。由于扩充过程不是在原有的空间后面追加，而是重新申请一块新的连续内存，所以所有迭代器都会失效。

## vector的底层原理

vector底层是一个**动态数组**，包含三个迭代器，start和finish之间是已经被使用的空间范围，end\_of\_storage是整块连续空间包括备用空间的尾部。如图：



经过以下操作：

```
vector<int> iv(2, 9);
iv.push_back(1);
iv.push_back(2);
iv.push_back(3);
iv.push_back(4);
```

vector 内存及各成员呈现  
左图状态

增加新元素(s)时，如果超过当时的容量，则容量会扩充至两倍。  
如果两倍容量仍不足，就扩张至足够大的容量。

注意，本图是直接原空间之后画上新增空间，其实没那么单纯。  
容量的扩张必须经历“重新配置、元素移动、释放原空间”等过程，  
工程浩大。

<https://blog.csdn.net/Wizardtoh>

## vector内存增长机制

当空间不够装下数据（`vec.push_back(val)`）时，会自动申请另一片更大的空间（1.5倍或者2倍），然后把原来的数据拷贝到新的内存空间，接着释放原来的那片空间（gcc是2倍，vs下的mingw是1.5倍）

当释放或者删除（`vec.clear()`）里面的数据时，其存储空间不释放，仅仅是清空了里面的数据。

因此，对vector的任何操作一旦引起了空间的重新配置，指向原vector的所有迭代器会都失效了。

## vector中的reserve和resize的区别

reserve是直接扩充到已经确定的大小，可以减少多次开辟、释放空间的问题（优化

push\_back)，就可以提高效率，其次还可以减少多次要拷贝数据的问题。reserve只是保证vector中的空间大小（capacity）最少达到参数所指定的大小n。**reserve()只有一个参数。**

resize()可以改变有效空间的大小，也有改变默认值的功能。capacity的大小也会随着改变。**resize()可以有多个参数。**

vector的reserve增加了vector的capacity，但是它的size没有改变！而resize改变了vector的capacity同时也增加了它的size！

原因如下：

1. reserve是容器预留空间，但在空间内不真正创建元素对象，所以在没有添加新的对象之前，不能引用容器内的元素。加入新的元素时，要调用push\_back()/insert()函数。
2. resize是改变容器的大小，且在创建对象，因此，调用这个函数之后，就可以引用容器内的对象了，因此当加入新的元素时，用operator[]操作符，或者用迭代器来引用元素对象。此时再调用push\_back()函数，是加在这个新的空间后面的。

可能大家平时用reserve()比较多，顾名思义，reserve就是预留内存。为的是避免内存重新申请以及容器内对象的拷贝。说白了，reserve()是给push\_back()准备的！而resize除了预留内存以外，还会调用容器元素的构造函数，不仅分配了N个对象的内存，还会构造N个对象。从这个层面上来说，resize()在时间效率上是比reserve()低的。但是在多线程的场景下，用resize再合适不过。

## vector中size()和capacity()的区别

size()指容器当前拥有的元素个数（对应的resize(size\_type)会在容器尾添加或删除一些元素，来调整容器中实际的内容，使容器达到指定的大小。

capacity()指容器在必须分配存储空间之前可以存储的元素总数。

size表示的这个vector里容纳了多少个元素，capacity表示vector能够容纳多少元素，它们的不同是在于vector的size是2倍增长的。如果vector的大小不够了，比如现在的capacity是4，插入到第五个元素的时候，发现不够了，此时会给他重新分配8个空间，把原来的数据及新的数据复制到这个新分配的空间里。（会有迭代器失效的问题）

## vector的元素类型可以是引用吗？

vector中的元素有两个要求：

1. 元素必须能赋值
2. 元素必须能复制

对于类类型来说，需要拷贝构造和赋值运算符支持，对于像map，set这种容器需要重载运算符<的支持，而引用是必须初始化的，指向一个特定对象，本质上是一个常量指针，因此引用是不能复制，意味着容器的拷贝复制就失效了

容器开辟的时候还没有值，无法初始化，你咋能用引用

## vector迭代器失效的情况

当插入一个元素到vector中，由于引起了内存重新分配，所以指向原内存的迭代器全部失效。

当删除容器中一个元素后,该迭代器所指向的元素已经被删除，那么也造成迭代器失效。erase方法会返回下一个有效的迭代器，所以当我们要删除某个元素时，需要it=vec.erase(it)

## vector在栈还是堆，能开10万个元素的vector吗，怎么扩容的，怎么开辟内存

### 参考链接

无论你的定义是：

```
vector<int*> *p = new vector<int*>;
```

还是

```
vector<int*> p
```

其元素都是在堆上进行分配。

C++语言中，所有 new 和 malloc 创建的变量均存放在堆区，这已经是一个共识。但是鲜为人知的是，STL库中的容器虽没有经过这两个关键字创建，但同样是存放在堆区。这与动态数组性质相同。如果从汇编角度观察便会发现，容器均调用了 allocator 来创建。

## vector作为函数返回值用法

1. 在C++11中提供了RVO/NRVO机制可以防止这种重复拷贝开销。另一种是RVO/NRVO机制实现复制消除机制。
2. RVO机制使用父栈帧（或任意内存块）来分配返回值的空间，来避免对返回值的复制。也就是将Base fun();改为void fun(Base &x);

## emplace\_back和push\_back

### 相同

emplace\_back和push\_back都支持左值和右值的传入。

我们这里就说类元素，不说内置和基本类型了。

传入左值的时候，会调用拷贝构造函数构造出一个匿名对象，然后将该对象存储到vector中

传入右值的时候，调用的是两个函数的移动构造函数。

### 不同

emplace\_back 还支持另一种调用方式，原地构造（in-place construction）！

即emplace\_back的参数是可变的，传入的参数可以是vector类型的构造函数的参数，直接原地构造

比如emplace\_back(10, "test")可以只调用一次constructor

而push\_back(MyClass(10, "test"))中MyClass(10, "test")调用了一次构造函数，同时值传递又调用拷贝构造函数。

## 一个vector内存很大但实际我只用了很小一部分怎么解决

swap

## vector元素是指针类型

清空 vector 数据时，如果保存的数据项是指针类型，需要逐项 delete，否则会造成内存泄漏。

## 频繁调用push\_back的影响

向 vector 的尾部添加元素，很有可能引起整个对象存储空间的重新分配，重新分配更大的内存，再将原数据拷贝到新空间中，再释放原有内存，这个过程是耗时耗力的，频繁对 vector 调用 push\_back()会导致性能的下降。

所以我们可以用reserve（容器预留空间）解决这个问题。

## Vector如何释放空间？

vector的内存占用空间只增不减，比如你首先分配了10,000个字节，然后erase掉后面9,999个，留下一个有效元素，但是内存占用仍为10,000个。

所有的内存空间是在vector析构时候才能被系统回收。

empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使clear()，vector所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用deque。如果vector，可以用swap()来帮助你释放内存。

```
vector(Vec).swap(Vec); //将Vec的内存清除;
vector().swap(Vec); //清空Vec的内存;
```

## vector的插入复杂度

| 插入                  | 删除                 | 访问     |
|---------------------|--------------------|--------|
| push_back<br>$O(1)$ | pop_back<br>$O(1)$ | $O(1)$ |
| insert<br>$O(n)$    | erase<br>$O(n)$    |        |

[https://blog.csdn.net/like\\_ithat](https://blog.csdn.net/like_ithat)

## vector为什么是成倍增长而不是固定大小的一个容量呢？

空间和时间的权衡。

简单来说，空间分配的多，平摊时间复杂度低，但浪费空间也多。

如果是固定数量扩容的话，通过数学方法可以算出来平摊下来的push\_back时间是 $O(n)$

而以倍数增长的扩容方式push\_back的时间复杂度是 $O(1)$

## 为什么选择以1.5倍或者2倍方式进行扩容？而不是3倍4倍扩容？

扩容原理为：申请新空间，拷贝元素，释放旧空间，理想的分配方案是在第N次扩容时如果能复用之前N-1次释放的空间就太好了。

使用2倍（ $k=2$ ）扩容机制扩容时，每次扩容后的新内存大小必定大于前面的总和。

而使用1.5倍（ $k=1.5$ ）扩容时，在几次扩展以后，可以重用之前的内存空间了。

linux下是按照2倍的方式扩容的，而vs下是按照1.5倍的方式扩容的。

## deque

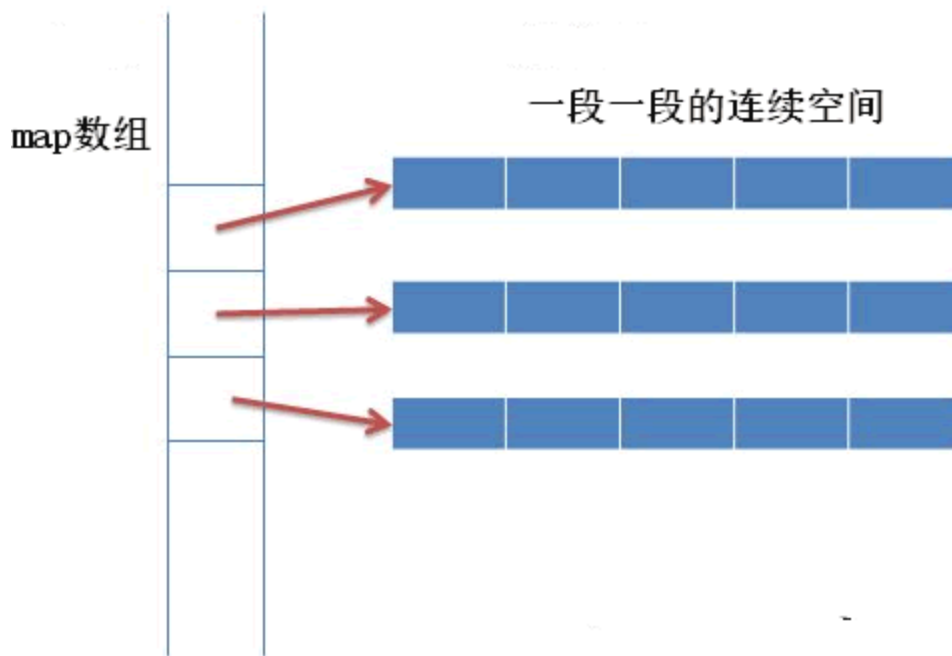
### deque描述

vector是单向开口的连续线性空间，deque是一种双向开口的连续线性空间。双向开口就是说deque支持从头尾两端进行元素的插入和删除操作相比于vector的扩容空间的方式，deque实际上更加贴切的实现了动态空间的概念。deque没有容量的概念，因为它是动态以分段连续空间组合而成，随时可以增加一段新的空间并连接起来。由于要维护这种整体连续的假象，并提供随机存取的接口，也就是说提供random access iterator，避开重新配置，复制，释放的轮回，代价是复杂的迭代器结构。也就是说如果非必要的话，我们尽量使用vector。

### deque容器的底层存储机制

deque 容器存储数据的空间是由一段一段等长的连续空间构成，各段空间之间并不一定是连续的，可以位于在内存的不同区域。





采用一块所谓的map作为主控，这里的map实际上是一块大小连续的空间，其中每一个元素我们称之为节点node，都指向了另一段连续线性空间，成为缓冲区，缓冲期才是deque的真正存储空间主体。

通过建立 map 数组，deque 容器申请的这些分段的连续空间就能实现“整体连续”的效果。换句话说，当 deque 容器需要在

头部或尾部增加存储空间时，它会申请一段新的连续空间，同时在 map 数组的开头或结尾添加指向该空间的指针，由此该空间就串接到了 deque 容器的头部或尾部。

**问：如果 map 数组满了怎么办？**

答：再申请一块更大的连续空间供 map 数组使用，将原有数据（很多指针）拷贝到新的 map 数组中，然后释放旧的空间。

## deque的iterator

deque另外一个关键的就是它的iterator设计。deque中的iterator有四个部分，cur指向缓冲区现行元素。First指向缓冲区的头，last指向缓出去的尾（有时会包含备用空间），node指向管控中心。所以总结来说，deque数据结构中包含了指向第一个节点的iterator star和指向最后一个节点的iterator finist，一块儿连续空间作为主控map，也需要记住map的大小，以备判断何时配置更大的map。

## deque和vector的区别

- vector是单向开口的连续区间，deque是双向开口的连续区间（可以在头尾两端进行插入和删除操作）
- deque没有提供空间保留功能，也就是没有capacity这个概念，而vector提供了空间保留功能。即vector有capacity和reserve函数，deque 和 list一样，没有这两个函数。

deque是在功能上合并了vector和list。

**优点：**

1. 随机访问方便，即支持[]操作符和vector.at()
2. 在内部方便的进行插入和删除操作
3. 可在两端进行push、pop

**缺点：** 占用内存多

## list

### list描述

与 vector 相比，list的好处就是每次插入或删除一个元素就配置或释放一个空间，而且原有的迭代器也不会失效。list 是一个双向链表，普通指针已经不能满足 list 迭代器的需求，因为 list 的存储空间是不连续的。list的迭代器必需具备前移和后退功能，所以list提供的是 BidirectionalIterator。list 的数据结构中只要一个指向 node 节点的指针就可以了。

### list和vector的区别

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 $O(n)$ 。它与数组最大的区别就是vector不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进行扩容。

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。非连续存储结构：list是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点（prev）和指向下一个元素的节点（next）。因此list可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

**区别如下：**

1. vector的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易

操作。

2. list的访问要遍历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。
3. 从遍历上来说，list是单向的，vector是双向的。
4. vector中的迭代器在使用后就失效了，而list的迭代器在使用之后还可以继续使用。

## 怎么找某vector或者list的倒数第二个元素

### vector

```
int mySize = vec.size();vec.at(mySize -2);
```

### list

list不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问list里的元素只能遍历，不过你要是只需要访问list的最后N个元素的话，可以用反向迭代器来遍历：

```
list<int> test_list;
for (size_t i = 1; i < 8; i++) {
 test_list.push_back(i*2);
}
list<int>::reverse_iterator rit = find(test_list.rbegin(), test_list.rend(), 8);
list<int>::iterator it(rit.base());
cout << *rit << endl;
cout << *it << endl;
```

## stack

是一种先进后出的数据结构，只有一个出口，stack 允许从最顶端新增元素，移除最顶端元素，取得最顶端元素。deque 是双向开口的数据结构，所以使用 deque 作为底部结构并封闭其头端开口，就形成了一个 stack。

## queue

是一种先进先出的数据结构，有两个出口，允许从最底端加入元素，取得最顶端元素，从最底端新增元素，从最顶端移除元素。deque 是双向开口的数据结构，若以 deque 为底部结构并封闭

其底端的出口，和头端的入口就形成了一个 queue。（其实 list 也可以实现 deque）

## vector, list, map等容器使用场合是什么？

顺序容器首选 vector，关联容器首选 unordered\_map。

## 关联式容器（map,multimap,set,multiset）

### 为何关联容器的插入删除效率一般比用其他序列容器高？

答：关联容器一般指map,multimap,set,multiset这四种底层实现都是红黑树。对于关联容器来说，存储的只是节点。插入删除只是节点指针的换来换去，不需要做内存拷贝和内存移动。

### set multiset 重复数据是怎么实现的？

set提供的插入函数接口：

```
pair<iterator,bool> insert(const value_type& elem);
iterator insert(iterator pos_hint, const value_type& elem);
```

multiset提供的插入函数的接口：

```
iterator insert(const value_type& elem);
iterator insert(iterator pos_hint, const value_type& elem);
```

set的返回值型别是由pair组织起来的两个值：

1. pair第一个元素返回新元素的位置，或返回现存的同值元素的位置。
2. pair第二个元素表示插入是否成功。

set的第二个insert函数，如果插入失败，就只返回重复元素的位置！

multiset中所有拥有位置提示参数的插入函数的返回值型别是相同的。这样就确保了至少有了一个通用型的插入函数，在各种容器中有共通接口。

## 为何每次insert之后， 以前保存的iterator不会失效？

因为每次只是节点指针的改变， 每个节点的内存没有改变。iterator就是指向每个节点内存的指针， 所以插入之后， 该指针是不会变的。

## 为何map和set不能像vector一样有个reserve函数来预分配数据？

最主要的原因是关联式容器内部存储的不是元素的本身， 同时还有元素的节点信息， 是无法预分配的。

## 当数据元素增多时（10000和20000个比较）， map和set的插入和搜索速度变化如何？

查找时间复杂度是 $\log n$ ， 所以不会有什么变化

## map用[]越界会发生什么？

- 首先要了解通过key获得map的方法都有哪些？
  - 首先要判断key存不存在， 有两种判断办法
    - a. 通过count函数计算key的个数， 0表示不存在， 1表示存在1个
    - b. 通过find函数， 如果 `map.find(key) != map.end()`， 即find指针不是最后一个就表明找到了
  - 获得value的方法
    - a. 通过重载的 `[]`， 即 `map[key]`
    - b. 通过map自身的迭代器iter，  
即 `iter->first = key` 和 `iter->second = value`
- 如果通过下标去访问map中的value会有越界的可能， 会发生什么？  
map源码如下：

```
mapped_type& operator[](key_type&& _Keyval){
 iterator _Where = this->lower_bound(_Keyval);
 //如果没有返回end指针
 if (_Where == this->end() || this->comp(_Keyval, this->_Key(_Where._Mynode())))
 _Where = this->insert(_Where, _STD pair<key_type, mapped_type>(_STD move(_Keyval), mapped_type()));
 return ((*_Where).second);
}
```

可以发现如果当没有要找的key的话，会返回end，然后会执行insert语句，插入一个pair类型的对儿，key是你输入的和默认的值。

所以会自动插入一个你搜索的key和默认的值。如果value为内置类型，其值将被初始化为0；如果value为自定义数据结构且用户定义了默认值则初始化为默认值，否则初始化为0。

## map和set为什么用红黑树

高度越小越好，BST这种有特殊情况，比如只有左子树有值，导致O(n)复杂度

AVL树平衡有点太变态了，导致每次自适应的时候效率低一点。所以综合来说红黑树是最优秀的

## map中[]与find的区别？

1. map的下标运算符[]的作用是：将key作为下标去执行查找，并返回对应的值；如果不存在这个key，就将一个具有该关键码和值类型的默认值的项插入这个map。
2. map的find函数：用key执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

## 容器内部删除一个元素

顺序容器（序列式容器，比如vector、deque）

erase迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list除外)，所以不能使用erase(it++)的方式，但是erase的返回值是下一个有效迭代器；

```
It = c.erase(it);
```

关联容器(关联式容器，比如map、set、multimap、multiset等)

erase迭代器只是被删除元素的迭代器失效，但是返回值是void，所以要采用erase(it++)的方式删除迭代器；

```
c.erase(it++)
```

## allocator的使用

[参考链接](#)

## 迭代器的底层机制和失效的问题

迭代器失效的定义：对容器的操作影响了元素的存放位置，称为迭代器失效。

失效情况分为三种：

1. 当容器调用 erase() 方法后，当前位置到容器末尾元素的所有迭代器全部失效。
2. 当容器调用 insert() 方法后，当前位置到容器末尾元素的所有迭代器全部失效。
3. 如果容器扩容，在其他地方重新又开辟了一块内存。原来容器底层的内存上所保存的迭代器全都失效了。

通过迭代器可以在不了解容器内部原理的情况下遍历容器

它的底层实现包含两个重要的部分：萃取技术和模板偏特化。

- 萃取技术

萃取技术可以进行类型推导，根据迭代器的不同类型处理不同流程。例如vector的迭代器类型为随机访问迭代器，list为双向迭代器

- 模板偏特化

比较深奥

- 迭代器的类型

- i. 输入迭代器。从容器中读取元素。输入迭代器只能一次读入一个元素向前移动
- ii. 输出迭代器。向容器中写入元素。输出迭代器只能一次一个元素向前移动。

- iii. 正向迭代器。组合输入迭代器和输出迭代器的功能，并保留在容器中的位置
- iv. 双向迭代器。组合正向迭代器和逆向迭代器的功能
- v. 随机访问迭代器。组合双向迭代器的功能与直接访问容器中任何元素的功能，即可向前向后跳过任意个元素

## 序列式容器失效

**\*\*失效原因：**\*\*因为 `vector`、`deque` 使用了连续分配的内存，`erase` 操作删除一个元素导致后面所有的元素都会向前移动一个位置，这些元素的地址发生了变化，所以当前位置到容器末尾元素的所有迭代器全部失效。

**\*\*解决办法：**\*\*由于 `erase` 可以返回下一个有效的iterator，因此 `q.erase(it)` 不行，但是 `it=q.erase(it)` 可以



```

int main() {
 vector<int> q{ 1,2,3,4,5,6 };
 // 在这里想把大于2的元素都删除
 for (auto it = q.begin(); it != q.end(); it++) {
 if (*it > 2)
 q.erase(it); // 这里就会发生迭代器失效
 }
 // 打印结果
 for (auto it = q.begin(); it != q.end(); it++) {
 cout << *it << " ";
 }
 cout << endl;
 return 0;
}

//解决办法
for(auto it=q.begin();it!=q.end();)
{
 if(*it>2)
 {
 it=q.erase(it); // 这里会返回指向下一个元素的迭代器，因此不需要再自加了
 }
 else
 {
 it++;
 }
}

```

## 链表式容器失效和关联式容器失效

**\*\*失效原因：**\*\*链表的插入和删除节点不会对其他节点造成影响，因此只会使得当前的iterator失效

**\*\*解决办法：**\*\*利用 `erase` 可以返回下一个有效的iterator的特性，或者直接 `iterator++`

```

//方法1
for (iter = cont.begin(); it != cont.end();)
{
 (*iter)->doSomething();
 if (shouldDelete(*iter))
 cont.erase(iter++);
 else
 iter++;
}
//方法2
for (iter = cont.begin(); iter != cont.end();)
{
 (*it)->doSomething();
 if (shouldDelete(*iter))
 iter = cont.erase(iter); //erase删除元素，返回下一个迭代器
 else
 ++iter;
}

```

## 迭代器和指针的区别

迭代器实际上是对“遍历容器”这一操作进行了封装。迭代器不是指针，是类模板。重载了指针的一些操作符如：`:`，`->`，`*`，`++`，`--`等。

在编程中我们往往会用到各种各样的容器，但由于这些容器的底层实现各不相同，所以对他们进行遍历的方法也是不同的。例如，数组使用指针算数就可以遍历，但链表就要在不同节点直接进行跳转。c++我觉得是一门非常讲究方便的语言，显然这种情况是不能够出现的。因此就出现了迭代器，将遍历容器的操作封装起来，可以针对所有容器进行遍历。

## 迭代器的种类

- 前向迭代器（forward iterator）  
则 `p` 支持 `++p`，`p++`，`*p` 操作，还可以被复制或赋值，可以用 `==` 和 `!=` 运算符进行比较。
- 双向迭代器（bidirectional iterator）  
双向迭代器具有正向迭代器的全部功能，除此之外，假设 `p` 是一个双向迭代器，则

还可以进行 `--p` 或者 `p--` 操作（即一次向后移动一个位置）。

- 随机访问迭代器（random access iterator）

随机访问迭代器具有双向迭代器的全部功能。除此之外，假设 `p` 是一个随机访问迭代器，`i` 是一个整型变量或常量，则 `p` 还支持以下操作：

- i. `p+=i`：使得 `p` 往后移动 `i` 个元素。
- ii. `p-=i`：使得 `p` 往前移动 `i` 个元素。
- iii. `p+i`：返回 `p` 后面第 `i` 个元素的迭代器。
- iv. `p-i`：返回 `p` 前面第 `i` 个元素的迭代器。
- v. `p[i]`：返回 `p` 后面第 `i` 个元素的引用。

- 输入迭代器 (input iterator)

可用于读取容器中的元素，但是不保证能支持容器的写入操作。

只支持自增运算

- 输出迭代器 (output iterator)

可视为与输入迭代器功能互补的迭代器；

输出迭代器可用于向容器写入元素，但是不保证能支持读取容器内容。

只支持自增运算

## 仿函数

## 算法

### **stable\_sort**

`stable_sort (first, last)`和`sort()` 函数功能相似，不同之处在于该函数不会改变它们的相对位置。

`stable_sort()` 函数是基于归并排序实现的。

`sort()` 函数是基于快速排序实现的。

## STL容器是线程安全的吗？

众所周知，STL容器不是线程安全的。对于vector，即使写方（生产者）是单线程写入，但是并发读的时候，由于潜在的内存重新申请和对象复制问题，会导致读方（消费者）的迭代器失效。实际表现也就是招致了core dump。另外一种情况，如果是多个写方，并发的`push_back()`，也

会导致core dump。

解法如下：

1、加锁

2、通过固定vector的大小，避免动态扩容（无push\_back）来做到lock-free！即在开始并发读写之前（比如初始化）的时候，给vector设置好大小。代码如下：

```
vector<int> v;
v.resize(1000);
```

注意是resize，不是reserve！

## c++STL中vector、list和map插入1000万个元素，消耗对比

毫无疑问vector最小

使用std::map和std::list存放数据，消耗内存比实际数据大得多

原因：std::list和std::map属于散列容器，容器的空间之间是通过指针来关联的，所以指针会占用一部分内存，当自身存放的数据较2\*8（std::list，双向链表）差别不大时，会有很大的额外内存开销。为了避免此开销，可以使用线性容器，std::vector。

+++

## c++工具类

### double buffer代替锁（无锁化）

在“读多写一”的场景下代替锁，实现高效的性能。

在《Linux服务器面试》那部分中详细讲述了锁的开销问题

**\*\*思路：**\*\*一个线程对一个变量进行写的话不会有任何问题，同时多个线程读一个变量也不会有

什么问题。所以我们可以想到，用两个变量或者对象是否可以代替锁的功能。具体的思路是假设有两个共享资源A和B，当前情况下，读线程正在读资源A。突然在某一个时刻，写线程需要更新资源，写线程发现资源A正在被访问，那么其更新资源B，更新完资源B后，进行切换，让读线程读资源B，然后写线程继续写资源A，这样就能完全实现了lock-free的目标，此种方案也可以成为双buffer方式。

**\*\*重点：**\*\*一定要保证写的时候那个buffer不能有读操作。因为写操作时删除的话，读操作读到了被删除的元素位置，会产生不可预估的问题。

**具体实现：**

- 双 buffer 的备份机制，避免了同时读写同一变量。双buffer 就是指对于通常要被多个线程访问的变量，再额外定义一个备份变量。由于是一写多读，写线程只向备份变量中写入，而所有的读线程只需要访问主变量本身即可。当写进程对备份变量的写操作完成后，会触发主变量指针和备份变量指针的互换操作，即指针切换，从而将原变量和备份变量的身份进行互换，达到数据更新的目的。
- 共享指针 shared\_ptr，由于其记录了对变量的引用次数，因而可以避免指针切换时的“访问丢失”问题。要解决的问题是就是如何判断一个对象上存在线程读操作。  
std::shared\_ptr内部有个成员函数use\_count()来判断当前智能指针所指向变量的访问个数

代码部分一直没看明白

1

2

3

4

**扩展**

双buffer方案在“一写多读”的场景下能够实现lock-free的目标，那么对于“多写一读”或者“多写多读”场景，是否也能够满足呢？

答：不太合适。首先在多写的场景下，多个写之间需要通过锁来进行同步，虽然避免了对读写互斥情况加锁，但是多线程写时通常对数据的实时性要求较高，如果使用双buffer，所有新数据必须要等到索引切换时候才能使用，很可能达不到实时性要求。其次多线程写时若用双buffer模

式，则在索引切换时候也需要给对应的对象加锁，并且也要用类似于上面的while循环保证没有现成在执行写入操作时才能进行指针切换，而且此时也要等待读操作完成才能进行切换，这时候就对备用对象的锁定时间过长，在数据更新频繁的情况下是不合适的。

### 应用场景：

双buffer方案在多线程环境下能较好的解决“一写多读”时的数据更新问题，特别是适用于数据需要定期更新，且一次更新数据量较大的情形。

## 内存池

### 内存池概述

我们在进行数据库操作的时候为了提高数据库（关系型数据库）的访问瓶颈，除了在服务器端增加缓存服务器（例如 redis）缓存常用的数据之外，还可以增加连接池，来提高数据库服务器的访问效率。一般来说，对于数据库操作都是在访问数据库的时候创建连接，访问完毕断开连接。但是如果在高并发情况下，有些需要频繁处理的操作就会消耗很多的资源和时间，比如：

1. 建立通信连接的 TCP 三次握手
2. 数据库服务器的连接认证  
数
3. 数据库服务器关闭连接时的资源回收
4. 断开通信连接的 TCP 四次挥手

### 连接数据库的步骤

MySQL 数据库是一个典型的 C/S 结构，即：客户端和服务端。如果我们部署好了 MySQL 服务器，想要在客户端访问服务端的数据，在编写程序的时候就可以通过官方提供的 C 语言的 API 来实现。

在程序中连接 MySQL 服务器，主要分为已经几个步骤：

- 初始化连接环境
- 连接 mysql 的服务器，需要提供如下连接数据：
  - i. 服务器的 IP 地址
  - ii. 服务器监听的端口（默认端口是 3306）
  - iii. 连接服务器使用的用户名（默认是 root），和这个用户对应的密码

iv. 要操作的数据库的名字

- 连接已经建立, 后续操作就是对数据库数据的添删查改(调用API完成)
- 如果要进行数据 添加 / 删除 / 更新, 需要进行事务的处理需要对执行的结果进行判断
  - 成功: 提交事务
  - 失败: 数据回滚
- 数据库的读操作 -> 查询 -> 得到结果集
- 遍历结果集 -> 得到了要查询的数据
- 释放资源