1. char\*与string有什么区别？

char\*与string的本质区别是string是一个容器，C++封装了一个string类，而char\*是一个指针，指向数组的首地址。

char\*支持向string的隐式转换。

Char\* c[]={1,2,3};

string p=c;

而string转char\*一般需要用到一个c\_str()函数

String的内存管理交给了系统，除非系统内存池用完，否则不会出现内存问题。而char\*的内存管理需要我们自己来处理。

当我们要存储一个串，但是不知道需要多少内存时，用string来处理比较好。当知道存储的内存时，可以两者都用。

String提供了各种成员函数来处理串中的每一个字符。

2．什么是多态？

同一操作作用于不同对象，可以有不同的解释，从而产生不同的执行结果。

编译时多态：编译时多态是通过重载实现的。对于非虚成员来说，系统在编译时，根据传递的参数，返回的类型等信息来决定实现何种操作。

运行时多态：运行时多态是通过虚成员实现的。

父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。

1. 虚函数是怎么实现的？

虚函数是通过虚函数表实现的。

如果一个类中含有虚函数，则系统会为这个类分配一个指针成员指向一张虚函数表，表中每一项指向一个虚函数地址，实际上就是一个函数指针的数组。

对象存的是虚函数表的指针。当此类被继承时，子类如果也写了虚函数，就在子类的虚函数表中将父类的函数地址覆盖，否则继承父类的虚函数地址。

实例化之后，对象有一个虚函数指针，虚函数指针指向虚函数表，这样程序运行的时候，通过虚函数指针找到的虚函数表就是根据对象的类型来指向的了。

类的虚函数表是一块连续的内存，每个内存单元中记录一个JMP指令的地址。  
注意的是，编译器会为每个有虚函数的类(不是为每一个对象创建，对象中只有一个指向虚函数表的指针)创建一个虚函数表，该虚函数表将被该类的所有对象共享。类的每个虚成员占据虚函数表中的一行。如果类中有N个虚函数，那么其虚函数表将有N\*4字节的大小。

虚函数表生成是在编译期间。一个类对象的内存布局首先是该类的vtbl指针，然后才是对象数据。

1. 构造函数调用虚函数？

class A{

public:

A（）{print();}

virtual void print(){printf(“I am A”);};

}

class B : public A{

public:

virtual void print(){printf(“I am B”);};

}

则语句 B b;会输出什么？

在构造函数调用虚函数不会发生作用，因为基类的构造函数在派生类之前执行，当基类构造函数运行时，派生类对象还没有被初始化。

如果基类构造函数调用期间的虚函数向下匹配到派生类，派生类的函数可能会涉及到本地数据成员，但是那些数据成员还没有被初始化，这是危险的，c++并不允许。因而c++直接执行基类的函数。

1. 初始化列表和赋值的区别？

没什么太多差别。主要差别在于：

1. 当类中含有const和reference成员变量时，他们只能被初始化而不能做赋值操作，因此只能用初始化列表。
2. 类的构造函数需要调用其基类的构造函数的时候，一定需要初始化列表。
3. 有了malloc/free 为什么还需要new/delete?

对于非内部数据类型的对象而言，对象在消亡之前需要自动执行析构函数。由于malloc/free只是库函数而不是运算符，不在编译器控制权限之内，不能把执行构造函数和析构函数的任务强加于malloc/free，因而只能使用new/delete运算符。

1. C和C++有什么区别？
2. C时面向过程的语言，是一个结构化的语言，就是考虑如何通过一个过程对输入进行处理得到输出。C++是面向对象的，主要的特征是“封装，继承，多态”。封装隐藏了实现的细节，使得代码模块化，派生类可以继承父类的数据和方法，扩展了已经模块化的代码，实现了代码重用，多态则是“一个接口，多种实现”。
3. C和C++动态管理内存的方法不一样，c使用malloc/free，c++除此之外使用new/delete关键字

class Test {

public:

    Test() {

        \_num = new int;

    }

    virtual ~Test() {

        delete \_num;

    }

private:

    int\* \_num;

};

int main() {

    Test\* t = new Test;

    free(t); //此时由于 Test 的析构函数没有被调用，将导致为 \_num 申请的内存没有被释放。

    return 0;

}

1. C++支持函数重载，C不支持函数重载
2. C++中有引用，C中不存在引用的概念。
3. C++中指针和引用的区别？
4. 指针是一个新的变量，存储了另一个变量的地址。而引用只是一个别名，还是变量本身，对引用的任何操作就是对变量本身的操作。
5. 引用只有一级，而指针可以有多级。
6. 引用声明就必须初始化，且初始化之后不能再改变为其它对象的引用，所以不存在空引用。指针则可以先声明，稍后初始化，这就会存在空指针。
7. 结构体struct和联合体union的区别？

结构体：不同类型的数据组合成一个整体，是自定义类型。结构体中每个成员拥有自己独立的地址，他们是同时存在的。

联合体：不同类型的几个变量共用一段内存，它们不能同时存在。

Sizeof(struct)是内存对齐后所有成员长度的总和，sizeof(union)是内存对齐后最长数据成员的长度。

1. 为什么要内存对齐？
2. 平台原因：不是所有的硬件平台都能访问任意地址上的任意数据；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
3. 性能原因：数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要访问内存两次，而对齐的内存只需要访问一次。

cpu把内存当成一块一块的，块的大小是2，4，6，8，16个字节，因此cpu读取数据时是一块一块地读取的。假设cpu读取块的大小为4字节，且我们需要的数据也是4字节，如果从0字节读取，则只需要读取0-3就可以了。而如果该4字节存储在1字节的位置，则要读取两次。

1. #define与const的区别？

1.#define定义的常量没有类型，所给出的是一个立即数；const定义的常量有类型名字，存放在静态区。

2.处理阶段不同。#define定义的宏变量在预处理阶段进行替换，const变量在编译时确定其值。

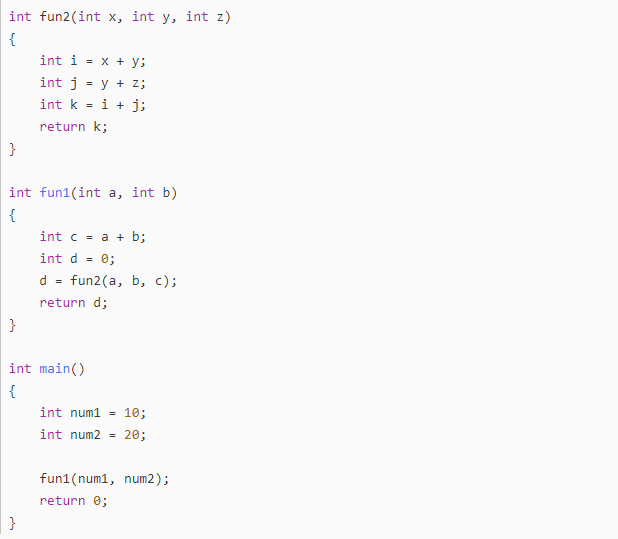
1. 重载overload 覆盖override 重写overwrite，这三者之间的区别？
2. 重载：将语义相近的几个函数用同一个函数名表示，但是参数和返回值不同。
3. 覆盖：派生类覆盖基类的虚函数，实现接口的重用。

C++11 override作用： 在成员函数声明和定义中，override确保该函数为虚函数并覆写来自基类的虚函数。位置：函数调用运算符之后，函数体或纯虚函数“=0“之前。在派生类的成员函数中使用override时，如果基类中无此函数，或基类中的函数并不是虚函数，编译器会给出相关错误信息。

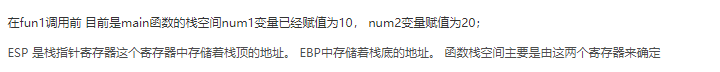
1. 重写：派生类的函数屏蔽了与其同名的基类函数，规则如下：
2. 如果派生类的函数与基类的函数同名，但是参数不同，此时不论有没有virtual关键字，基类的函数将被隐藏。
3. 如果派生类的函数与基类的函数名相同，且参数也相同，但是基类函数中没有virtual关键字，基类的函数被隐藏。
4. delete与delete[]的区别？

Delete只会调用一次析构函数，而delete[]会调用每个成员的析构函数。用new 分配的内存用delete释放，用new[]分配的内存用delete[]释放。

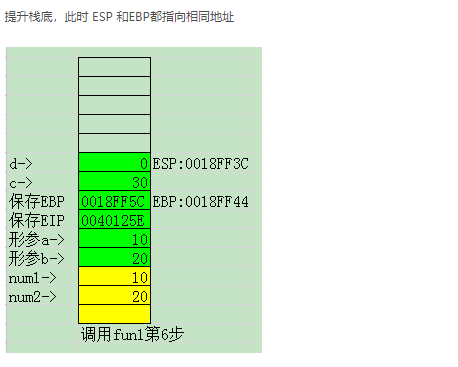
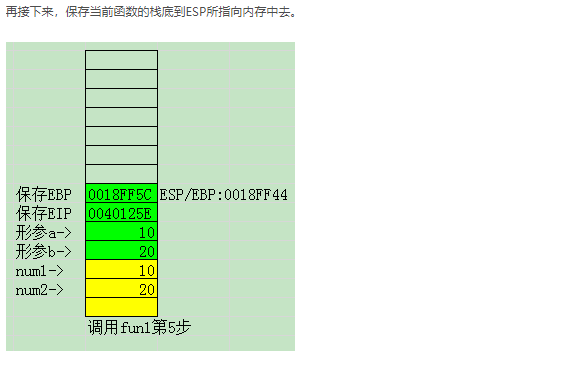
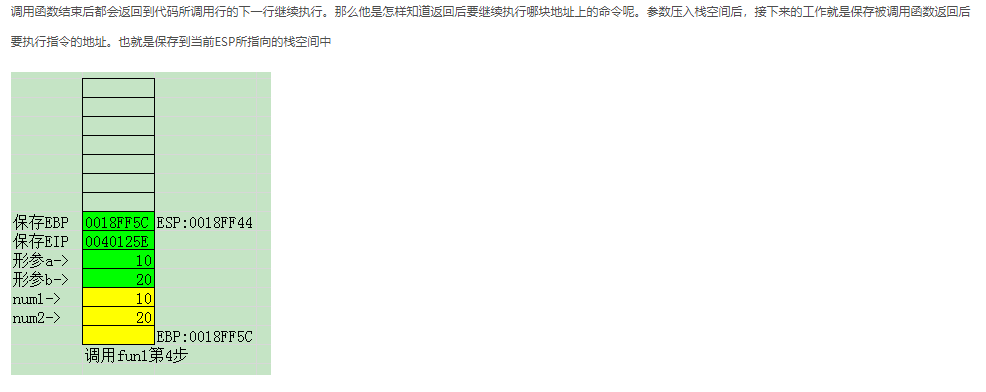
1. 函数栈的空间解析

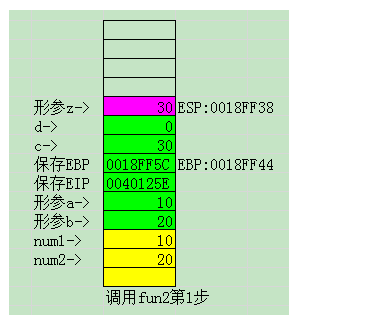


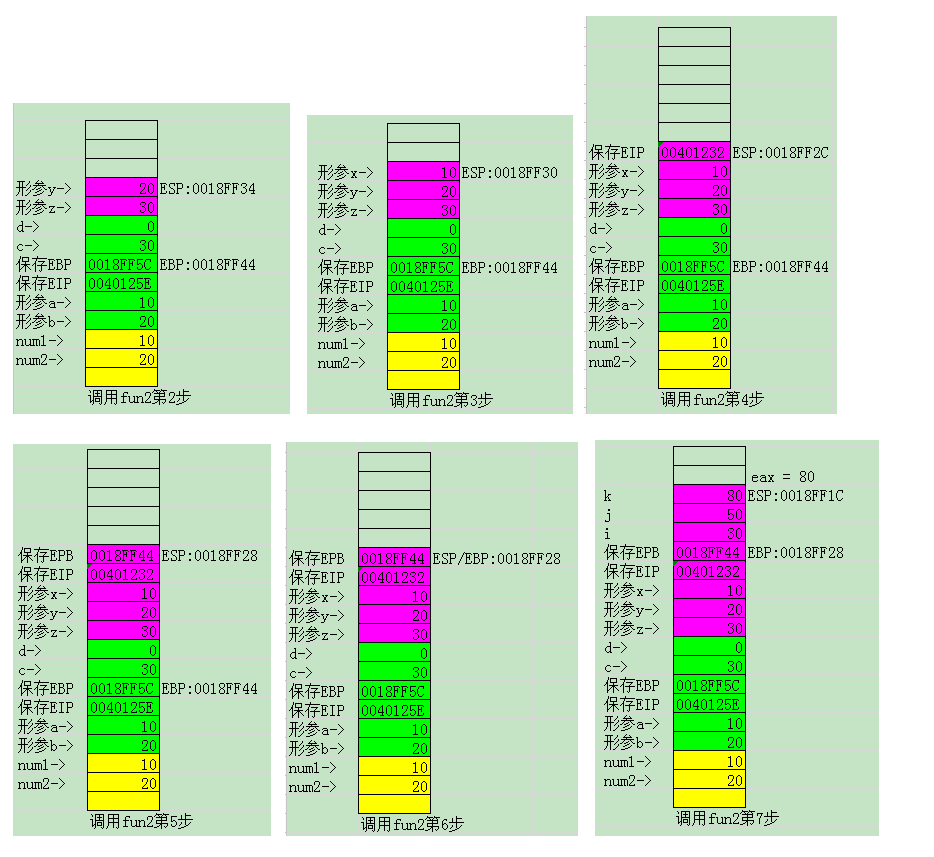












1. STL中map和set的原理？

Map和set的底层都是通过红黑树来实现的，红黑树是一种二叉查找树。满足一些规则：

1. 每一个节点要么红色，要么黑色
2. 根节点和叶子节点一定会是黑色
3. 如果一个节点是红色，其子节点必须为黑色
4. 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。
5. C++文件编译与执行的四个阶段：
6. 预处理
7. 编译
8. 汇编
9. 链接
10. C++内存管理：

C++内存被分成五个区：

堆，栈，静态区（存储全局变量和静态变量），自由存储区（由malloc分配的内存，和堆十分相似，由对应的free进行释放），全局区（存放全局变量和静态变量），常量区（存放常量，不允许被修改）

1. 构造函数为什么不定义为虚函数？

因为创建一个对象时需要知道对象的类型，而虚函数是在运行时确定其类型的。而在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型，是类本身还是类的派生类。

从另一个角度来说，虚函数的调用需要虚函数表，而该指针存放在对象的内存空间中。若构造函数声明为虚函数，那么对象都还没创建，还没有内存空间，更没有虚函数表地址来调用虚函数了。

1. 引用是否能够实现动态绑定？

可以。因为引用既可以指向基类对象也可以指向派生类对象。用引用调用的虚函数在运行时确定，被调用的函数是引用所指向的对象的实际类型所定义的。

1. 引用作为参数以及返回值的好处

对比值传递，引用传参的好处：

1. 在函数内部可以对此参数进行修改
2. 提高函数调用和运行效率（没有传值和生成副本的时间和空间消耗）

用引用作为返回值最大的好处就是内存中不产生被返回值的副本。

限制：（1）不能返回局部变量的引用。因为函数返回后局部变量被销毁

（2）不能返回函数内部new分配的内存的引用。这个是因为虽然这块内存是new出来的，但是被返回的引用只是一个临时变量，那么这个引用所指向的空间就无法释放。

1. 纯虚函数

只有声明没有实现的虚函数，是对子类的约束，是接口继承。

包含纯虚函数的类是抽象类，他不能被实例化，只有实现了这个纯虚函数的子类才能生成对象。

Override在一个虚函数后面添加表示这是覆盖的继承来自基类的虚函数并且对它进行重新更改。

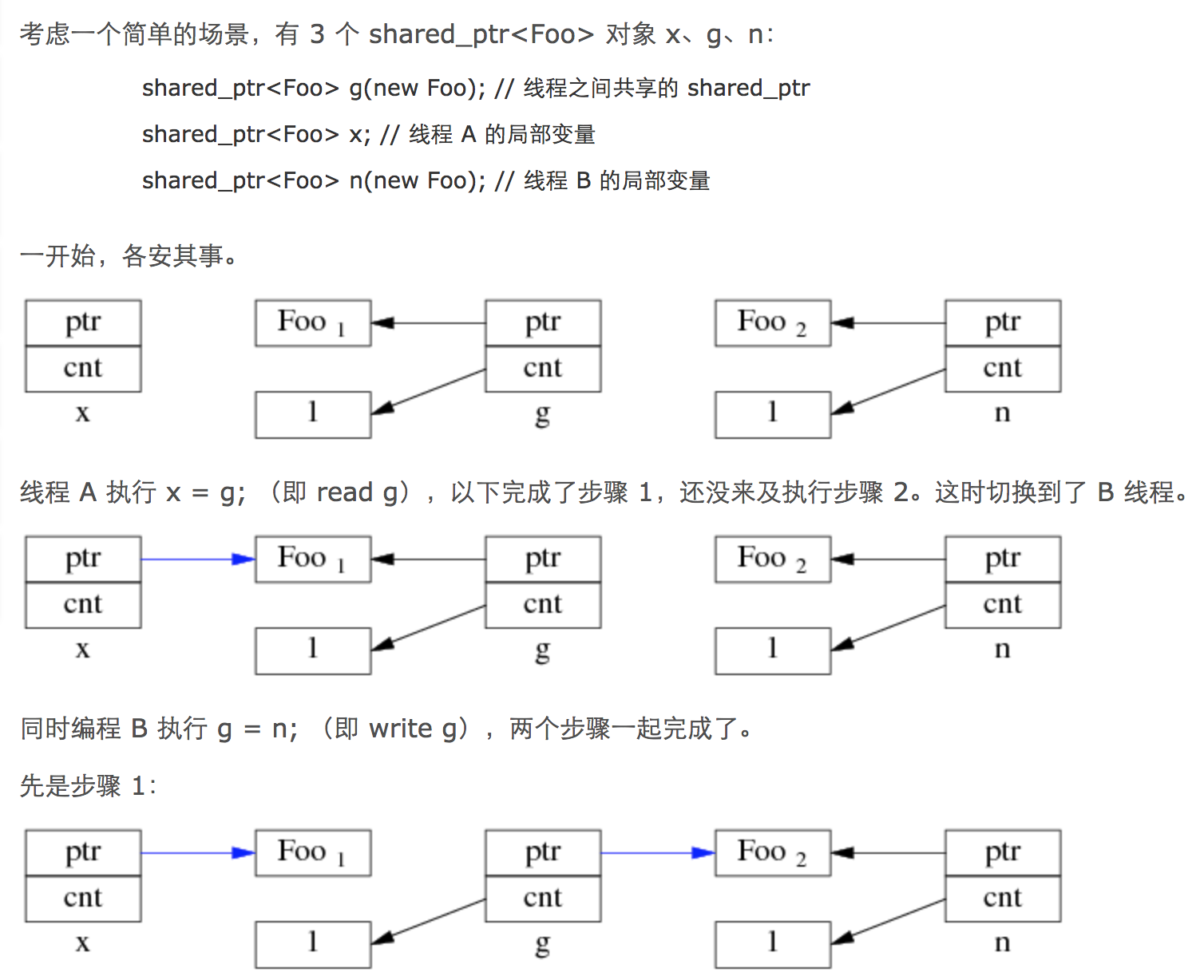
1. 野指针出现的原因
2. 指针变量没有被初始化
3. 指针指向的内存被释放了，但是指针没有置为NULL
4. 指针超过了变量的作用范围，比如b[10],指针b+11
5. 线程安全与线程不安全

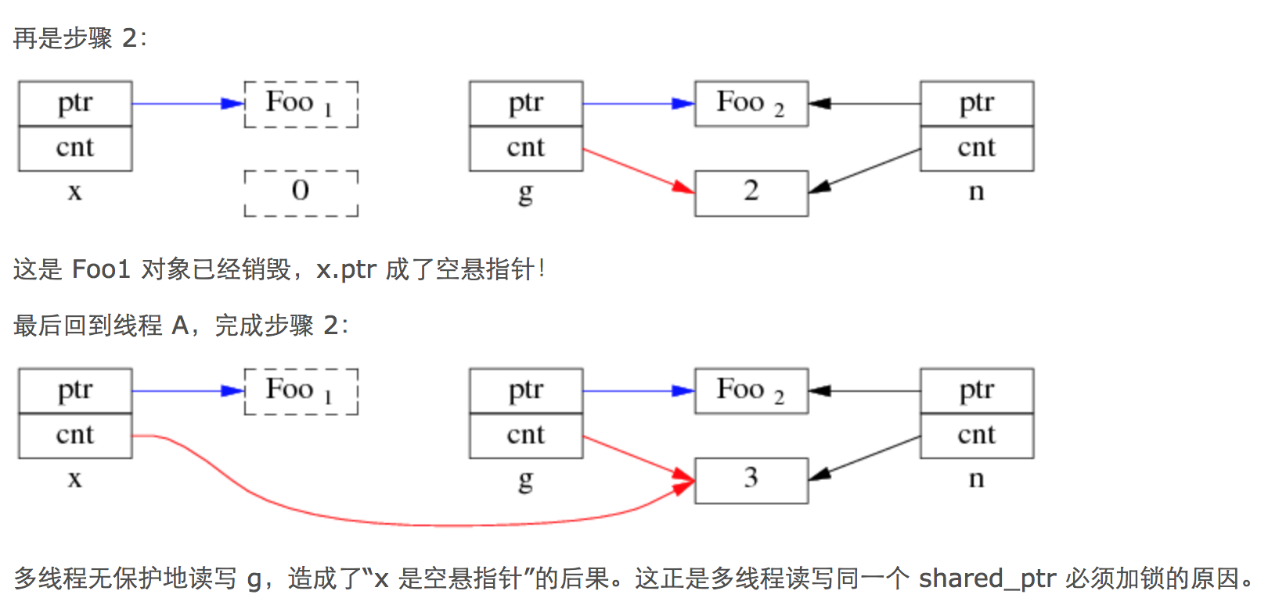
线程安全就是多线程访问时，采用了加锁的机制，当一个线程访问该类的某个数据时，进行保护，其他线程不能进行访问直到该线程读取完，其他线程才可以使用，不会出现数据不一致或者数据污染。

线程不安全就是不提供数据访问保护，有可能多个线程先后更改数据所得的数据就是脏数据。

1. shared\_ptr是线程安全的吗？

shared\_ptr的底层实现是引用计数模型。use\_count是new出来的，保存在堆上，在调用拷贝构造函数和赋值函数时，use\_count++,在析构时，use\_count--。如果在多线程中使用shared\_ptr, 它的引用计数本身是安全且无锁的，但对象的读写则不是，因为 shared\_ptr 有两个数据成员，读写操作不能原子化。我们可以对其进行加锁(定义一个全局的互斥锁)和atomic原子操作来实现。





1. shared\_ptr的循环引用问题？

假设我们自定义的类，其中数据成员也是shared\_ptr类型，我们设置智能指针指向这两个类对象，且设置各自数据成员中的指针指向对方，则各自的use\_count为2，在离开作用域时，由于各自为2，则各自减一，并不会释放堆上的对象。

1. C++中内存泄漏的几种情况？
2. 构造函数和析构函数中new和delete没有配套
3. 在释放对象数组时没有使用delete[],使用了delete
4. 没有将基类的析构函数定义为虚函数，当基类指针指向子类对象时，如果不将析构函数定义为虚函数，则不会向下寻找到子类的析构函数，则子类的资源不会释放。
5. 友元函数和友元类

友元提供了不同类的成员之间，类的成员函数和一般函数之间进行数据共享的机制。通过友元，一个不同函数或者另一个类中的成员函数可以访问类中的私有成员和保护成员。

友元关系不允许被继承。友元关系是单向的，不具有交换性，友元关系不具有传递性。

友元成员函数：使类B中的成员函数成为类A的友元函数，这样类B的改成员函数就可以访问类A的所有成员了。但是限制条件是必须先定义包含成员函数的类。才能够将成员函数设为友元。

1. C++中几种锁机制
2. 互斥锁

互斥锁用于控制多个线程之间共享资源互斥访问的一个信号量。在某一个时刻，只有一个线程可以获取互斥锁，在释放互斥锁之前其他线程都不能获取该互斥锁。如果其他线程想要获取这个互斥锁，只有阻塞等待。

头文件：<pthread.h>

类型：pthread\_mutex\_t

函数：

pthread\_mutex\_init(pthread\_mutex\_t \* mutex, const phtread\_mutexattr\_t \* mutexattr);//动态方式创建锁，相当于new动态创建一个对象

pthread\_mutex\_destory(pthread\_mutex\_t \*mutex)//释放互斥锁，相当于delete

pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;//以静态方式创建锁

pthread\_mutex\_lock(pthread\_mutex\_t \*mutex)//以阻塞方式运行的。如果之前mutex被加锁了，那么程序会阻塞在这里。

pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex)

int pthread\_mutex\_trylock(pthread\_mutex\_t \* mutex);//会尝试对mutex加锁。如果mutex之前已经被锁定，返回非0,；如果mutex没有被锁定，则函数返回并锁定mutex

//该函数是以非阻塞方式运行了。也就是说如果mutex之前已经被锁定，函数会返回非0，程序继续往下执行。

2.条件锁

条件锁就是所谓的条件变量，某一个线程因为某个条件为满足时可以使用条件变量使改程序处于阻塞状态。一旦条件满足以“信号量”的方式唤醒一个因为该条件而被阻塞的线程。最为常见就是在线程池中，起初没有任务时任务队列为空，此时线程池中的线程因为“任务队列为空”这个条件处于阻塞状态。一旦有任务进来，就会以信号量的方式唤醒一个线程来处理这个任务。这个过程中就使用到了条件变量pthread\_cond\_t。

3.自旋锁

互斥锁是一种sleep-waiting锁，即如果想要对资源加锁，却发现已经有锁了，那么线程T2会阻塞等待，当然如果有等待队列的话，此时会将阻塞等待的线程加入等待队列中。但是处理T2的处理器会去处理其它的事务。

自旋锁是一种busy-waiting锁，如果线程T1正在使用自旋锁，而T2也去申请这个自旋锁，那么T2肯定得不到这个自旋锁，而此时，运行T2的处理器core2会一直循环检查锁是否可用，直到获得这个自旋锁为止。

28. sizeof和strlen的区别

sizeof是C/C++中的一个运算符，其作用是返回一个对象或者类型在内存中所占用的字节数。

 注意：sizeof后面如果是类型则必须加括号，如 sizeof(char)；而如果是变量名则可以不加括号，如 sizeof a; 但是建议使用时  均加上括号。sizeof不能返回动态地被分配的数组的大小。

strlen是C语言中的库函数，所在头文件为#include <string.h>其函数原型为unsigned int strlen(char \*s); 其中s为指定的字符串。

strlen只能用char \*作为参数，它求的是字符串的实际长度，方法是从开始到遇到第一个'\0'结束。

1. char str[20] = "0123456789";
2. int a = strlen(str); */\*a = 10\*/*
3. int b = sizeof(str);*/\*b = 20\*/*

上面结果为a = 10，这是因为strlen计算的是字符串的实际长度，以第一个'\0'为结束符；b = 20,这是因为sizeof计算的是分配的数组str[20]所占的空间大小，不受里面存储内容的影响。

1. char \*ss = "0123456789";
2. int i = sizeof(ss); */\*i = 4\*/*
3. int j = sizeof(\*ss); */\*j = 1\*/*
4. int k = strlen(ss); */\*k = 10\*/*

上面结果为i = 4，这是因为sizeof获得的是一个指针的值所占的地址空间，为long int型的，占4个字节；j = 1，这是因为 \*ss 为第一个字符 '0' 所以占一个字节；k = 10，这是因为strlen所获得的是此字符串的实际长度，所以要想获得此字符串的长度，则一定要用strlen。

1. char buffer[] = "Hello";
2. int m = strlen(buffer); */\*m = 5\*/*
3. int n = sizeof(buffer); */\*n = 6\*/*

上面结果为m = 5，这是因为strlen测量的是字符串的实际长度，以遇到的第一个'\0'为结束标志；n = 6，这是因为字符串以'\0'为结束标志，'\0'也需要占用一个字节，所以sizeof测得的结果为6。

29.const static 等

static的作用：

对变量：

1.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。

  1）内存中的位置：静态存储区

  2）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（自动对象的值是任意的，除非他被显示初始化）

  3）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。

 注：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。**

2.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。

 1）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）

 2）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（自动对象的值是任意的，除非他被显示初始化）

 3）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。

注：static修饰全局变量，**并未改变其存储位置及生命周期，而是改变了其作用域，使当前文件外的源文件无法访问该变量**，好处如下：（1）不会被其他文件所访问，修改（2）其他文件中可以使用相同名字的变量，不会发生冲突。**对全局函数也是有隐藏作用。**而普通全局变量只要定义了，任何地方都能使用，使用前需要声明所有的.c文件，只能定义一次普通全局变量，但是可以声明多次（外部链接）。注意：全局变量的作用域是全局范围，但是在某个文件中使用时，必须先声明。

 对类中的：

　1.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(初始化格式： int base::var=10;)，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 。因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的。**

**特点：**

         1.不要试图在头文件中定义(初始化)静态数据成员。在大多数的情况下，这样做会引起重复定义这样的错误。即使加上#ifndef #define #endif或者#pragma once也不行。

         2.静态数据成员可以成为成员函数的可选参数，而普通数据成员则不可以。

         3.静态数据成员的类型可以是所属类的类型，而普通数据成员则不可以。普通数据成员的只能声明为 所属类类型的指针或引用。

    2.成员函数

1. 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。
2. 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。base::func(5,3);当static成员函数在类外定义时不需要加static修饰符。
3. 在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员。因为静态成员函数不含this指针。

**不可以同时用const和static修饰成员函数。**

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。

我们也可以这样理解：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。

const的作用：

1.限定变量为不可修改。

2.限定成员函数不可以修改任何数据成员。

3.const与指针：

const char \*p 表示 指向的内容不能改变。

char \* const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

30.C++内存泄漏检测

1.工具类

Linux下的valgrind可以用于内存泄漏排查，windows下 vs里面有CRT库可以检测内存泄漏

2.对象计数

方法：在对象构造时计数++，析构时--,打印对象的数量

3.运算符重载

重载new/delete运算符，定期打印

31.函数指针

函数指针

一个函数总是占用一段连续的内存区域，函数名在表达式中有时会被转换成该函数所在区域的首地址，这和数组名非常类似。

我们可以把函数的这个首地址（或称入口地址）赋予一个指针变量，使指针变量指向函数所在的内存区域，然后通过指针变量就可以找到并调用该函数。

bool lengthCompare(const string &, const string &);

//该函数的类型是bool (const string &, const string &)。

bool (\*pf) (const string &, const string &);

//pf指向一个函数，该函数的参数是两个const string的引用，返回值是bool类型

1. 使用函数指针

当我们把函数名作为一个值使用时，该函数自动转换成指针。

pf = lengthCompare; //pf指向名为lengthCompare的函数

pf = &lengthCompare; //等价的赋值语句：取地址符是可选的

使用函数指针调用函数：

bool b1 = pf("hello", "goodbye"); //调用lengthCompare函数

bool b2 = (\*pf) ("hello", "goodbye"); //一个等价的调用

bool b3 = lengthCompare("hello", "goodbye"); //另一个等价的调用

我们可以为函数指针赋一个nullptr或者值为0的整型常量表达式，表示该指针没有指向任何一个函数。

2. 函数指针形参

虽然不能定义函数类型的形参，但是形参可以是指向函数的指针。此时，形参看起来是函数类型，实际上却是当成指针使用：

//第三个形参是函数类型，它会自动转换成指向函数的指针

void useBigger(const string &s1, const string &s2, bool pf (const string &, const string &));

//等价的声明：显式的将形参定义成指向函数的指针

void useBigger(const string &s1, const string &s2, bool (\*pf) (const string &, const string &));

//可以直接将函数作为实参使用，此时它会自动转换成指针

useBigger(s1, s2, lengthCompare);

3. 返回指向函数的指针

和数组一样，虽然不能返回一个函数，但是能返回指向函数类型的指针。然而，我们必须把返回类型写成指针形式。  
想要声明一个返回函数指针的函数，最简单的方法是使用类型别名：

using F = int(int \*, int); //F是函数类型，不是指针

using PF = int(\*)(int \*, int); //PF是指针类型

PF f1(int); //正确：PF是指向函数的指针，f1返回指向函数的指针

F f1(int); //错误：F是函数类型，f1不能返回一个函数

F \*f1(int); //正确：显式的指定返回类型是指向函数的指针

32.线程进程的区别与联系

进程是表示资源分配的基本单位，又是调度运行的基本单位。在微内核系统中，真正调度运行的基本单位是线程。因此，实现并发功能的单位是线程。

线程是进程中执行运算的最小单位，亦即执行处理机调度的基本单位。如果把进程理解为在逻辑上操作系统所完成的任务，那么线程表示完成该任务的许多可能的子任务之一。

一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程是操作系统可识别的最小执行和调度单位。

资源分配给进程，同一进程的所有线程共享该进程的所有资源。 同一进程中的多个线程共享代码段(代码和常量)，数据段(全局变量和静态变量)，扩展段(堆存储)。但是每个线程拥有自己的栈段，栈段又叫运行时段，用来存放所有局部变量和临时变量。

33.c语言如何实现多个返回值？

方法一：利用全局变量

定义多个全局变量即可实现多返回值

方法二：传递数组指针

考虑把多个返回值作为数组元素定义成一个数组的形式，并使该数组的地址作为函数的形式参数，以传址方式传递数组参数。

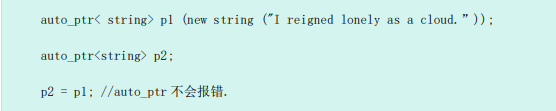
方法三：传递结构体指针

可以考虑把要求返回的数个值定义成一个结构体，然后同样以传递结构体指针方式把结构体的指针传递给形参结构体指针，那么函数中对形参结构体的修改即是对实参结构体的修改，函数被调用后获取的实参结构体成员即为函数的多个返回值。

34.C++智能指针

智能指针的作用是管理一个指针，因为存在申请的空间在函数结束时忘记释放的，造成内存泄漏。使用智能指针可以避免这个问题。智能指针实际上是一个类，当超出类的作用域时，类会自动调用析构函数，析构函数会自动释放资源。

**auto\_ptr**



如果两个auto\_ptr指针指向同一个对象时，当该对象的生存周期结束后，系统会调用析构函数，这样导致的结果是程序对同一个对象删除了2次，造成程序出错。根本原因是这里其实发生的是浅拷贝。

如何解决？

1、通过定义赋值运算符，实现深拷贝，从而两个指针指向不同的两个对象，其中一个对象是另一个对象的副本；

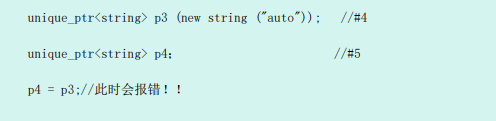
2、将对象的删除所有权设置成唯一。即只允许一个对象拥有删除对象的权力，其他指针只能指向该对象，不享有删除对象的权力。auto\_ptr、unique\_ptr都是用这样的机制，但是后者比前者更加的严格；

3、采用引用计数的方法。当有一个一个指针指向对象时，引用计数加1，当指向该对象的指针销毁时，引用计数减1，当且仅当最后一个指向该对象的指针销毁时，也就是引用计数为0时，对象才能够被删除。智能指针shared-ptr就是采用该机制来实现的。

auto\_ptr导致的问题在于，当所有权转让之后，原来的指针被用来访问原来指向的对象，当程序访问该指针指向的内容时，会发现这是一个悬空指针，程序就会出错，但是如果换成shared\_ptr程序就能正常运行。

**unique\_ptr(替换auto\_ptr)**

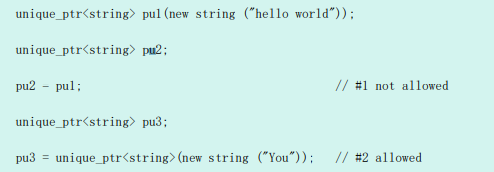
unique\_ptr使用的也是所有权模型，却是独占式的，同一时间只有一个智能指针可以指向该对象。同样是上面那个例子



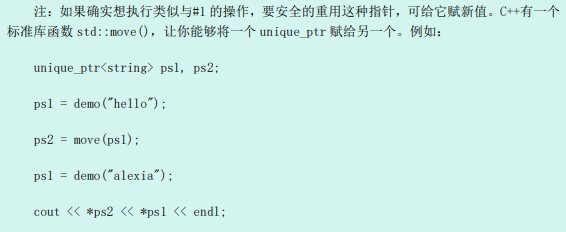
此时编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。

同时，unique还有一个改进的地方：

当程序试图将一个unique\_ptr赋值给另一个时，如果源unique\_ptr是个临时右值，编译器允许这么做；如果源unique\_ptr将存在一段时间，编译器将禁止这么做。



第2种情况不会留下悬挂的unique\_ptr，因为它调用的unique\_ptr的构造函数，该构造函数创建的临时对象在其所有权让给pu3后就会销毁。



35.静态函数和虚函数的区别

静态函数在编译时就已经确定运行时机。虚函数是在运行时动态绑定。虚函数因为用了虚函数表机制，调用时会增加一次内存开销。

36.请你来写一个函数在main函数执行前先运行

**gcc里面:**

#include <stdio.h>

\_\_attribute((constructor)) void before\_main()

{

printf("%s/n",\_\_FUNCTION\_\_);

}

\_\_attribute((destructor)) void after\_main()

{

printf("%s/n",\_\_FUNCTION\_\_);

}

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

{

printf("%s/n",\_\_FUNCTION\_\_);

return 0;

}

37.隐式类型转换

对于内置类型，低精度的变量给高精度的变量赋值时会发生隐式类型转换。

 int ival = 3;

 double dval = 3.14259;

 ival + dval;     // ival被提升为double类型：3.0

对于只存在单个参数的构造函数来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。

需要有默认的构造函数？

38.说说你了解的类型转换

reinterpret\_cast：可以用于任意类型之间的指针之间的转换，对转换的结果不做任何保证。

dynamic\_cast:dynamic\_cast本身只能用于存在虚函数的父子关系的强制类型转换，对于指针，转换失败返回nullptr,对于引用，转换失败会抛出异常。

const\_cast：对于未定义const版本的成员函数，使用const\_cast来去除const引用对象的const,完成函数调用。

const int val = 5;

int test = 5;

//int \*pval = &val; //err: cannot convert from 'const int \* to int \*'

int\* pval = const\_cast<int\*>(&val);

cout << "Before:" << endl;

cout << " val:" << val << endl;

cout << "\*pval:" << \*pval << endl;

\* pval = 10;

cout << "After:" << endl;

cout << " val:" << val << endl;

cout << "\*pval:" << \*pval << endl;

这里使用const\_cast虽然做了强制转换了，实际上改变了内存中该地址对应的值，但是编译器在返回val的值时，其实val已经保存在了编译器中，所以还是返回5

39.RTTI

运行时类型检查，在c++层面主要体现在dynamic\_cast和typeid.

为什么有RTTI机制的产生？

C++是一种静态类型语言，其数据类型是在编译期间就确定的，不能在运行时更改。然而由于面向对象中多态性的要求，c++中的指针或者引用本身的类型，可能与它实际代表的类型并不一致，例如一个基类指针却指向了派生类。有时我们需要将一个多态指针转换为其实际指向的对象的类型，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。

RTTI实现上主要有两个：

1.dynamic\_cast

将基类类型指针或引用安全地转换为派生类类型的指针或者引用。

注意dynamic\_cast只能用于含有虚函数的类。

这种强制转换可以使得一个基类指针原本指向的是派生类，直接将基类指针强转为派生类指针。

如果我们只是需要用基类指针去访问派生类的虚函数还好，不需要强制转换。主要是如果指向派生类的基类指针想访问派生类中除虚函数以外的成员，这时就需要强制类型转换了。

dynamic\_cast转换符只能用于指针或者引用。dynamic\_cast转换符只能用于含有虚函数的类。Dynamic\_cast转换操作符在执行类型转换时首先将检查能否成功转换，如果能够成功转换则转换之，如果转换失败，指针转换失败则返回一个0值，引用失败则抛出bad\_cast异常。

2.typeid

Typeid返回指针或者引用所指的实际类型

Typeid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>

当需要具体的类型信息时，可以通过成员函数来提取。

type\_info类的拷贝构造函数和赋值运算符都是私有的，不允许用户自己创建type\_info的类，唯一要使用type\_info类的方法就是使用typeid函数。

将typeid设置为该类的友元函数，这样typeid就可以访问type\_info类的私有成员，从而创建type\_info类的对象，从而可以创建返回类型为type\_info类的引用。

对于含有虚函数表的类，type\_info在虚函数表的-1项

A\* s = new B();

if (typeid(s) == typeid(B)) {

cout << "B" << endl;

}

else {

cout << "A" << endl;

}

40.虚函数表具体是怎么实现运行时多态的？

子类如果重写父类的虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在vs中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态。

41.请你说一说C语言是怎么进行函数调用的？

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp指针压栈。

42.c++的拷贝构造函数的形参能否是值传递？

不能。调用拷贝构造函数时，首先要将实参传递给形参，这个传递又要调用拷贝构造函数，如此循环，无法完成拷贝，栈会满。

43.map和set的区别？

map和set都是c++关联容器，其底层实现都是红黑树，基本上map和set的操作，都只是转调RB-tree的操作行为。

map和set的区别在于：

1.map中的元素是key-value对，key起到的是索引作用，值则表示与索引相关联的数据；set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字。

2.set的迭代器是const的，不允许修改元素的值；map允许修改value值，但是不允许修改key。其原因是因为map和set是根据关键字来保证其有序性的，如果允许修改key的话，那么首先要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是应该指向改变后的位置。所以stl中将set的迭代器设置为const，不允许修改迭代器的值，而map则不允许修改key值，允许修改value值。

3.map支持下标操作，set不支持下标操作。Map可以用key做下标，map的下标运算符将关键码作为下标去执行查找，如果关键码不在，则插入一个具有该关键码和mapped\_type类型默认值的元素至map中

44.介绍一下STL中allocator

Stl的分配器用于封装STL容器在内存管理上的底层细节。在C++中，其内存配置和释放如下：

New运算分两个阶段：

1.调用：：operator new 配置内存

2.调用对象构造函数构造对象内容

Delete运算：

1.调用对象析构函数

2.调用：：operator delete释放内存

为了精密分工，STL allocator将两个阶段操作区分开来：

内存配置由allocate()负责，内存释放由deallocate()负责；对象构造由construct()负责，对象析构由destroy()负责。

同时，为了提升内存管理的效率，减少申请小内存造成的内存泄漏问题，采用两级配置器，当分配的空间大小超过128B时，使用第一级空间配置器，当分配的空间大小小于128B时，将使用第二级空间配置器。第一级空间配置器直接使用malloc(),realloc(),free()进行内存空间的分配和释放，而第二集空间配置采用了内存池技术，通过空闲链表来管理内存。

45.STL迭代器删除元素

1.对于序列容器vector,deque来说，使用erase(iterator)后，后边的每个元素的迭代器都会失效，但是后面每一个元素都会往前移动一个位置，但是erase会往前移动一个位置，erase会返回下一个有效的迭代器。

2.对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator。

46.vector和list的区别，应用

1.vector

连续存储的容器，动态数组，在堆上分配内存空间

底层实现：动态数组

两倍容量增长：vector增加新元素时，如果超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器失效。

2，list

底层实现是循环链表，在堆上分配空间。每插入一个元素都会分配空间，每删除一个元素都会释放空间。

区别：

vector底层实现是数组，list是双向链表

vector支持随机访问，List不支持。Vector是顺序内存，list不是。Vector在中间节点进行插入删除会导致内存拷贝，list不会。

Vector一次性分配好内存，不够时才会进行2倍扩容；list每次插入新节点都会进行内存申请。

Vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好。

通过查看STL源码可以看到vector有四个成员变量   
\_A   allocator;   
iterator   \_First,   \_Last,   \_End;

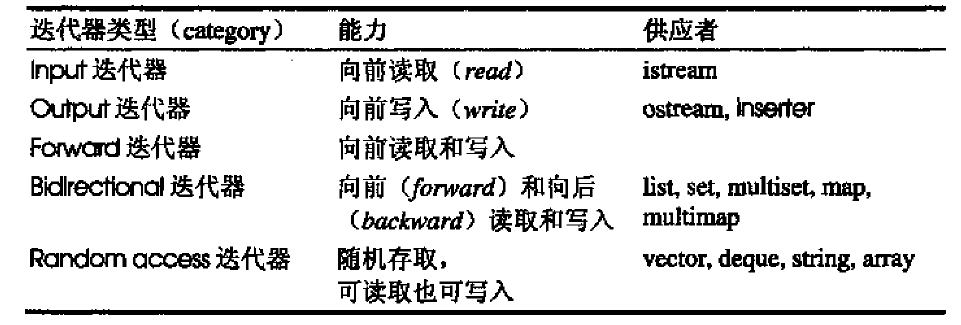
47.STL迭代器作用，有指针为何还要迭代器？

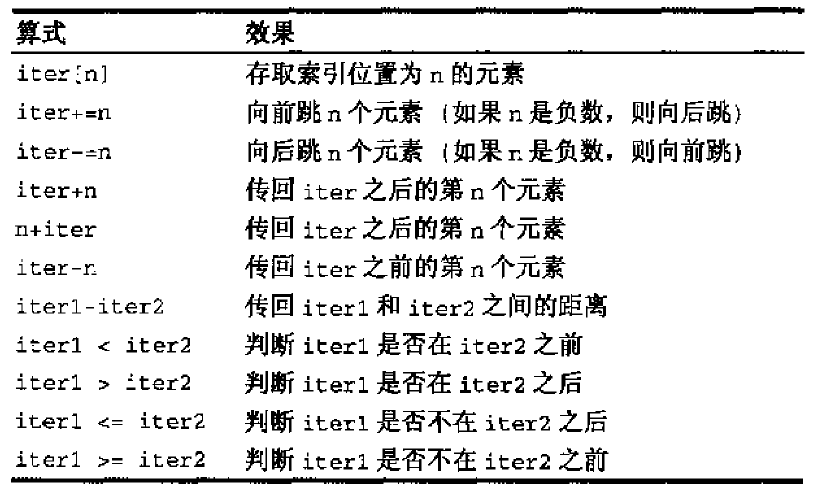
迭代器模式又称之为游标模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又不需要暴露该对象的内部表示。

迭代器和指针的区别：

迭代器不是指针，是类模板，表现的像是指针。他只是模拟了执政的一些功能，通过重载指针的一些操作符，如-> \* ++ --等。迭代器封装了指针，是一个“可遍历STL容器内全部或部分元素”的对象，本质上是封装了原生指针，是指针概念的一种提升，提供了比指针更加高级的行为，相当于一种智能指针，它可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对对象引用而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身。





48.请你回答什么是右值引用，跟左值又有什么区别？

右值引用是c++11中引入的新特性，它实现了转移语义和精确传递。它的主要贡献在于：

1.消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率

2.能够更加简洁明确地定义泛型函数

左值和右值的概念：

左值：能对表达式取地址，或具名对象/变量。一般指表达式结束后依然存在的持久对象。

右值：不能对表达式取地址，或匿名对象。一般指表达式结束就不再存在的临时对象。

根本区别在于左值是可以被寻址的，那么就可以给左值赋值，而右值不可以。

49.c++源文件从文本到可执行文件经历的过程

四个阶段：

1.预处理阶段：对源代码中文件包含关系，预编译语句进行分析和替换，生成预编译文件

主要做了三件事：

（1）文件包含，就是把源程序中include扩展为文件正文，把包含的.h文件找到并展开到#include所在处

（2）条件编译：预处理器根据#if和#ifdef 等编译命令以及其后的条件，将源程序中的某部分包含进来或者排除在外。

（3）宏展开：将文件中使用到定义的宏的地方展开成相应的宏定义

2.编译阶段：将预处理后的预编译文件转换成特定的汇编代码，生成汇编文件

3.汇编阶段：将编译阶段生成的汇编文件转换成机器码，生成目标文件

4.链接阶段：将多个目标文件以及所需要的库连接成可执行文件

50.malloc的原理

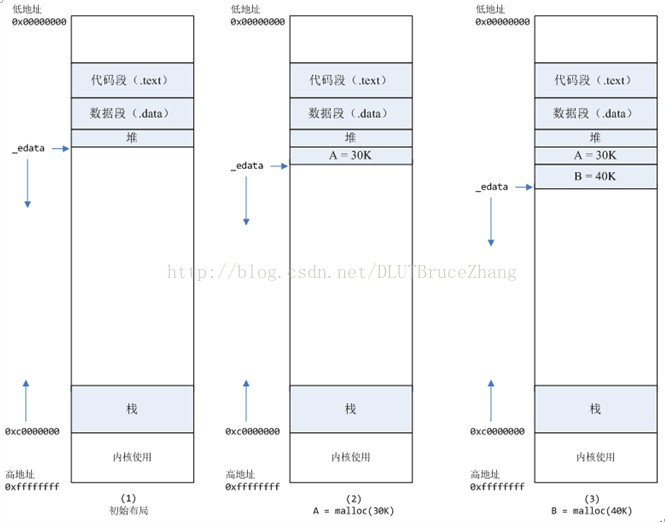
Malloc其采用的是内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。Malloc采用隐式链表结构将堆区分成连续的，大小不一的块，包含已分配的块和未分配的块；同时，malloc采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来

每一个空闲块记录一个连续的，未分配的地址。

当进行内存分配时，malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每一个块的前后块是否已经分配来决定是否进行块合并。

Malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请。其中申请内存小于128K时，会使用系统函数brk在堆区分配，如果申请内存大于128K，会使用系统函数mmap在映射区分配。

内存分配的原理：



1、进程启动的时候，其（虚拟）内存空间的初始布局如图1所示。

      其中，mmap内存映射文件是在堆和栈的中间（例如libc-2.2.93.so，其它数据文件等），为了简单起见，省略了内存映射文件。

      \_edata指针（glibc里面定义）指向数据段的最高地址。   
2、进程调用A=malloc(30K)以后，内存空间如图2：

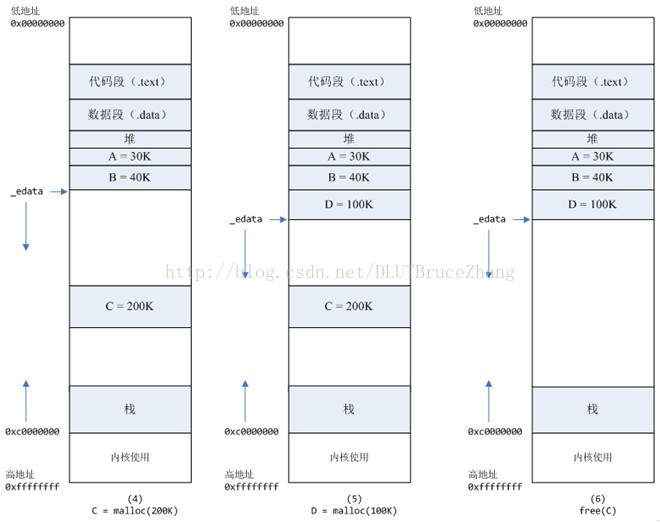
      malloc函数会调用brk系统调用，将\_edata指针往高地址推30K，就完成虚拟内存分配。

      你可能会问：只要把\_edata+30K就完成内存分配了？

      事实是这样的，\_edata+30K只是完成虚拟地址的分配，A这块内存现在还是没有物理页与之对应的，等到进程第一次读写A这块内存的时候，发生缺页中断，这个时候，内核才分配A这块内存对应的物理页。也就是说，如果用malloc分配了A这块内容，然后从来不访问它，那么，A对应的物理页是不会被分配的。   
3、进程调用B=malloc(40K)以后，内存空间如图3。

当malloc小于128K时，使用brk分配内存，brk做的事情就是将\_edata往高地址推，注意进程虚拟地址空间是不对应实际物理空间的，因此malloc时并没有初始化。第一次读/写数据时，引起内核缺页中断，内核才分配对应的物理内存，然后虚拟地址空间建立到物理地址空间的映射。

当malloc大于128K时，使用mmap分配内存，在堆和栈之间找一块空闲内存分配

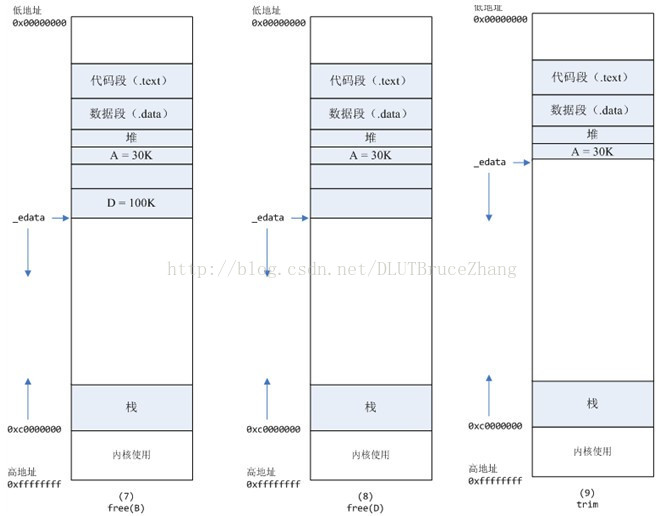


4、进程调用C=malloc(200K)以后，内存空间如图4：

      默认情况下，malloc函数分配内存，如果请求内存大于128K（可由M\_MMAP\_THRESHOLD选项调节），那就不是去推\_edata指针了，而是利用mmap系统调用，从堆和栈的中间分配一块虚拟内存。

Brk分配的内存只有等到高地址的释放了才能释放，而mmap系统调用却是可以直接释放的。

这就是内存碎片产生的原因



**5、**进程调用D=malloc(100K)以后，内存空间如图5；  
**6、**进程调用free(C)以后，C对应的虚拟内存和物理内存一起释放。

7、进程调用free(B)以后，如图7所示：

        B对应的虚拟内存和物理内存都没有释放，因为只有一个\_edata指针，如果往回推，那么D这块内存怎么办呢？

当然，B这块内存，是可以重用的，如果这个时候再来一个40K的请求，那么malloc很可能就把B这块内存返回回去了。   
8、进程调用free(D)以后，如图8所示：

        B和D连接起来，变成一块140K的空闲内存。

9、默认情况下：

       当最高地址空间的空闲内存超过128K（可由M\_TRIM\_THRESHOLD选项调节）时，执行内存紧缩操作（trim）。在上一个步骤free的时候，发现最高地址空闲内存超过128K，于是内存紧缩，变成图9所示。

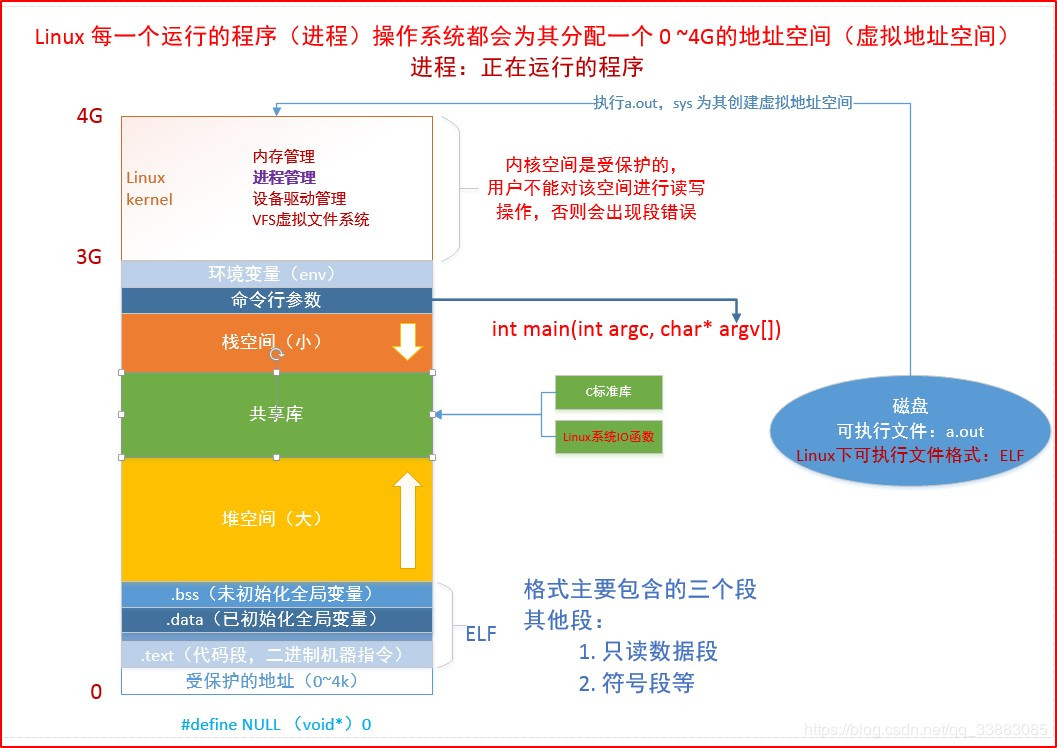
Malloc实质：

维护一个空闲链表，调用malloc函数时，他沿着链表找到一个满足条件的内存块，然后将这个内存块分为两块，一块用于分配内存，另一块剩余的空间返回到空闲链表上面。

调用free函数时，将释放的内存连接到空闲链表中。

最后，空闲链表会被切割成一块一块的小内存片段，如果这时，用户想要申请一个大内存，但是空闲链表上没有满足条件的内存块了，于是malloc就会开始在空闲链表上面检查各内存片段，对他们进行内存整理，将相邻的小空闲块合并成较大的内存块。

51.进程地址空间



1.保留区（受保护的地址）

        保留区即为受保护的地址，大小为0~4K，位于虚拟地址空间的最低部分，未赋予物理地址（不会与内存地址相对应，因此其不会放任何内容）。任何对它的引用都是非法的，用于捕捉使用空指针和小整型值指针引用内存的异常情况。大多数操作系统中，极小的地址通常都是不允许访问的，如NULL。C语言将无效指针赋值为0也是出于这种考虑，因为0地址上正常情况下不会存放有效的可访问数据。将指针赋值为0，意味着该指针将永远不会被使用，从而不会出现野指针情况。#define NULL 0 与 #define NULL (void\*)0   在C语言中是等效的，而在C++中，只能用#define NULL 0，后面 #define NULL (void\*)0的使用会出错。

2.代码段

        代码段也称正文段或文本段，通常用于存放程序执行代码(即CPU执行的机器指令)。一般C语言执行语句都编译成机器代码保存在代码段。通常代码段是可共享的，因此频繁执行的程序只需要在内存中拥有一份拷贝即可。代码段通常属于只读，以防止其他程序意外地修改其指令(对该段的写操作将导致段错误)。某些架构也允许代码段为可写，即允许修改程序。

3.数据段（.data段）

        数据段通常用于存放程序中已初始化的全局变量和静态局部变量。数据段属于静态内存分配(静态存储区)，可读可写。由于全局变量未初始化时，其默认值为0，因此值为0的全局变量位于.bbs段（不位于数据段）。对于未初始化的局部变量，其值是不可预测的。注意：在代码段和数据段之间还包括其它段：只读数据段和符号段等。

4..bbs段

        该段用于存放未初始化的全局变量和静态局部变量，包括值为0的全局变量。 数据段和.bbs段又称为全局数据区，前者初始化，后者未初始化。

        ELF段包括：代码段、其它段（只读数据段和符号段等）、.data段（数据段）和.bbs段，都属于可执行程序部分。

5.堆空间

        new( )和malloc( )函数分配的空间就属于对空间，用于内存空间的分配，其从下往上。  堆用于存放进程运行时动态分配的内存段，可动态扩张或缩减。堆中内容是匿名的，不能按名字直接访问，只能通过指针间接访问。当进程调用malloc(C) 和new (C++)等函数分配内存时，新分配的内存动态添加到堆上(扩张)；当调用free(C)/delete(C++)等函数释放内存时，被释放的内存从堆中剔除(缩减) 。

6.内存映射段（共享库）

        此处，内核将硬盘文件的内容直接映射到内存, 任何应用程序都可通过Linux的mmap()系统调用请求这种映射。内存映射是一种方便高效的文件I/O方式， 因而被用于装载动态共享库。如C标准库函数（fread、fwrite、fopen等）和Linux系统I/O函数，它们都是动态库函数，其中C标准库函数都被封装在了/lib/libc.so库文件中，都是二进制文件。这些动态库函数都是与位置无关的代码，即每次被加载进入内存映射区时的位置都是不一样的，因此使用的是其本身的逻辑地址，经过变换成线性地址（虚拟地址），然后再映射到内存。而静态库不一样，由于静态库被链接到可执行文件中，因此其位于代码段，每次在地址空间中的位置都是固定的。

7.栈空间

        用于存放局部变量（非静态局部变量，C语言称为自动变量），分配存储空间时从上往下。栈和堆都是后进先出的数据结构。

8.命令行参数

        该段用于存放命令行参数的内容：argc和argv。

9.环境变量

        用于存放当前的环境变量，在Linux中用env命令可以查看其值。

52.段错误

段错误就是指访问的内存超出了系统所给这个程序的内存空间。

发生段错误的几类原因：

1.访问不存在的内存地址

2.指针越界访问

3.某个函数内的数据开的过大，造成爆栈

等等

53. C++11 std::atomic

atomic提供了原子操作，这是c++11中的新特性，并且提供了多种原子操作数据类型，例如atomic\_int,atomic\_bool等。如果我们在多个线程中对这些类型的共享资源操作，编译器保证这些操作都是原子的。

std::atomic对Int char bool等数据结构进行了原子性封装，在多线程环境下，对std::atomic对象的访问就不会造成竞争-冒险。利用std::atomic可以实现数据结构的无锁设计。

Atomic类是如何实现原子操作的呢？

Java里面了解到的CAS机制实现的原子操作：

**CAS机制**

例如多个线程都想要对同一个变量进行自增操作

假设初始时内存地址中保存的值为10

线程1先读取到10，保存下来这个旧的预期值=10，待更新的值为11，准备提交更新

但是这时，线程2却率先完成了自增操作，内存地址中已经变为了11

这时线程1准备提交更新，会检查当前内存地址中保存的值是不是等于旧的预期值

发现不相等，则会提交失败，失败了就重新尝试，如果运气比较好，提交成功了，则变量更新为12

**C++ atomic实现的机制：**

总线锁：

处理器提供一个LOCK信号，当一个处理器在总线上输出次信号时，其它的处理器请求将会被阻塞，该处理器可以独占内存。C++在编译期间生成的汇编代码中，如果某一条指令前面加了前缀LOCK，经过汇编以后的机器码就会使得CPU在执行这条指令的时候把总线锁住，从而使得同一总线上别的cpu无法通过总线访问内存。

缓存锁：

由于总线锁阻塞了其他处理器与内存之间的通信，而实际上输出LOCK信号的CPU可能只需要锁住特定的一块内存区域，因此总线锁的开销较大。

缓存锁定指的是当某个CPU对缓存数据进行更改时，会通知缓存了该数据的其他CPU抛弃缓存的数据或者从新从内存读取。

54.reactor模型

reactor模型要求主线程只负责监听文件描述符上是否有事件发生，有的话就立刻将该事件通知工作线程，除此之外主线程不做其他实质性的工作，读写数据，接受新的连接以及处理客户请求全部在工作线程中完成。

55.STL的内存优化

两级配置器

第一级配置器：当想要分配的是大于128字节的空间，使用malloc直接分配内存

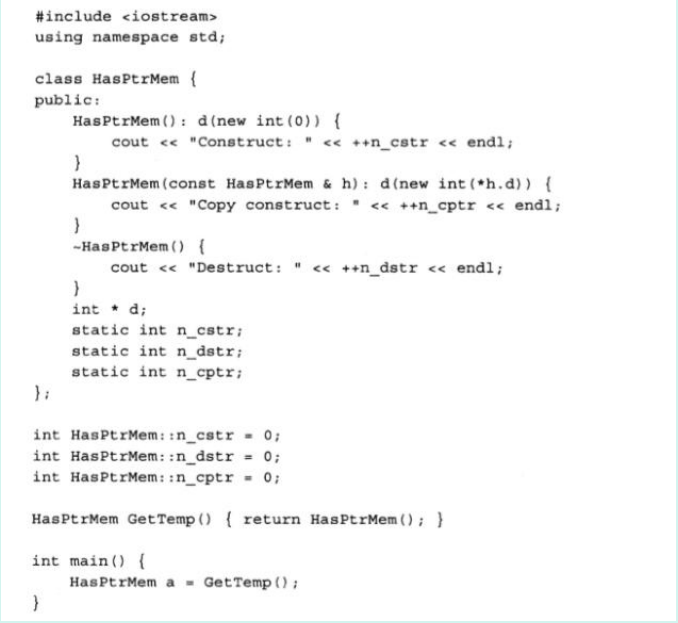
第二级配置器：为了避免内存碎片，进行了更加细致的优化

管理了一个内存池。先配置了一块较大的内存（4K），并维护对应的16个空闲链表，如果有相同大小的内存需求，则直接从空闲链表中取得，如果有小块内存被释放，则由配置器回收到空闲链表中。

56.理解移动语义

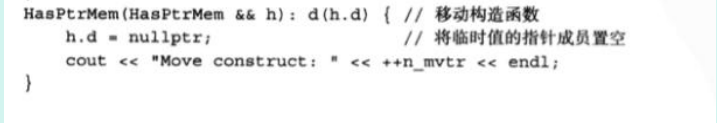
编译器默认的拷贝构造函数是浅拷贝，浅拷贝的问题在于仅仅只是简单的指针赋值，这会造成在析构时发生两次析构，而空悬指针析构会出错。

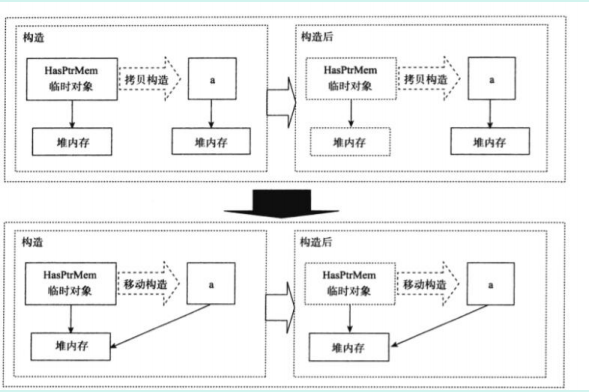
一般会自定义拷贝构造函数完成深拷贝，先判断待拷贝的指针和原指针是不是一个指针，如果是的话，就不需要拷贝，不是的话，要分配新的内存，并进行内容的拷贝。



例如这里调用成员函数GetTemp,其返回值类型为HasPtrMem,如果我们定义了深拷贝的拷贝构造函数，那么在调用该函数时需要调用两次拷贝构造函数。第一次是生成GetTemp函数返回时的临时变量，第二次是将该返回值赋值给main函数中的变量a。与此同时需要调用三次析构函数来释放内存。

上述，使用临时变量构造a时会调用拷贝构造函数分配堆内存，而临时对象在语句结束后会释放它所使用的堆内存。这样重复申请和释放内存，在申请内存较大时候会严重影响性能。因此c++使用移动构造函数，从而保证使用临时对象构造a时不分配内存，从而提高性能。





所谓的移动构造，就是把原来是临时对象申请的堆内存，给复用到，然后把临时对象中指向动态分配的堆内存的指针置为空，从而保证临时对象在析构时不会释放堆内存。

完美转发

完美转发是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另一个函数。保留了原来的左右值属性。

move是怎么实现的？

1.Move的形参类型是一个右值引用类型，通过引用折叠，你传入的是个左值，就变成了左值引用，传入的是个右值，就变成了右值引用。

传入的实参是一个左值引用，则 T 推导为左值引用， t引用叠加为左值引用

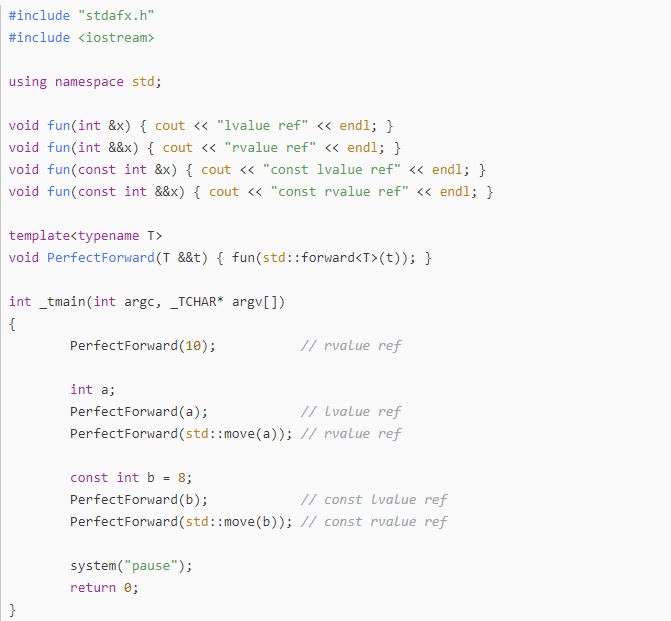
传入的实参是一个右值引用，则T推导为非引用类型，t引用折叠为右值引用

传入的实参是一个左值（非引用），则T推导为左值引用，t引用折叠为左值引用

2.通过remove\_reference\_t，通过模板特化的方式，通通返回他们的不带引用类型的T

3。通过static\_cast来强制转换一个左值为一个右值引用，从而可以返回右值引用

forward是怎么实现的？



完美转发：在模板函数编程中，有一种场景是需要将模板参数传递给模板函数内调用的函数。

在完美转发的场景下。

完美转发中，模板函数形参一定是一个右值引用，而在函数内，形参一定是一个左值引用（因为已经取到地址了），此时forward拿到的数据类型一定是一个左值引用。关键的问题是，我们想知道原来的实参类型是一个左值引用还是一个右值引用。

如果完美转发的实参是一个左值或者左值引用，模板参数T被解析为左值引用。

如果完美转发的实参是一个右值引用，模板参数T被解析为非引用类型。

template<class T>

T&& forward(remove\_reference\_t<T>& t) {

return static\_cast<T&&>(t);

}

template<class T>

void forward(remove\_reference\_t<T>& t);

//使用forward，必须有模板参数T

template<class T>

void func(T&& t) {

return func2(std::forward<T>(t));

}

57.Lambda表达式

Lambda表达式定义了一个匿名函数，把函数看作对象，lambda表达式可以像对象一样使用，可以将其赋给变量或者当作参数使用。

[capture](parameters) mutable ->return-type{statement}

例如：

[](int a, int b) -> int { return a + b; };

1.capture:捕捉列表，位于最开始处，【】括起。编译器根据该对[]判断接下来的代码是否是lambda函数。捕捉列表能够捕捉上下文中的变量供lambda使用。

2.parameters:参数列表。与普通的参数列表一致。如果不需要参数传递，则可以连同()省略。

3.mutable：mutable修饰符。默认情况下,lambda函数总是一个const函数，mutable可以取消其常量性。

4.->return\_type: 返回类型；如果不需要返回类型，连同->符号可以一起省略。在返回类型明确的时候，也可以让编译器对返回类型进行推导。

5.{statement}：函数体。内容与普通函数一致。可以使用参数，也可以使用捕获变量

// 定义一个简单的lambda表达式

auto basicLamda = [] {cout << "hello world!" << endl; };

basicLamda(); // 调用

// 如果需要参数，那么就要像函数那样，放在圆括号里面，如果有返回值，返回类型要放在->后面，即拖尾返回类型

// 当然你也可以忽略返回类型，lambda会帮你自动推断出返回类型

auto add = [](int a, int b) -> int {return a + b; };

int result = add(4, 5);

cout << "result = " << result << endl;

// 自动推断出返回类型

auto multiply = [](int a, int b) {return a \* b; };

int result1 = multiply(5, 6);

cout << "result1 = " << result1 << endl;

捕获列表详解:

C++变量传递有传值和传引用两种方式。

（1）[]//没有定义任何变量

（2）【x,&y】//x以传值方式传入，y以引用方式传入

（3）【&】//任何使用到的变量全部以引用方式

（4）【=】//任何使用到的变量全部以值传递引用

（5）【&，x】x以传值方式引用，其余的全部以引用方式引用

（6）【=，&z】z以引用方式使用，而其余的都以值传递方式使用

58.auto相关

C语言里面也有auto关键字，在C中使用auto是声明一个变量为自动变量。

C语言相关关键字解析：

1.auto

声明变量的生存期为自动。生存期设置为自动的意思是：比如不在类内，结构体内，枚举内，联合内，函数中的变量视为全局变量，而在函数中定义的变量视为局部变量。所有的变量都默认就是auto的。

2.register

这个关键字就是命令编译器尽可能将变量存储在CPU内部存储器中而不是通过内存中，以提高效率。

3.static

Static变量存储在静态区。当用static来修饰全局变量时，改变了这个变量的作用域，使得不能被别的程序extern,仅仅把该变量限制在了当前文件中。

4.const

….

5.volatile

表明某个变量可能在外部被改变，优化器在用到这个变量时，必须每次都小心地读取这个变量的值，而不是使用保存在寄存器里面的备份。

**C++ auto关键字**

实现就是通过模板类型推断

例如：

int a=5;

auto c=a;

实际上利用了模板推断：

template<typename T>

void func\_for\_c(T param);

func\_for\_c(a);

59.函数参数传递的时候什么时候用指针？什么时候用引用？

1.需要返回函数内局部变量的内存的时候用指针。例如我在函数内部开辟了一块内存，这时使用指针传参，用完要记得释放指针，不然会内存泄漏。返回局部变量的引用没有意义

2。对栈空间比较敏感的时候，例如递归的时候用引用，使用引用传递不会创建临时变量，开销较小

3.类对象作为参数传递的时候使用引用，这是c++类对象传递的标准方式

传值方式：直接开辟一块和实参一样的空间，里面存放了跟实参一样大小的值。

传指针方式：传递的是地址

传引用方式：引用是别名。引用的本质还是传递的是指针，但是这是一种间接寻址。因为指针容易被改变，就找不到原来的地址了，而引用是对指针的封装，做法上更加安全。

60.堆快一点还是栈快一点？

栈快一点。因为操作系统对于栈提供了在底层的支持，会专门分配寄存器存放栈的地址，栈的入栈出栈操作也相对简单，并且有专门的指令执行。而堆的操作是由C/C++函数库提供的，在分配内存时需要一定的算法来寻找合适的内存。访问堆上的内存时，需要两次访问，第一次访问指针，第二次根据指针指向的地址来访问内存。

61.malloc/free new/delete

Malloc/free是c的库函数，是已经编译好的代码。对于非基本数据类型的对象，对象的创建不仅需要分配内存空间，还要调用构造函数进行初始化，所以需要new,delete的。所以只能对于new/delete这样的运算符进行重载等等。

62.c++和python的区别？

Python是一种脚本语言，是解释执行的，而c++是编译语言，是需要在编译后在特定平台上面运行的。Python使用缩进来区分不同的代码块，而c++使用花括号来区分。

C++对于变量需要先定义变量的类型再使用，而python的基本数据类型只有数字，布尔值，字符串，列表，元组等

Python库比较多

63. vector和list的实现

Vector拥有一段连续的内存，list是双向链表，内存上面不连续。Vector因为内存连续，能够支持+ += <等操作符，list的内存空间是可以不连续的，不支持随机访问，支持+ ，+=，<等。

因为push\_back()时需要考虑内存不够时要扩容，这个过程比较浪费时间。

64.零拷贝

零拷贝指的是在进行操作时，避免CPU从一处存储拷贝到另一处存储。在Linux中，我们可以减少数据在内核空间和用户空间的来回拷贝，比如通过调用mmap()来代替read()

65.动态链接是如何实现的？

函数名就是一个内存地址，调用函数就是压入参数，保存返回地址，然后跳转到函数名指向的代码即可。

66.linux下gdb调试程序

首先使用gcc g++编译程序时带上 -g参数

然后使用gdb 可执行文件 就可以运行程序了，也可以使用gdb program core 来结合core文件调试程序

1.启动gdb后可以使用l命令来查看代码

2.使用 b 15命令可以对第15行打断点 info break 查看断点信息

3.r命令表示运行程序

4.n命令单步调试

5.bt查看函数堆栈

6.p i表示打印i变量的值

7.finish表示退出函数

8.q表示结束调试

67 STL使用场景总结：

需要大量的存取，需要随机访问，但是不需要高效的插入和删除效率，使用vector

需要大量的删除和插入，不关心存取，使用List

关心存取，需要在两端进行插入和删除，使用deque

键值对，一对一映射，采用map

68. reinterpret\_cast和static\_cast转换的不同点

class A {

    public:

    int m\_a;

};

class B {

    public:

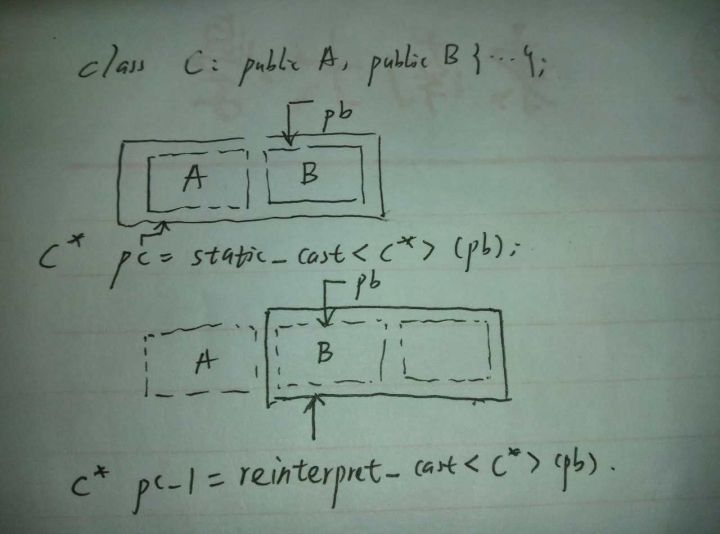
    int m\_b;

};

class C : public A, public B {};

C c;

printf("%p, %p, %p", &c, reinterpret\_cast<B\*>(&c), static\_cast <B\*>(&c));



static\_cast在多重继承中会计算父子指针转换的偏移量