<http://www.cplusplus.com/reference>

推荐书籍： 《c++ primer》《stl源码剖析》《深度探索C++对象模型》

面试只需要： 《c++ primer》 《effective c++》

1. 如果不开启对C++11的支持的话-std=c++11或者-std=c++0x，那么我用的是C++98.

0.2 edit-compile-debug就是用printf，std::cout去调试。需要能够保证每一次输出的时候及时刷新缓冲区内数据。

0.5.C允许从void\*隐式转换到其它的指针类型，但C++不允许

//下面这个C程序居然可以,而C++不行

#include "stdio.h"

#define XNAME(n) t##n

int main(void)

{

for(int i1 = 0;i1 < 3;i1++) {

int XNAME(i1);

XNAME(i1) =i1;

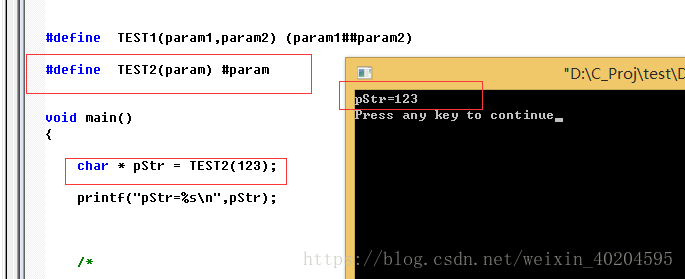
printf("%d", XNAME(i1));

}

return 0;

}

还有一种宏定义:#的作用就是讲#后面的宏参数进行字符串的操作，也就是将#后面的参数两边加上一对双引号使其成为字符串



1. 变量命名：



public静态成员前面还是不加了，函数前面也没有

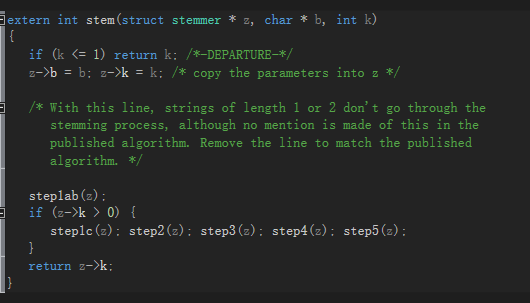
2.//这种空格写法真好

camController->setLookSpeed( 1870.0f );

camController->setCamera(camera);

1. 函数名应该叫showAnswer而不是answer，函数是一种过程行为肯定名字里面要有动词。

如果是restful风格，函数名字是answer，并且需要一个参数来指明操作行为：（不推荐，不然一个函数代码会因为分支变得很长）



上图中step2这种取名我曾经也试过，但是如果中间插入或者删除了一个步骤，名字改动起来就会很麻烦。

4.switch 中的序号常量右边应该来一个注释。或者使用枚举、const、#define等常量实现。

5.专门对useage这种打印函数进行封装

char \*\*p=argv; 通过while(\*p)来访问一个个传入参数

while(\*p) {

if(strcmp(\*p,"-v")==0)

{

printf("Socket data transport tool.\r\nVersion:%s\r\n",VERSION);

p++;

continue;

}

}

6.东西长久放在缓冲区中不刷新有什么坏处？如果程序crash，那么输出就留在了输出缓冲区了，会对程序错误的地方产生误判。

7.对于char：Most systems, including x86 GNU/Linux and Microsoft Windows, use signed char, but those based on PowerPC and ARM processors typically use unsigned char. C++14好像动手改了。

有符号整数的溢出是不确定的，大概是因为符号位能不能改的问题。

有的编译器把数组的赋值当作扩展而允许，所以还是不建议使用。不具有可移植性？（使用一样的编译器就好）编译器会往里面加入特定代码，所以生成的执行文件在同平台上没问题，但是代码不可移植。但是大家喜欢这么用。

对浮点数使用!是未定义的

cout << i++ << i; 没有顺序点，结果未定义

8. 空语句的使用需要引起别人的注意，注释。

9. #include<bits/stdc++.h>包含C++中所有的头文件 但是增加了一定的编译时间

11. 成员函数empty()是个是否问题，是就返回1，否就返回0。

12. 几个写程序类型的规则：①不会是负的就加上unsigned，用了就要小心比较,--等；③注意加减乘除以及绝对值可能引起超范围的现象；④用double而不用float

13. 初始化变量（而且编译器默认初始化不一定就会是0）

15.这样子的函数头：

inline const string &

shorterString(const string &s1, const string &s2)

16. 类中类型别名的定义放在开头，这样之后的都可以看见。

17. 宏名不能以数字开头：#define 1\_H 定义规则与变量一样

C语言限制了变量命名长，因为编译器解析的时候使用了固定长度的数组

1. 函数tie一样的设计思想，如果有参数就会改变内部内容，如果没有就会返回当前的内容。
2. 这里的if确实是应该的：

void print(const char \*cp)//实现遍历所以就不用传大小，要传的话就是size\_t

{

if (cp) // if cp is not a null pointer

while (\*cp) // so long as the character it points to is not a null character

cout << \*cp++; // print the character and advance the pointer

}

函数都应该检查传入参数的值

20. 给函数传参如果需要转换类型，这是不是意味着函数类型设计不好。

21. 对于一个商品类，我们要用平均价格，每一个对象都要有自己的价格，所以整体的平均价格应该是static类型。

22. Advice: Use Private Utility Functions for Common Code。

23.

deque<string> input;

for (string str; cin >> str; input.push\_back(str))

;//null loop

24. const与int算是类型声明倒是可以互换。但是默认const在前。

const可以在类名后面。

25. 运算符重载函数的operator与符号之间可以有空格。

1. 变量按功能分类注释：

如果只是为了一个函数，推荐使用static变量

全局变量可以和头文件写在一起

//为了皇后算法的全局

int num =8;

int \*xPoint;

bool \*\*chess;

//线程控制

bool threadState;

//控制窗口的初始位置（确保那段代码只被执行一次）

bool wndPos;

一种不错的布局管理

#include <stack>

using std::stack;

#include <string>

using std::string;

#include <iostream>

using std::cout;

using std::endl;

27.

if(expression == 0 ) <=> if(expression)

if(expressioin != 0 ) <=> if(!epression)

if(a<b && c>d) </=> if(a<b & c>d) 因为顺序点的存在

if(a && b)</=> if(a&b)因为符号左右表达式的值并不是限制在0和1。

28.《C与指针》说：使用逗号来消除多余的代码。

29.存根：写一个空函数（我见过有的人会有一句printf的消息等等），为以后的代码提前占好位置。

30.windows API中使用的变量前缀

b 表示byte

w 表示word

dw 表示dword

h 表示句柄（也是一种要回收的资源）

lp 表示指针

sz 表示以0结尾的字符串

lpsz 表示指向0结尾的字符串的指针

f 表示浮点数

st 表示一个数据结构

main\_i这样的名字适用于函数中的临时变量。觉得没有必要

31. 结构的自引用现象：

struct SELF\_REF {

int a;

struct SELF\_REF \*point;

}

34. #include "error.h"是很不好的，因为你不知道它是本地文件还是有人傻逼了用这样去表示库文件（首先搜索当前路径中是否存在这个文件）。所以使用#include ""最好能加上路径。

35. using namespace std; 这样子的语句容易造成命名空间的污染。编译器会给出警告，甚至有的编译器会在之后随意选择一个，同一个项目不同时候运行被选择的对象不同，于是代码就会时好时坏。https://blog.csdn.net/ipmux/article/details/17376297

36. unsigned short int 是unsigned short在gcc内部的称呼。

37. 宏定义与函数的穿插：

printf("abnormal termination,signal number =%d%s\n",

WTERMSIG(status),

#ifdef WCOREDUMP

WCOREDUMP(status)?"core file generated" : "");

#else

"");

#endif

宏定义实现模板函数：

\_\_TYPE\_\_ 是集成自一个基类的父类

#define CREATE\_FUNC(\_\_TYPE\_\_) \

static \_\_TYPE\_\_\*create() \

{ \

    \_\_TYPE\_\_ \*pRet = new \_\_TYPE\_\_(); \

    if (pRet && pRet->init()) \

    { \

        pRet->autorelease(); \   调用是基类函数

        return pRet; \

    } \

    else \

    { \

        delete pRet; \

        pRet = NULL; \

        return NULL; \

    } \

}

mallloc的一个常见错误就是忘记了检查，所以

#define malloc

#define MALLOC(type, num) (type \*)my\_malloc( (num)\*sizeof(type) )

通过这样禁止在本文件中使用malloc，同时通过链接来引入自己的带检查的malloc。

#define max(a,b) (a)>(b)?(a):(b)

#if defined(\_\_GNUC\_\_) && ((\_\_GNUC\_\_ >= 5) || ((\_\_GNUG\_\_ == 4) && (\_\_GNUC\_MINOR\_\_ >= 4))) \

|| (defined(\_\_clang\_\_) && (\_\_clang\_major\_\_ >= 3)) || (\_MSC\_VER >= 1800)

#define DISALLOW\_COPY\_AND\_ASSIGN(TypeName) \

TypeName(const TypeName &) = delete; \

TypeName &operator =(const TypeName &) = delete;

#else

#define CC\_DISALLOW\_COPY\_AND\_ASSIGN(TypeName) \

TypeName(const TypeName &); \

TypeName &operator =(const TypeName &);

#endif

#define CC\_SAFE\_DELETE(p) do { delete (p); (p) = nullptr; } while(0)

CC\_SYNTHESIZE(int, num, Num);

上面之所以使用while(0)循环来包裹语句块：<http://www.cnblogs.com/mfmdaoyou/p/6849274.html> 我的话估计会使用逗号表达式

#define CC\_SYNTHESIZE(varType, varName, funName)\

protected: varType varName;\

public: virtual varType get##funName(void) const { return varName; }\

public: virtual void set##funName(varType var){ varName = var; }

auto closeItem = MenuItemImage::create("CloseNormal.png",

"CloseSelected.png", CC\_CALLBACK\_1(HelloWorld::menuCloseCallback, this));

using std::bind;

using std::placeholders::\_1;

using std::placeholders::\_2;

using std::placeholders::\_3;

#define CC\_CALLBACK\_0(\_\_selector\_\_,\_\_target\_\_, ...) bind(&\_\_selector\_\_,\_\_target\_\_, ##\_\_VA\_ARGS\_\_)

#define CC\_CALLBACK\_1(\_\_selector\_\_,\_\_target\_\_, ...) bind(&\_\_selector\_\_,\_\_target\_\_, \_1, ##\_\_VA\_ARGS\_\_)

#define CC\_CALLBACK\_2(\_\_selector\_\_,\_\_target\_\_, ...) bind(&\_\_selector\_\_,\_\_target\_\_, \_1, \_2, ##\_\_VA\_ARGS\_\_)

#define CC\_CALLBACK\_3(\_\_selector\_\_,\_\_target\_\_, ...) bind(&\_\_selector\_\_,\_\_target\_\_, \_1, \_2, \_3, ##\_\_VA\_ARGS\_\_)

39.5. 通过宏定义控制log输出：

宏定义与可变参数：<https://www.cnblogs.com/caosiyang/archive/2012/08/21/2648870.html>

补充对于 #define LOG(format, ...) fprintf(stdout, format, ##\_\_VA\_ARGS\_\_)当LOG函数只有一个实参传入时候，我以为fprintf后面中间有一个逗号导致报错，但是实际上因为使用了##，最后一个逗号也被消去了。

C可变参数函数（stdarg.h）几个函数是借助宏定义进行重命名，基于堆栈操作，所以必须要提供正确的类型。那么要向能够将多种类型输出，不论C/C++必须要类似于fprintf的实现。

C：

#include <cstdio>  
#define DEBUG  
  
#ifdef DEBUG  
#define LOG(target,format, ...) fprintf(target, format, ##\_\_VA\_ARGS\_\_)  
#else  
#define LOG(target,format, ...)  
#endif

#define DEBUG **true**#if DEBUG  
#define log(target,format,...) fprintf(target,format,##\_\_VA\_ARGS\_\_)  
#else  
#define log(target,format,...)  
#endif

可以看到虽然是true，但是编译器确实在运行的时候选择了else。除非将true改成常量1，或者将#if后面的条件改成DEBUG==true。考虑到true本身也是宏定义，建议使用常量1.

C++：

因为类型不一定一样，所以不能使用std::initializer\_list。所以基本不能够通过<<实现，必须要对内容输出进行类似于C语言格式化。

所以综上来讲还是C的输出输入比较有实用性，C++虽然显的更加简单。不过如果能够确保输出类型都是或者都能够被转换成string，那么还是可以结合宏定义：

总头文件：

#ifndef TINY\_GLOBAL\_H  
#define TINY\_GLOBAL\_H  
  
#define DEBUG  
  
#endif //TINY\_GLOBAL\_H

头文件：

#ifndef TINY\_FUNCTION\_H  
#define TINY\_FUNCTION\_H

#include "global.h"  
#include <iostream>  
#include <initializer\_list>  
**using** std::cout; **using** std::flush;  
**using** std::ostream; **using** std::initializer\_list; **using** std::string;  
  
**void** log(ostream &o, initializer\_list<string> lst);  
  
#endif //TINY\_FUNCTION\_H

源文件:

#include "function.h"  
  
**void** log(ostream &o, initializer\_list<string> lst) {  
#ifdef DEBUG  
 **for**(**auto** beg =lst.begin();beg != lst.end(); ++beg)  
 o << \*beg << " ";  
 o << flush;  
#else  
#endif  
}

40.这些数字一旦要改动是真的烦，可以尝试使用常量一个连一个

这种连续的数字比较适合在类里面定义enum，直接就是const常量。



41.



42. 将数组某一个位置四种方向上的遍历，使用for循环，x、y上的变化存储到数组中。而不是使用if-elseif-eles来编写代码。

43.C语言库函数基本没有异常，一旦产生会直接导致程序退出

44. 初始化 char string[10] =""; 在gcc中这等价于char string[10]={0};

45. 对大小写字母之间的转换等 算术运算 ... 优化逼急了，可以转换为位运算

46.

《C++编程规范》

1.由于不同编辑器设置Tab与空格的转换关系设置不同，当没有使用Tab的源代码在不同的非零转化下就会有不同的表现，不过一般是没有关系的，毕竟这设置一般不设一般就是4的转化。最保险并适合我的的就是进行1：4的Tab与空格转化。

2.尽可能去除警告：

使用无法修改的库头文件可能包含引起警告（可能良性）的构造。诸如此类如果是良性的警告就可以用编译器控制进行暂时的警告消除：

#pragma warning(push)

#pragma warning(disable:4512)

#pragma warning(disable:4180)

#include <boost/lambda/lambda.h>

#pragma warning(pop)

对于未使用的函数参数：这个参数由于未被使用所以没准会被编译器报警告，这时候据说把参数的变量名注释掉即可没有警告。或者对变量进行一次无意义的使用即可。

3.书上有个建议是在switch中用case把所有可能举完，然后再default中进行assert("shoult not get here!")....

4.使用自动构建系统；一次按键就可以把源文件翻译为软件。构建有两种：增量构建：只构建上次构建以来变化的部分。另一种是完全构建。

5.版本控制

7.亮出自己的代码同时阅读同事的代码进行审查。

8.不是必要的话，重要程度上，可读性优于算法。

9.进程除了死锁以外还有活琐(livelock)：两个进程因为对方变化而互相变化。

10.在类里面有些时候可能要将复制运算符声明为private但是不定义从而达到禁用的目地（没有必要定义为private）。还有=delete

13.在需要表示符号常量或者受限整数时可以考虑使用enum。

14.如果经常用到dynamic\_cast或者static\_cast就表示类的接口设计有问题。

15.在C++中几乎不需要用宏，可以用const或者enum来定义常量，用inline来避免一些函数的调用开销，用template指定函数与类型系列，用namespace避免名字冲突。

宏还可重命名函数、重定义函数（上面的malloc）

16.有两种错误最难解决：第一是宏，第二是模板错误。

17.确实有不少人都推荐：最好不用宏，用了就尽可能快用#undef去除。

书上说：宏不要放在通用头文件当中。

宏能认识的仅小括号和大括号，对于模板的尖括号不认识，然后进行匹配。所以对于MACFO(Foo<int, double>)宏会认为传入的是Foo<int和double>。

21.不要过分使用try：优先使用析构函数进行自动清除而避免使用try代码块。

22.标准算法对可读性的提升。

23.在不同的编译单元中定义两个名字空间级的对象时，先调用哪一个对象的构造函数是不确定的。所以他们之间不要相互依赖。在任何名字空间对象的初始化中，不能假设其他编译单元中定义的任何其他对象都已经初始化。据说在使用构造函数之前，就已经用0静态初始化过了。

24.我认为相互依赖这个问题其实应该交给编译器解决。

如果能使用前向声明去实现，那就不用去把类的定义#include。适用的场景就是两个类互相是对方成员，可以用前向声明去打破依赖。

25.一开始经过模板定义的时候会把它声明一下，然后进行实例化的时候再回到定义处细读内部一些情况进行定义。函数的话就要是用到用到再细读。（模板类中的成员函数就是这样，由于没有声明那就是用到了才进行细读）。

26.在一些情况下，为了很少使用的模板函数就要包含一个大的头文件，这时候应该把函数点成非成员函数，放到一个单独的头文件中，再在这个头文件中包含那个大的头文件。如果有必要的话，比如说可能是函数重载这些，可以放拿到命名空间里面去。

27.据说编译器只认识文件开头（第一个不是宏的命令之前的部分吧）与结尾位置的文件防重的宏。

文件防重我们又称为文件添加保护，这个宏就叫保护符，又有内外部之分，内部保护符就是我那样直接写在对应的头文件中的保护符。外部保护是对#include<...>进行保护，这个比内部保护不好。

28.对于复制开销比较低的，在函数参数中就直接使用。不要认为复制开销就是大。

29.对于类重载运算符：可以进行a+b=c这样的代码，只给了一个临时变量，如要禁止这种的话就直接把返回值定义成const，这样出现这种代码就会报错。不过这样子也就去除了一些有力的代码，比如说对于std::string，a=(b+c).replace(pos, n, d);？？？。还有就是定义重载函数返回值要是左值。

30.对于函数重载，在一些情况下比如说像复数乘法这样会显著改变左边参数，这时候就应该去重载oprator\*，再用\*去写\*=。一般的话是\*=写\*。

31.从汇编以上的语言去实现++，--的前缀与后缀代码是很不同的。后缀要靠前缀实现，还要一个新的对象才能把原值传递回去。所以后缀比前缀从只+1的方面去看多做了一个靠临时对象进行原值的传递。不过嘲弄汇编而言，如果可以直接控制寄存器以及内存的存储，那么就可以将后缀与前缀变得一样。

32.一个基类应该有公有的虚拟析构函数或者受保护的非虚拟析构函数。

33.一个异常类最好从std::exception继承，它应该有一个公有的析构函数，一个不会失败的构造函数。有虚拟函数经常实现克隆与访问。

34.不要去设计那种巨类，应该是小类。（功能全要是必要的就对了）

35.在软件设计的时候人们比较容易被复杂的关系弄乱，所以就要尽力减少耦合。继承和友元是造成耦合关系的两大元凶，继承就可以用类的组合（把一个类型作为另一个类型的成员变量）来代替。

36.独立使用的类作为基类是一种严重的设计错误，应该避免。要添加行为就应该添加非成员函数而不是成员函数。要添加状态，应该使用组合而不是继承。

37.没想到优化都是一件要慎重的事情，书上有一句话：

优化的第一原则：不要优化。优化的第二原则（仅适用于专家）：还是不要优化。再三测试，而后优化。

38.程序员应该避免去心算

39.让变量的生存周期与变量的使用周期尽可能接近。

如果变量的初始化还没能定下就不要创建变量，或者变量的定义语句的分号先不要写。

40.真的最好最好不要让函数过长。代码块嵌套过深。

41.可以通过makefile来制定编译顺序。但还是要避免跨编译单元的初始化依赖。对于依赖都是要尽可能的减少的。绝对不能有过于复杂的依赖。

42.每一个头文件都能提供自己内容的所有依赖。

43.非成员重载运算符函数有着更高的效率。比如说\*与\*=，后者要有单参数的类构造函数，残能做到将左边的参数变成类，然后执行。而\*由于非成员又有对称，所以除了类构造，也可以将类类型转换函数转为其他类型的变量。

44.书上居然说：考虑重载以避免函数传参隐含类型的转化。

45.书上认为应该用组合代替继承，但是我没感觉到组合比继承好在哪里。

46.string这种就不该作为基类,以之为基类派生出super\_string,那么对string有用的函数你可以用么.继承也是一种比较强的关系,在正常情况下是不要使用的.你要扩充功能就用非成员函数。

47.抽象接口：没有成员数据的抽象类。应该遵守DIP（依赖性倒置原理）:高层模块不应该依赖于低层模块，两个都应该依赖于抽象；抽象不依赖于细节，细节应该依赖于抽象。

接口必须要保证，接口弄错了以后可能不被允许修改了。

48.显示地把子类中的虚函数声明出virtual，这样更明显。

49.子类的覆盖东东要保持可替换性。

51.要避免使用隐式转化，隐式转换经常是弊大于利，在使用前要三思而行。隐式转换有两个主要问题：它们会在最意料不到的地方抛出异常；他们并不总是与语言中的其它元素有效地配合。

隐式转化会让一些不正确的表达式式变得正确：

class String{

//....

public:

operator const char \*();

};

int x =s1-s2;

52.

指针的值其实也是数值，但是不能把除了0以外的数值赋给指针。

void Display( int )

void Display( std::string )

Display( NULL )//调用第一个，因为在函数重载就算都合适也要看程度（之前分过级）。不过因为NULL原则int会给出warning。

53.如果一个类有get/set函数那么也就意味着可以访问与更改数据成员，那么这个类就可以考虑是一个简单的struct，让数据成员直接public。

如果提供了对private数据的访问，就要控制好，比如说string的c\_str是返回const char\*。

54.Pimpl指针：

Pimpl指针缩短编译时间（使得此文件可以不依赖于数据成员中一切类型，只要在链接的时候将Impl引过来就可以了）。而且用指针的话在自己实现对象交换函数的时候是比较好的。可以让私有函数避开函数重名引发的重载。还可以在复制的时候做到较强保证。增强构造函数与析构函数的可读性。

class Map {

//...

private:

struct Impl;//这个声明应该放在类的里面

shared\_ptr<Impl> pimpl;

};

class Widget{//...

public:

Widget& operator=(const Widget&);

private:

struct Impl;

shared\_ptr<lpml> pimpl;

};

Widget& Widget::operator=(const Widget&) {

shared\_ptr<Impl> temp( new Impl(...) );

//如果构造失败就会抛出异常

pimpl =temp;

return \*this;

}

class Widget {                                // in header "widget.h"  
public:  
    Widget();  
     …  
private:  
    std::string name;  
    std::vector<double> data;  
    Gadget g1, g2, g3;                    // Gadget is some user-  
};                                                   // defined type

因为Widget的数据成员包括了std::string，std::vector，Gadget类型，这些类型的头文件必须和Widget去编译，那说明Widget客户端必须#include <string>, <vector>, 以及gadget.h。这些头文件增加了Widget客户端的编译时间。这里讲不需要编译的部分拉了出来。

55.优先编写非成员非友元函数 。当函数需要与其左参数不同的类型（例如操作符>> 或者 <<）；需要对其最左参数进行强制类型转换；能用类的工哦你用接口单独实现的就设置为非成员友元函数。

56.new运算符有三种形式：plain、in-place、nothrow。

in-place就地形式：（这一种new不需要有对应的delete）

void\* operator new(std::size\_t, void\* p) {return p;}

plain普通形式的函数声明：

void\* operator new(std::size\_t);

nothrow不抛出形式的函数声明：

void\* operator new(std::size\_t, std::nothrow\_t) throw();

我们一旦对其中一个new自定义，那么就会导致其余的两个也被隐藏，因为他会找的就只会是我们定义的那个域。书上说要避免隐藏in-place形式，在STL容器中有运用。

如果有人试图编写一个和普通旧式new C一样的表达式，编译器会拒绝对其进行编译。其根据是无法找到旧式的new运算符。

但是首先我不知道怎么把类型作为参数传递，其次我试了下上面三种都不能使用。

57.using B::operator new;

58.在构造函数的初始化类表，我们被建议顺序与类的成员数据定义顺序一样，C++这么做就是为了确保销毁成员的顺序是唯一的。这个销毁顺序就取决于构造对象的构造函数。当我们不遵守的时候，有的编译器会引发警告。（不过我还是不知道顺序不对有什么问题）

60.避免在构造函数和析构函数中调用虚拟函数：虚拟函数只是几乎总是表现得虚拟，在构造函数与析构函数中并不虚拟（子类不存在了），更糟的是，从构造函数或析构函数直接或间接调用未实现的纯虚拟函数，会导致未定义行为。

有一些设计方案需要后构造，就是在构造了完整的对象后立刻调用虚拟函数：

推卸责任：把这个问题交给用户，在文档中说明，让用户保证在构造了对象以后调用后初始化函数。

迟缓后初始化：在第一次成员函数调用的时候进行后初始化。所以基类就要有个标志标识表示是否已经后构造化过了。

使用虚拟基类语义：语言规则要求构造函数最底层的派生类决定调用哪一基类构造函数。

使用工厂函数:

class B{

protected:

B() {...}

virtual void PostInitiallize(){...}//构造之后打算立刻调用的函数

public:

template<class T>

static shared\_ptr<T> Create() {

shared\_ptrz<T> p(new T);

p->PostInitialize();

return p;

}

};;

class D: class B {...}

shared\_ptr<D> p=D::Create<D>();//创建一个D类

61.将基类的虚构函数设置为公有且虚拟或者保护且非虚拟。不过如果不想被多态使用那就不必虚拟。例外情况：COM、CORBA是一些组件架构。刚好这两个组件架构没有使用标准额度删除机制，而是重新编写了各种不同的对象清楚协议。

62.析构函数、释放与交换操作绝对不能失败。前面两者是为了在发生错误的时候进行有保证的回滚。delete和delete[]也是释放操作。交换操作不允许失败是为了成功后保证能够提交结果，以下是一段交换操作:

T& T::operator=(const T& other) {

T temp（other）;

Swap(temp);//用交换操作去代替一些大型的赋值操作

}

63.puts(NULL);是没效果的哦。

66.//104键盘上小键盘与上面的数字部分键码是不一样的，因为用google拼音会出现不同的效果。

67.赋值构造函数不能设置为虚拟。不能把大的类赋值给小的类。也不要返回const T&，这虽然能组织一些错误代码但是你也将不能把对象放到容器里面

T& operator=(const T&) {

T temp(other);

swap(other);/\*swap可以做到很好的防错以及高效。但是如果这个类型没有实现swap，那先看std::swap能不能在你这类型上正常工作，如果不能那就要用智能指针指向，然后对它们使用std::swap\*/

return \*this;

}

定义自己的swap函数的时候应该在与用户定义类型的相同的名字空间中提供一个非成员交换函数。此外，还可以考虑为自己的非模板类型特化std::swap。

namespace std {//没想到这样就加到std里面去了

template<> swap( MyType& lhs, MyType& rhs) {

lhs.swap( rhs );

}

}

68.绝对不能这样使用：

T& T::operator=（constructionT& rhs） {

if(this != &rhs) {

this->~T();

new (this)T(rhs);

}

return \*this;

}。

69.将类型及其非成员函数接口置于同一名字空间中。对于x.f()而言，他会在x的作用域里面查找f的定义。

70.

#include<vector>

namespace N {

struct X {}

template<typename T>

int\* operator+(T, unsigned) {/..../}

}

int main() {

std::vector<N::X> v(5);

v[0];

}

vector<T>::operator[] 内部隐藏着v.begin()+n这样的代码，而这个operator+函数的名字查找可能会进入，是否会进入取决于vector<T>::iterator 在这个函数库中怎么实现。其实标准库不应该编写这样的可能存在问题的代码，这就是插入了一个用户的自定义点，要么安排v.begin()的类型不要以任何方式依赖于模板参数，要么就将operator+这种调用限定。不过我觉得如果使用这样定义的标准库，那么我觉得对于迭代器的加法没住std::operator+更合适，不过进了N这个命名空间就只能是这个自己写的operator+了。避免这种问题：因为ADL查找，我在附近使用类型，会导致在命名查找的时候导入了对应的名字空间，然后就比较容易出现问题。除非命名空间里面是只存放与类型相关的类型。

71.不要在头文件或者#include之前使用using，怕影响到了头文件中定义的命名查找。

72.不要在头文件中定义具有链接的实体：具有链接的实体包括名字空间级的变量和函数，都需要分配空间，在头文件中定义这样的实体会导致连接时错误或者内存的浪费。请将所有具有链接的实体放在实现文件中。

头文件中放了函数定义，有不少文件包含了这个头文件，也就会相当于每个源文件中定义了函数体，并分配了空间，认为你每一个文件都想私有一个函数，链接的时候没准会出错。你可以在头文件中放函数声明，前面extern可有可无，没关系的。

不要试图在头文件中使用未命名空间来绕开此问题，因为结果是无法令人满意的：

namespace {

int fudgeFactor;

string hello("Hello, world!");

void foo() {/.../}

}/\*不像其他名字空间，未命名的名字空间的定义局部于一个特定的文件，不能跨越多个文本文件。许多 C++实现都支持全局静态声明，但是，随着越来越多的 C++实现都支持名字空间全局静态声明的用法将会被未命名的名字空间成员所取代。 \*/

不过，以下具有外部链接的实体可以放入头文件：内敛函数，函数模板，类模板的静态数据成员，这三个是可以被链接器认出多个副本的。一般静态数据也不要在头文件中定义，不过有一种全局数据初始化技术叫Schwarz计数器或者灵巧计数器，他需要在头文件中放入静态数据。

73.并不是所有的静态都是程序启动前初始化，局部的就不是。它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束。至于分配内存的时刻么，似乎没什么明确的约束，但总之能确定是在初始化前。一般实现会生成预留存储的代码，在程序开始的时候就分配内存，只是没有被这些变量占用。

74.对一个程序的不同模块如果不能确定使用同一种编译器和编译选项，那就不要在模块之间传递什么，比如说异常。

75.以下地方应该要有捕获所有异常的catch(...)：

在main函数附近：捕获并用日志记录任何将使程序不正常终止而其他地方有没有捕获的异常

。

对于不能失败的函数比如说析构函数这些就要将异常处理掉，把自我保护好。

操作系统和程序库会提供一些框架，可以传递一个指向以后才会调用（例如，发生异步事件）的函数指针，不要让异常传播到回调函数外面去，因为调用回调函数的代码可能使用不同的异常处理机制，还可能不是C++写的。

在线程边界附近：线程是在操作系统内部创建的，这是为了能不让异常往系统传递，否则将出乎系统预料。

一个库就是一个模块，异常是要在内部自己解决掉的。

76.抽象层次越低比如说char\*于string，那么可移植性越好。同一个string在不同的情况下（不同的库、编译器选项...）可能会有不同的实现版本，这样子，练级的时候就会出现问题。

77.静态多态和动态多态：

#include <iostream>

using namespace std;

//模板实现虚函数多态

template<class derive>

class base

{

public:

void print()

{

derive::print();

}

void m\_print()

{

downcast()->derive::m\_print();

}

protected:

inline derive \* downcast()

{

return static\_cast<derive \*>(this);

};

inline const derive \* downcast()const

{

return static\_cast<const derive \*>(this);

};

};

class der:public base<der>

{

public:

der(int foo):\_foo(foo){}

static void print()

{

cout<<"in der print"<<endl;

};

void m\_print()

{

cout<<"in der member fun m\_print"<<endl;

cout<<"has member foo="<<\_foo<<endl;

}

private:

int \_foo;

};

//继承

template<class base>

class der2:public base

{

public:

static void print()

{

cout<<"in der2 print"<<endl;

};

void m\_print()

{

cout<<"in der2 member fun m\_print"<<endl;

}

};

class tmpclass

{

public:

void test()

{ cout<<"in test"<<endl;}

};

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

{

//模板实现虚函数多态

base<der> \* pb= new der(100);

pb->print();

pb->m\_print();

//继承

der2<tmpclass> d2;

d2.print();

d2.m\_print();

d2.test();

return 0;找。

}

78.有意地进行显示自定义并记入文档：

t.foo()

foo(t)

S<T>::foo(t)

避免无意的自定义点（破坏掉ADL查找就可以了）：

将模板内部使用的任何辅助函数都放入自己的内嵌空间，，在调用的时候都显示调用从而禁用ADL。

因为某种原因而提到模板参数的一个名字，许多编译器都不支持C++标准所要求的依赖名的两阶段查找，而且这意味着使用依赖名的模板代码在不同编译器中将有不同的行为，除非在使用依赖名的时候小心地说明。存在依赖名时需要特别注意，这往往出现在类模板从其模板参数之一继承（template<typename T>class C: T {}）或者从用气模板参数之一构造的一个类型继承（template<typename T>class C: x<T> {}）。简而言之，在调用依赖基类的成员时，应该总是用基类名或者this->显示地进行限定，从而让编译器知道是去基类里面

79.不要特化函数模板，不能对函数模板部分特化，函数模板特化决不能参与重载。解决办法是使用类模板来实现函数模板，这样就可以避开局限性。如果想重载一个特化的函数模板，那改成非模板函数。

80.使用empty()代替size() == 0因为empty可读性更强而且更加通用，list还不知道是什么是没有size()的。

81.在调式模式而且没有定义NDEBUG的时候，assert函数才会起作用。

建议使用方式：

assert(a!=0 && "information")

assert(!"information")

79.对于错误处理：有三种力度的保证：

基本保证：确保程序出错的时候处于有效状态

强保证：进一步保证回到最初状态或者到达目标状态

不会失败的保证：再进一步就是必定不会失败

80.除了异常处理以外，对错误的处理还有借助错误码。异常比错误码好在不能不加修改的忽略，异常是会自动传播的而且布局也不错，而却错误码不能用于构造函数与易购函数。但是有些时候项目可能会遇到压力而考虑关闭异常处理，因为编译器异常处理机制的时间保证最差，一些关键操作很难或者不可能满足必须的时间要求。

通过值抛出通过引用捕获。书上还有句话:抛出指向栈分配的值的指针是不可行的，因为在指针到达调用处之前栈还没有展开，抛出一个动态内存分配的指针都是没有问题。

接受并发出同样的错误是为了添加测试错误代码。

书上说有的是将两种错误处理机制混合在一起，捕获一个异常并且发送一个错误码。

81.异常规范：提供了一种方案，可以随着函数声明列出该函数可能抛出的异常，并保证该函数不会抛出任何其他类型的异常。T Pop() throw(popOnEmpty);表明我们认为这个函数它直会抛出这个异常。 如果在运行时，函数抛出了一个没有被列在它的异常规范中的异常时（并且函数中所抛出的异常，没有在该函数内部处理）则系统调用C++标准库中定义的函数unexpected()。

虚函数中的异常规范：派生类的虚拟函数的异常规范必须与基类虚函数的异常一样或更严格。

class CBase

{

public:

virtual int fun1(int) throw();

virtual int fun2(int) throw(int);

virtual string fun3() throw(int,string);

};

class CDerived:public CBase

{

public:

int fun1(int) throw(int); //错！异常规范不如throw()严格

int fun2(int) throw(int); //对！有相同的异常规范

string fun3() throw(string); //对！异常规范比 throw(int,string)更严格

}

不过《C++编程规范》上说要避免使用异常规范，专家程序员也不应该这么做。事实上，异常规范会使编译器在函数体周围以隐式的try/catch块的形式增加运行开销，从而运行时检查是否只抛出了定下的异常（可能为空也就是告诉编译器不会抛出异常），除非编译器能够静态证明绝对不能违反异常规范，只有这种情况下他才可以自由地优化掉动态检查。有时候可能还会组织进一步优化，例如有些编译器就拒绝内联具有异常规范的函数。异常规范并不是灵活的工具，一旦违反就会终止程序。所以无用而有害。

82.关联容器有set、map。

基于散列的容器名字以hash\_、unorder\_开头

默认set采用红黑树实现，支持lower\_bound函数，返回大于给定参数的set中的最小值，已经upper\_bound。STL中提供的lower\_bound函数一般用于二分法，需要先将数据排序完成。Java中对应为TreeMap的lowerKey、higherKey。

如果重视查找速度，那么优先基于散列的容器，然后vector，再set或map。

只有当元素数量达到几千时，容器之间的性能差异会显现。

vector用于小型类表几乎总是优于liist的，即使在序列中插入对vector而言是个线性时间对list却是个常数时间。当数据量大了，或者需要对插入有强保证那就要用list了。

83.现代标准库中包含了许多优化。

84.使用vector或者string.c\_str来与非C++ 函数交流数据。vector由于存储空间是连续的，所以可以直接弄一个地址回去个函数就好了，&\*v.begin()、&v[0]、&v.front(),要获取第n个元素地址例如&v.begin()[n]、&v[n]（书上说要先运算在获取指针，不能先获取头指针在运算，因为前一种情况可以有机会验证没有越界发生）。

有些STL会把vector::iterator定义为指针，只是有些哦。

85.在容器中只存储值和智能指针，对于指针容器，如果容器拥有所指对象的话，就要使用引用计数的智能指针容器（比如list<shared\_ptr<Widget> >），否则使用原始指针的容器（比如list<Widget \*>）或者使用其他类似指针的值（比如list<vector<Widget>::iterator >）。

关于容器拥有对象这句话的理解：

如果指针容器内的指针所指对象仅仅被他指向，那么就是拥有。不拥有的话可以是这样，一个地址赋值给容器中的一个元素。

auto\_ptr<T>的对象并不类似与值，因为它们具有所有权转移的复制语义，使用包含auto\_ptr的容器应该无法编译，万一能编译也不要这样写。从c++11开始, auto\_ptr已经被标记为弃用, 常见的替代品为shared\_ptr。shared\_ptr的不同之处在于引用计数, 在复制(或赋值)时不会像auto\_ptr那样直接转移所有权，auto\_ptr对资源的拥有者只能有一个, 当两个auto\_ptr进行等于号(赋值)操作时, 等于号后面的auto\_ptr将失去资源的所有权。

异构容器：就是指容器存储和拥有不同但是相关的实际对象的虚拟函数。就是使用了类的继承，估计要用的是安全指针。

如果需要map<Thing, Widget>，但是有些Thing没有相关联的Widget，就应该使用map<Thing, shared\_ptr<Widget> >。

为了获得一些特殊的元素顺序，创建分容器指向主容器。

86.在序列中追加元素：

vecto<int> vec\_i;

vec\_i.resize(vec.size()+1, 1); {1}

vec\_i.insert(vec.end(), 2); {1, 2}

vec\_i.push\_back(3); {1, 2, 3}

使用后置迭代器

不太清楚为什么他说第一种和第二种方法的性能是二次的。

无法访问容器，标准算法就不能直接调用pusk\_bakc。但是可以通过bakc\_inserter，要求编译器使用push\_back。

据说push\_back对内存的申请是指数级增长的。

87.删除容器元素的方法：

container<T> (c).swap(c);//可以去除类似container类型容器的c的多余容量

container<T> ().swap(c);//去除全部内容和容量

remove算法并不真正地从容器中删除元素，而是将要删除的元素移至范围的开头，并返回一个迭代器指向最后一个不应该删除的元素的下一个位置。要真正的删除还要再remove之后嗲用erase。比如c.erase(remove(c.begin(), c.end()), c.end() );如果有成员函数版本就尽量使用成员函数版本。

shrink-ti-fit对写时复制方式实现的std::string不适用。总是可行的是使用s.reserve(0);或者用上面的swap惯用法。

88.大多数带检查STL会通过在容器和迭代器中添加附加的调试和支持信息，来自动地检查这些错误。这样会导致迭代器臃肿，仅限测试期间。

89.用算法调用代替手工编写的循环。不过简单的循环但也是不错，简单而明了。

90.Lambda函数将函数代码移至调用点，函数别太长就增加了可读性。

也可以选择使用标准绑定器：bind2nd不过很笨拙

deque<double>::iterator current =d.begin();

for(size\_t i =0;i < max; ++i) {

current =d.insert(current.data[i]+41 );

++current;

}

以上代码是有问题的，迭代器会在内存重新分配空间的时候时效，那么就不该使用++。

而算法调用可以绕过这个陷阱：

transform(data.begin(), data.end(), //从data复制元素

inserter(d, d.begin()), //到d的头部

bind2nd(plus<double(), 41>) ); //每个都加41

书上的Lambda函数是这样的：

transfrom(data, data+max,insert(d, d.begin(),\_1+14))

91.对于有序范围来说，应该用一下4个二分查找算法: binary\_search，lower\_bound，upper\_bound， equal\_range他们都是对数时间。binary\_search是个好名字，但可惜的是他只返回一个bool来表示找到了没有。一般使用lower\_bound，upper\_bound。equal\_range的开销相比就多了点。

lower\_bound是返回指向第一次匹配的迭代器，upper\_bound是返回指向最后一次匹配的迭代器。

对于有序范围内的计数：p=equal\_range(first, last, value); distance(p.first, p.second);是count(first, last, value);的更快版本。

92.排序一般总是不用这么完全地进行，按开销从低到高的大致顺序，有如下标准算法：partition、stable\_partition、nth\_element、partial\_sort（变体是partial\_sort\_copy）、sort和stable\_sort。

partition、stable\_partition、nth\_element的时间是线性时间。

nth\_element、partial\_sort、sort、stable\_sort需要随机访问迭代器，如果只有双向迭代器(例如list<T>::iterator )，就无法使用它们。那就使用索引容器：创建一个支持随机访问的迭代器容器，其中的迭代器指向原范围，然后我们对这个迭代器容器使用算法。

nth\_element和partial\_sort无法保留相等元素排序前的相对顺序不变，毕竟名字没有stable\_。partial就是因为工作少而比sort快，但是如果是要对范围中的大多数或者全部都进行排序那么sort是更快的。

如果使用set/multiset或map/multimap或则priority\_queue适配器，而且只需要一种排列数序，那么其中额度元素将总是有序的。

partition就是根据断言（又称谓词）将范围划分成两组并返回迭代器。

nth\_element：

以你的代码为例：你的第2个参数是intVect.begin()+3，就是想看看intVect这个序列中，【如果排序的话】，它的第4个应该是多少？这个intVect如果全排序，应该是：

0 1 3 5 6 7 9，

因此nth\_element在确保第4个元素=5的时候，就停止排序了。 因此你后面输出intVect的值是：

1 0 3 5 9 6 7

就是第4个=5后，当时intVect的值。

一般在nth\_element()后，都有一个类似这样的语句：

nth\_element(intVect.begin(),intVect.begin()+3,intVect.end());

cout << intVect[3]<< endl;

给你添加一点代码，体会一下nth\_element的概念，它其实可以有【第4个参数】的：

nth\_element(intVect.begin(), intVect.begin() + intVect.size()/2, intVect.end());

cout << "The median is " << intVect[intVect.size()/2] << endl;

nth\_element(intVect.begin(), intVect.begin() + 1, intVect.end(), greater<int>());

cout << "The second largest is " << intVect[1] << endl;

之后在看看intVect的输出：

The median is 5

The second largest is 7

9 7 6 5 3 1 0

sort 的排序选择：

1.当长度n小于32的时候，直接调用插入排序

2.当长度n大于32时，先用快排处理1.5\*log2(n)次，如果长度还是大于32——说明这个数组不太适合用快排，所以调用堆排序来处理

3.处理1.5\*log2(n)次后如果长度小于32，调用插入排序

93.谓词就是断言，也就是返回bool的函数对象。

书上有一段话不是很懂，不过意思就是小心使用谓词。

class FlagNth {

public:

FlagNth( size\_t n):current\_(0), n(n) {}

template<typename T>

bool operator()(const T&) {return ++current\_ == n\_; }

private:

size\_t current\_, n\_;

}

v.erase(remove\_if(v.begin(), v.end(), FLahNth(3) ));的TL实现中，删除的不仅仅是第三个元素，还有第六个元素。以为一般remove\_if是通过find\_if和恶remove\_copy\_if实现的，它会将谓词的一个副本传给这些函数，而在此之前可能它自己已经执行了某些会影响谓词状态的操作。

94.算法和比较器的参数应多用函数对象少用函数。函数对象的适配性好，而且一般比函数快。有些时候对于已有函数，还必须要用ptr\_fun或者mem\_fun（给成员函数用）将其封装起来以提高适配性，例如为了用绑定器来构造更加复杂的表达式：

inline bool IsHeavy(const Thing&) {...}

find\_if(v.begin()，v.end(), not1(IsHeavy) );//这会因为不可失陪而产生错误

find\_if(v.begin(), v.end(), not1(ptr\_fun(IsHeavy) ));//一次失败的尝试

ptr\_fun能够准确推演出参数和返回值类型（泥马，明明可以推出返回值类型那为毛重载的时候不算上），此处参数类型被推演为const Thing&，之后就继续创建内部机制，在此过程中会尝试添加另一个&，但是C++中没有对引用的引用，所以就出错了。

如果使用正确编写的函数对象，那么完全用不着这样：

struct IsHeavy: unary\_function<Thing bool> {

bool opeartor()(const Thing&) const {...}

};

fidn\_if(v.begin(), v.end(), not1(IsHeavy() ));//可以适配了

直接用函数去实例化模板是非法的：

bool ComareThings(const Thing&, const Thing&);

set<Thing CompareThings> s;//错误

struct CompareThings: public binary\_function<Thing, Thing, bool> {

bool oeprator()(const Thing&, const Thing&) const;

}

set<Thing CompareThings> s;//正确

95. C++的成员变量就是根据声明的顺序存储的。

96.避免使用reinterpret\_cast。reinterpret\_cast的后果取决于具体编译器实现如何定义，因此甚至无法知道编译器是否会那样做，它既不可靠又不可移植。除了在有些与特定系统相关的底层编程，要使用reinterpret\_cast在端口串行输入和输出数据，我们都要最大限度的减少使用这个不安全的强转。

T1 \*p1 =...;

T2 \*p2 =reinterpret\_cast<T2 \*>(p1);可以转为

T1 \*p1 =...;

coid \*pv =p1;

T2 \*p2 =static\_cast<T2 \*>(pv);

reinterpret\_cast就和使用union中心解释一样。

97.不要对动态对象的指针使用static\_cast，安全的代替方法有使用dynamic\_cast，重构，乃至重新设计。用这个类型去创建，却按另一个类型去销毁当然会问题。书上建议使用dynamic\_cast代替static\_cast，它说虽然效率差了点，但是能价差非法强制转换。你可以之后再换回static\_cast来发布。

书上说C的强制转是某些情况下的static\_cast或者某些情况下的reinerpret\_c

ast。

98.书上说编译器可能会把常量数据存在ROM中或则写保护的RAM中，对于这种物理上的const对象，如果强制转换掉其常量性那么可能会导致内存故障。就要求避免使用const\_cast。

99.不要使用可变参数(...)：

告诉编译器，类型不要检查，由我接管，启动reinterpret\_cast

在C++中通过它传递C内置类型以外的类型，是未定义行为

100.在函数传递的时候C风格字符串可以赋值给string，涉及到的其实并不是隐式类型转换

1.using namespace std;这个的位置要放在最前面，因为函数中变量声明string需要它。你可以using namespace std;后再using namespace libxl;是一个空间里没找到就在另一个空间中找。C++标准库所有的标识符都被定义在std这个命名空间中。这种贪方便的做法容易引起污染。

using std::cin;仅对cin起作用。 cin->std::cin 不支持using cin=std::cin;说后面的cin不能解析

可以在使用了其中一个后再用上另一个，两个规则又不冲突。因为在两个命名空间里不存在同名变量或者函数，存在的话就要自己加上域运算符去避免。

using std::string; <=> using string = std::string;

using INT =int;用using来声明变量别名是要C++11支持的。

using int4 =int[4];注意数组的写法，也只有这样写了

using namespace std;就是增加了域名的查找范围。

using std::cin,std::cout;是不合法的，using只能后面跟一个

using std::string; => using size\_type=std::string::size\_type <= typedef std::string::size\_type size\_type

2. 把一个int以%s输出，不会被编译器找出，但这超出了调节范围，于是error?我觉得是将int当做字符串的地址解析，但是这部分内存不访问于是段错误

4.string：string是basic\_string这个类模板的char的实例化。

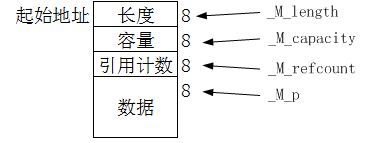
空串是什么都没有，VS、g++默认暗中实现了以\0结尾。因为''表示空字符，但是不支持string s1(n, '');由sizeof("")可以知道""<=>"\0"，所以用字符数组给string赋值的时候会涉及\0的去除与数字统计。这是两种思想。

(默认初始化为空串)string s1; <=> string s1(""); 直接赋值,隐含着从C到string的转化,是空串</=> string s1 ="";复制赋值,是空串，也有转化，当然也就有string str{""}（有与没有等于号两种形式）对空string也就是空串输出什么都没有。string sNull2(3,'\0');这种方式先申请8个字节的空间，再push\_back三个字符。任意char都可以压入。string可以被{1,2}列表初始化，并不是个错误，而是将1,2转换为char，然后合并成为字符串。str{"a"}>str2{2, 'z'} 区分str2{2, 'z'}与str(2, 'z')。

char \*a ="abc";在C中是经过一定操作的（"abc"是字符串，因为它能被隐式或者显式的转化为地址，所以估计就是可以被理解为字符数组。char数组自动被转化为char指针）。

'0x02' 明明是一个字符，但是内部输入了一个字符串，会将最后一个字符作为本常量数值

C++中的string不是不可变对象。string是个类，不是个地址，也不是个指针。在我的g++下sizeof一个string变量是4字节，不管内容是什么！sizeof(string)=4可能是最典型的实现之一，不过也有sizeof()为12、32字节的库实现。 但是MS2015测试后sizeof(string)=40.还是跟编译器有关。gcc中是一个指向下图这样结构体的指针：



采用写时复制，一般通过引用计数记录来确定回收内存的时机。

字符串操作符：<< >> getline(stream, str,‘\n’)原型是istream& getline ( istream &is , string &str [, char delim ]);其中 istream &is 表示一个输入流，譬如cin；string&str表示把从输入流读入的字符串存放在这个字符串中（可以自己随便命名，str什么的都可以）；char delim表示遇到这个字符停止读入，该字符不会被读入，在不设置的情况下系统默认该字符为'\n'，也就是回车换行符（遇到回车停止读入），函数返回流的状态。但是在>>之后直接使用被视为无效语句直接过去，cin.getline(str,num,’\n’)同样，需要调用cin.ignore()忽略一个换行符 s.empty() s.size()或者s.length().返回一个string::size\_type类型(在string这个类中定义即string::size\_type（C中关于数据的索引是size\_t，是sizeof的返回类型），这个类型只能确定是unsigned与足够大去容纳元素 s[n](由于size\_type是unsigned类型，所以n会被当unsigned处理) substr() c\_str()

std::fpos<std::string> 、string::iterator、string::size\_type可以用于string的遍历

加上一段字符：

s.insert(s.size(), " 4th Ed."); //第一个参数是想插入的位置

s.append(" 4th Ed.");

s += " 4th Ed."

字符串的拼接效率不同（从高到低）：append()、+=、stringstream、sprintf()，两种自带方法本质上都是函数调用，<https://www.zhihu.com/question/56727144> 两者的性能其实应该差不多，但是因为写法的特殊导致，string增长过程中最重要的一个耗时函数被多次触发，所以一般推荐append。 Java中对字符串需要频繁更改推荐使用 StringBuilder 进行函数性能比较的时候关闭编译器的自动优化，与一些不需要关闭优化的要确保优化在同一级别，甚至优化会影响测试效果。

string的搜索成员函数（找到返索引，没找到返npos，都是std::string::size\_type类型）：

while ((pos = name.find\_first\_of(numbers, pos))

!= string::npos) { //npos的值是-1，用来代表不存在的位置

cout << "found number at index: " << pos

<< " element is " << name[pos] << endl;

++pos; //这个不错

}

string与整数、浮点数之间的转化：

数字到string：

to\_string()这个是C++11函数我的g++还不支持转换函数

自己实现的方法：push\_back()将字符一个个装入，低效

#include <iostream>

using namespace std;

int main() {

string str = "";

str += 123;

cout << str << endl; // 输出{

cout << ""+123 << endl; //副作用无效

}

string到整数，参数都是(s, p, b)：stoi(), stol(), stoul(), stoull() p是指向size\_t的指针，话会size\_t不是数组的吗，默认 b是numeric base也就是进制，默认10

string到浮点数，参数都是(s, p)：stof(), stod(), stold() p和string转整数相同。浮点数一般不会进制选择所以就没有b。

string s2 = "pi = 3.14";

d = stod(s2.substr(s2.find\_first\_of("+-.0123456789")));//就像正则寻找一样，但并不是很好

如果string不能被转化为数字，那么函数就会抛出invalid\_argument的异常。如果转化出来的数字不能被表示，函数就抛出out\_of\_range。

字符串分隔：者通过strtok()又或是线程安全的strtok\_r()，或者需要自己进行实现（使用string的find和substr来快速实现）。一般应该叫做split

string str(NULL);这种写法虽然可以编译通过，但是会导致运行错误。

通过上面总结：一般类需要知道的就是构造、析构、增删改查、与部分基本类型之间的转换与配合

6.C++里面的类文字常量：string(10, 'c')。不对他进行命名，想到了异常处理，不过这个实质上是常量，而异常处理从本质上看还是一条赋值语句。

7.让cin不可用①输入不合适的字符（类型不对的字符）②以文件结尾去关闭输入流 那么要让getline()返回一个不可用的流的状态只能是文件尾了。它错误以后就一直是那个错误状态除非你刷新。

cin是从缓冲区中提取数据， >> 这两个操作绝对不会读取空白字符即使是读给char，但原理是：字符先经过他们(缓冲区与一系列的识别处理std::cin是对象呀。)才给char。这样去理解C里面的scanf的转换说明符就明白了。联系正则的话，那就是种匹配。通过 cin.getchar() 读取空白字符

cin>>获取输入之后，getline会直接失败，并继续执行程序其他语句

#include <iostream>

using namespace std;

int main() {

int w,h;

cin >> w >> h;

char map[h][w];

for(int i2=0;i2<h;++i2) {

cout << "1" << endl;

cin.getline(map[i2],w);

}

return 0;

}

//输入 1 2或者2 2就会明白需要一个cin.ignor();

ungetc 将一个字符重新放回到流中，支持stdin

cin.unget会让输入流回退一个字符

cin.putback 能将一个变量插入到输入流的下一个读取位置

7.5. 格式化输出

C语言printf是可变域宽和精度，两者都可以用\*，这部分作为占位符使用。前导0的使用要辅以字段宽度，如%02d。

0.0/0.0 是-nan

cout << hex << 数字型变量 //输出十六进制 最前面没有0x

//字符类型或者char\*不会被转换为hex数字输出

cout << boolalpha << true; // boolalpha不会将整数转换为true、false输出

9.error：stray '\241' in program我按了shift+Space结果打了个全角的空格，这是一般空格的两倍大小，而中文空格却正是英文空格。

10.cctype多出的c就是表示这是ctype.h的引用。

11.超范围的读取会导致不必要的内存页加载进来，只要没有读到不可访问内存页（不可读写）以及终止符，就能够一直读取下去。

12.foreach遍历方法：

for (auto c : str) // for every char in str,新建一个对string中每一个字符的类型char。在不用auto而用std::fpos<std::string>for(auto &c: str) 如果想去改变的话:

str为const string，那么&c是要加上底层const的（如果不用auto就要用const char），就是c的话倒不用。auto可以自带底层const。

15.C++11前verctor<>的尖括号里面如果是模型的话，比如看vector<vector<...> >后面两个>之间要求必须有一个空。这样有个空确实好看了，不然编译器不能解析。

26.c++文件后缀：.cc, .cxx, .cpp, .cp, and .C。 头文件(如果有后缀的话)：.h，.hpp

一个标准输入object：cin，3个标准输出object：cout，cerr(标准错误流(非缓冲))，clog(标准错误流(缓冲))在运行窗口出现的时候他们都产生了

缓冲区的，都是缓冲区满或者遇到人为刷新时才输出.

26.1 我说的高水平top-leve低水平low-level在中文翻译中是顶层const与底层const。全局const在C语言中，是运行时变量，但是会被编译器在编译时进行优化，这时候效果相当于C++标准中的constexpr(没有全局要求)。constexpr用在类的时候，类需要至少有一个constexpr构造对象，如果需要传入参数，只能使用默认函数参数。

const int&和const int\*的const都是低水平const

const int\*不能赋值给int \*,而int \*const可以。前者的const修饰指针变量，后者修饰指针所指的值。

26.2 auto的原理是先计算出值，再去看值，最后再去修饰得到最终类型。由于&\*对于const变量是必须要有const修饰的，所以auto推荐加上const，这样做就是再次提醒，而且对于常量的auto引用是必须的。那对于auto &a=(const int)2，这个类型转换会被无视。还有(const int)a不会报错。decltype则应该是直接根据表达式的形式去判断及返回类型，所以decltype(别名)即引用，这个表达式是个指针变量的值或者是含有变量且最外层有括号的也认为是引用。引用参与运算的表达式形式上已不是引用,\*p也被认为成是引用类型形式（有问题啊居然不是指针引用啊，因为C++用引用），同auto，高水平const会被无视，低水平const会自动添加或者自己突出。auto sz = 0, pi = 3.14;是不正确的，会判断出两个是不同的类型，这或许是为了可读性，反正这种情况被当做一种错误了。虽然0是常量，但是sz是个int而非const int，因为这个0是int 0而非const int 0。const是对变量而言的。

auto &h = 42; //error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'这个错误说明了42是int常量而非const int，在引用时是要加上cosnt的。由此看见常量是右值么。报的错应该与int &h =42;相同

//auto 例0

const int a =1;

auto b =&a; //b有着低水平const,这个设置是必要的

const auto c =&a;//我们加的这个是顶层const!!!

const auto &j = 42;//我们加的这个是底层const，因为42是常量，当作特例去记。

//decltype 第一个例子

#include <iostream>

int main() {

decltype(1) a; //int a

decltype((1)) b; // int b

decltype ((a)) c; //int &c 且必须初始化

const int &i = 1;

decltype(i) d=2; //int d 且必须初始化

int i1=1;

int &i2=i1;

decltype(i2) e; //int & 且必须初始化

decltype(i2+0) e; //int

return 0;

}

//decltype 第二个例子

#include <iostream>

int function(int x=3) {

return x;

}

decltype(function) f;//这个当作了函数声明来用，这个声明里面包含了默认初始化。

int main(void) {

decltype(function()) a;//这可不是函数声明

std::cout << f() << std::endl;

return 0;

}

int f(int x) {

return x;

}

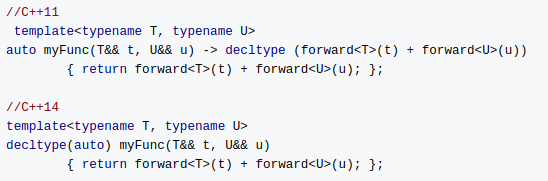
constexpr需要类型是常量类型:目前有算数类型，引用与指针。string和vector由于性质要求就可变故不是，类极可能也不是。引用与指针(除去对指针的常量赋值)为constexpr只需是对固定地址的即可（不可以是局部变量这会自己没掉的，全局倒是可以）。所谓的固定是指贯穿整个函数的小固定或是整个文件的。

指针可以constexpr int \*q = nullptr; 因为nullptr是一个固定的地址呀。

int j = 0;//j是全局变量

constexpr int \*p = &j;

我也觉的应该是可以的，不过指针与引用还要加上const（而不仅仅是constexpr，如果你被指向或者引用的对象是constexpr或者const的话），以符号规则，仅用constexpr会说不能把const int \*给int \*。但是明明cosntexpr声明了以后就是read-only呀，可是编译器蠢，认为就是要有吧。



26.5. sizeof：

返回size\_t类型（在cstddef（stddef.h）中定义），ptrdiff\_t(有正负作为保存两个指针相减的结果的类型)，size\_t string里面的下标运算符是std::string::size\_type

C++中sizeof不能直接对函数名进行使用。对函数指针肯定是可以的。

下面这段代码如果gcc test.cpp报C++错，只能gcc test.c：

#include <stdio.h>

void func(void) {

}

int main() {

printf("%d",sizeof(func)); //输出1

}

《=》

#include <iostream>

using namespace std;

void func(void) {

}

int main() {

cout << func; //输出1

}

sizeof是在编译期间静态解析，最好加上圆括号使用。sizeof求到的是一个const expression。

sizeof不会对表达式进行值的计算(这种计算都是在运行中)，所以sizeof \*p中的p可以是无效的。sizeof只需要知道的是类型。确实如此，已经试验。

sizeof对引用实际上是对object，引用应该就是在编译器的变量表里了！

int x[10]; cout << sizeof(\*x) << endl;这时候是个4，这是首元素的大小

30.在linux中<<缺少右边的操作数会导致程序卡在那里，而windows没事。

34.直接初始化之()是调用了类型的构造函数初始化。将创建的临时对象直接作为左侧对象。对于内置类型来说，编译器有默认的构造函数。如果带有三个参数似乎是代表将第一个参数的第二个参数位置到第三个参数的位置部分复制过来，不过string的第三个参数是复制的个数，位置以索引号表示。

列表初始化{} 前面可以有=可以没有，这种方法没有使用构造函数，所以凡是能导致精度降低、范围变窄等等的初始化情况，统称为 narrowing conversion，编译器都会警告（因为不是我们主动做的），不要这么做。常用于POD结构即全是数据成员的结构。vector<string> v7{10};这时的10只能代表size（string不支持{}）。而对于vector<int> 只能是内容上的初始化了。列表初始化可以用于任何初始化与一部分赋值的情况:

std::vector<std::string> vec\_s{10};//correct

std::vector<std::string> vec\_s={10};//这就被认为好似类表初始化而报错

复制初始化= 总是调用复制构造函数。复制初始化首先使用指定构造函数创建一个临时对象，然后用复制构造函数将那个临时对象复制到正在创建的对象

上面的()、{}是要C++11支持。

对于vector而言，std::vector<int> v1 =12;是不可以的，因为右值是单独的int，不存在这样的转换,这实际是要push\_back才能放进去，而这种不符合构造函数。

vector<int> ivec(10);会被默认初始化为0

vector::size\_type是错误的，vector<int>::size\_type才是完整的

35. c++的错误一个错就会引发一大片的错误报告。一般看开头与结尾

往往要两三个一起看下，不一定第一个就是准的。

同一个地方可能被报两个一模一样的错，比如说你用了一个全角的空格。

错误判断不可轻信，我多个const，应该报错说read-only，但是.......哗了狗了

36.已有int a[4];再int (\*p\_array) [不是4]=a;是会报不能转化的error。

39.main函数是由操作系统运行的。main函数的参数列表是void再接受参数也不会出错，因为这时候不是编译阶段了，编译器管不了。

41.查看返回值类型：windows ：echo %errorlevel%（在书上%内大写）

unix ：echo $?（结合$标志符记忆）

44.endl被认为是个特殊值是一种流操作者（manipulator），他们都是object。

46.当EOF发生在scanf与>>上是不同的，前者大概每次启用都会更新流的状态，而后者不会，所以就是会卡在那里直到你更新。scanf 没有流的状态描述变量

50. #include <iostream>

int main()

{

std::cout << std::cout;

return 0;

}

输出：0x477864而且就一直都是这个值咯，这个对象每次都在固定的位置。

std::cout是个object，而它自身没有定义输出符号<<。

51.>>操作很可能要套个if！！！

52.最少位数：

wchar\_t 16位 char16\_t 16位 char32\_t 32位

它们用于扩展字符集：wchar\_t是被保证足够大去支持机器中最大位数的扩展字符集，而后两者就是为Unicode准备的。在我的机器中分别是2,2,4字节，刚好是最小要求。

int 16位-》 long 32位-》 long long 64位

float 32位-》 double 64位-》 long double 96或128位

浮点数最少有效数字：

float 6位((⊙o⊙)…，书上说编译器一般会是7位) double与long double都是10位（py里面是17位诶）

55.一个十进制常量能适合地赋值给int，long，long long，因为他是个int型。如果给unsigned是要有u/U后缀的。而八进制与十六进制常量则可以适合地给int以上所有整型，因为都会有转化。

56.常量修饰符：前缀:u Unicode16位编码char16\_t U Unicode32位编码 L wchar\_t(L可能是Large的意思) u8 utf-8 char（用于字符串常量）

后缀：long与long double的都是l或者L（不过l基本不用，虽然编译器可能有颜色区分）

wchar\_t,char16\_t, char32\_t要开启C++11支持。 加上bool, char, signed char, unsigned char, short, and unsigned short，他们也会自动转化为int开始，unsigned...一级一级上。总的来说就是int下所扩张向上的选择。

有符号类型与无符号类型在一起只要无符号类型在位数上不低于有符号类型，就会把有符号类型转化为该无符号类型。注意是在位数上，long和int都32位那么他们等级应该是一样的。

一个char加上一个float，那么char是先转int再转化为float，先扩展位。一个char加上一个long，那么char转化为long。从int开始的整型都可以转浮点数。一个int加一个float加一个double，那么由于不知道编译器那部分先运算所以其中的转化也是不知道的。不过如果要扩展位的话都是先扩展位。

int i;

const int &j = i;

const int \*p = &i;//这两个初始化发生了隐含的转换，先把值const再给过去。毕竟是两个类型了。

毕竟被认为成不同的类型呀。

57.转义字符：

newline \n horizontal tab \t alert (bell) \a

vertical tab \v backspace \b double quote \"

backslash \\ question mark \?（直接输出问号也没有问题） single quote \'(直接使用单引号也没有问题)

carriage return \r formfeed \f（Linux下等效于\v）

60.引用默认是左值引用(lvalue reference)，右值引用是C++11标准加入的多用于类里面以配合移动操作。

61.C++中的初始化就是在声明时赋值，声明后的都叫赋值。引用是要初始化的，所以不能进行再次引用。从根本上讲就是符号表的不可改变。

由以下程序证明，引用与宏本质差不多的

#include <iostream>

int main(void) {

int a =1;

int &b=a;

int \*c =&b;

std::cout << \*c << std::endl;

return 0;

}//尝试着走引用的地址或者sizeof一个double的引用看大小

62.把一个指针初始化为空有三种办法，其中nullptr是C++11才引入的，它是个能被自动转化为任何指针的特殊类型，而NULL则是个宏名被定义为int 0。C++没有了C那样自动转化的灵活了，C用void指针0来宏定义NULL。

63.对引用的读取与指针是从右往左读。

64.常量表达式是指值不能改变且在编译时就能定下来的表达式。constexpr是让编译器去判断这个是不是常量表达式。

66.结构的声明后面要有分号因为这个又可以是定义。

67.一般类型是适合直接硬盘操作的，string与vector之类的就是不可以直接操作。

69.vector与string比较重要的初始化方式：

vector<int> ivec(begin(int\_arr), end(int\_arr));。

字串函数加一个参数或者两个参数<=>string test6(test2,pos[,num]); //从test2中的第pos个位置开始，拷贝个数为num个字符，没有num就到顶呗，如果剩下的字符不足num那也仅仅是到顶。test2可以是数组或者指向C字符串的指针。

string test(point, n); //从指针的数组头开始的n个元素

string自然也可以使用两个迭代器（用了迭代器一般不会与数字的组合）

string test9(num,ch); //拷贝num个字符型ch到test9，ch接受的是字符，你可以用数字常量表示

string test10{'a', 'a', 0};

string test11{"aa"};//不等价于上面的，那个0也被放进去了，而字符串后米的\0被无视

70.用range for有个要求就是范围最多是有限的增加。也要小心扩容造成迭代器、指针的无效。

71.所有的库容器都支持迭代器，但只有一部分支持下标、迭代器的加减、大小比较操作。一般迭代器的比较只有==和!=。

74.int ia[] = {0,1,2,3,4,5,6,7,8,9};

auto ia2(ia); // ia2 is an int\* that points to the first element in ia

auto &ia2(ia);//ia2 还是个指针

auto i = 0, \*p = &i;是完全没问题的。//一句auto声明了不同的类型，不过基础类型都是一样的

76.空指针的算术操作有以下几个：加减0；两个空指针的相减(结果为0，ptrdiff\_t类型)。就算NULL在C++中被定义为int 0还是不可以相加，因为这是来自类型的限制。NULL+NULL只是编译的时候被编译器检测出来包一个warning错误，我分别使用了stdio.h和iostream头文件都是这种错误。并不影响运行，由此或许可以证明NULL的定义。

77.数组，vecotr，string的下标都不准为负，这几个类型都是unsigned。不过没了类型限制就可以为负，指针指向数组后，指针的下标就可以为负。

79.string的c\_str()成员函数返回的是const char\*，这是为了C语言返回一个指针(地址)，看样子还不是复制一个串那样，否则也就不用const了，这样的话一旦string析构就会导致这个指向一个垃圾内存。所以

char c[20];

string s="1234";

strncpy(c,s.c\_str(),20); //memcpy也行

这样才是正确使用的方法。

81.指针可以接受所有的地址，这句话里面的指针是包括了指针的指针。

指针不能与指针相加的原因就是他们确信这对你没有什么用就warning。

91.直接auto一个数组得到的是指针，因为还有赋值。不过使用引用的话，得到的是数组

93.移位运算由结合性可知，1<<2<<4即1<<6。

94.有或无符号型的long以及指针，在32位与64位上用的位数不同。

95.从对位的操作理解取余。联想一下分离一个整数中每一个数字。

96.返回0或1的函数不一定用if，可以用一条表达式，可以用强制类型转换也可以取余。

99."溢出"的范围是有限的，不会过0。我还以为可以直接一轮又一轮只要有个数够大，但这个数如果本身已经溢出了（这就是你类型选择的失败了），在取过来运算时就只取会取的部分。有符号溢出以0去判断，无符号则以自身去判断。（因为最大值与最小值有符号与无符号不同）。

100.函数里面的操作列表这个操作数()是无限操作数。\*既属于一元操作符又属于二元的，在不同上下文中有不同的意义。

对操作符的重载仅是其作用方面，不能改变它的优先级与结合性（定义优先级是编译器语法分析的时候固定了的）。所以\*用于dereference与mutiply是并不是重载。

101. C++左值与右值概念比C麻烦，左值可以是个object或函数（返回一个对象）。但粗略来讲就是右值是内容，左值是可以取到地址，右值只是来用，左值还可以改等等。除了一种特例外，我们能用左值去代替右值，用的是地址上的内容，但不可以用右值代替左值。

102.<<没有保证计算的顺序：

cout << i << " " << ++i << endl; // undefined

对实参值的计算顺序也是未定义的，由编译器决定（两种压栈方向），倒是可以人为修饰。

#include <iostream>

using namespace std;

int main() {

int a=2;

cout << a++ << a << ++a << a; //34444

int b=2;

cout << b++ << b; //23

return 0;

}

程序语言通常都规定了执行中变量修改的最晚实现时刻（称为顺序点、序点或执行点）。程序执行中存在一系列顺序点（时刻），语言保证一旦执行到达一个顺序点，在此之前发生的所有修改（副作用）都必须实现（必须反应到随后对同一存储位置的访问中）：

C/C++语言定义（语言的参考手册）明确定义了顺序点的概念。顺序点位于：  
1. 每个完整表达式结束时。完整表达式包括变量初始化表达式，表达式语句，return语句的表达式，以及条件、循环和switch语句的控制表达式（for头部有三个控制表达式）；  
2. 运算符 &&、||、?: 和逗号运算符的第一个运算对象计算之后；  
3. 函数调用中对所有实际参数和函数名表达式（需要调用的函数也可能通过表达式描述）的求值完成之后（进入函数体之前）。

小心：程序里有代码片段“...a[i]++ ... a[j] ...”，假定当时i与j的值恰好相等（a[i] 和a[j] 正好引用同一数组元素）

&&，||以及逗号是规定了顺序的，同时被编译器认为是一个顺序点。

注意：区分结合性与计算顺序(顺序点)。结合性适用于理解以及辨认表达式的。

没规定顺序应该比规定顺序会快，cpu可以多个流水同时执行

105.if (true == iVal) 要求val必须是1才能相等，因为true被转换为int而不是val被转化为bool。

106.再次说明最好把0赋值给指针,C++中，不过我觉得没有什么问题，因为他会自动将0从int转化为合适的类型。非0整数传给指针需要强制类型转换，浮点数强制类型转换也传不了。

107.a++比++a多一些工作，所以一般用++a。

熟练\*a++与\*++a

++\*iter;第一眼看++优先级高，但这里就是谁近谁‘优先’

109.格式：

finalgrade = (grade > 90) ? "high pass"

: (grade < 60) ? "fail" : "pass";

条件运算符虽然是右结合性，但是顺序是从左至右的。所谓的右结合性是指 (grade < 60) ? "fail" : "pass";结合成一个表达式并入到前面的条件运算符中。

#include <cstdio>

int main() {

int grade = 70;

(grade > 90) ? printf("1")

: printf("2"),(grade < 60) ? printf("3"): printf("4");

return 0;

}// 所以输出24

110.string s = "word";

string pl = s + s[s.size() - 1] == 's' ? "" : "s" ;

由于加号优先级高于条件所以相当于string pl =（（ s + s[s.size() - 1]） == 's'） ? "" : "s" ;

116.++!a会引发error:lvalue required asd icrement operand.

解释：!a是个表达式了

想到了decltype，但是++(!a)还是一样的结果

119.&&b会说error:label 'b'used but not defined，但b是定义了的，只是&b不是个object。左值引用要对变量进行。

b是a的别名，c是b的别名？其实我认为c不是b的别名，而是a的别名。

123.return 0;后面可以紧跟着又一个return 0;又没有限制只能一个return，当然可以咯。constexpr函数就不允许存在多个return，检测机制比较简单。

124.在IO库中定义了istream和ostream至bool的转化。参与判断的最后都会转换为bool类型！

126.cast是个危险的继承结构，用于强制类型转换，下面是格式：cast\_name<type>（expressioin）

static\_cast除了低水平const的相关类型的转换都可以用。如果强转了一个不匹配的指针类型，那么结果未定义。之所以说相关就是别struct变int，int变指针，老版本也做不到。

const int a =2;

int \*b =&static\_cast<int>(a);//错误，应为statci\_cast得到的不是左值，算是一个表达式，应该static\_cast<const int \*>(&a) 或者使用const\_cast

const\_cast就是对有低水平const进行丢弃。不过本身已经是nonconst也合法。

dynamic\_cast 转换失败会返回NULL，算是一种安全的转换

reinterpret\_cast将数据已二进制形态重新解释，没有对位动手脚。

老版本的强转：type (expr) //函数风格，要求这个type不是指针

(type) expr //C风格

131.存储性数组(比如说考试分数等级的数组)可以用来代替switch（自己写一个巧妙的hash函数就能将数据对应到数组的一个个索引上）。在数组的索引使用特殊的表达式。

133.switch注意：case要是const 整数的表达式。一个标签是不能真什么都没有，可以用空语句或者空块。在一个case中定义一个变量在另一个case中使用是可以的毕竟都在一个块中，但是它绝对不能被初始化不论是明显的还是默认的也不论是否同一个case，因为这会导致初始化可能被跳过。初始化会引发error:jump to case label还有error：crosses initialization of '类型 名字'。解决方法:用个block，定义被限定在block里面就可以初始化了。

If-else有时候涉及的是变量之间的比较，所以switch还是不能用来取代if-else（注意switch在汇编层面的加速）。swift中的switch可以用于替代if-else结构。

#include <iostream>

using namespace std;

int main() {

int a =2;

switch(2) {

case 1+1: //允许是一个常量表达式，会自动计算为2

cout << "1";

break;

}

return 0;

}

134.C++中while里面可以是个定义但必须只有一个变量而且带上初始化，但do-while不可以，条件运算符也不行，C也不可以，只是对while做了特殊的开放。

135.格式：for (decltype(s.size()) index = 0;

index != s.size() && !isspace(s[index]); ++index)

for循环那部分没有就在那部分多来几个空吧

140.goto向后跳会导致变量的销毁与重建。

goto end; //error:from here

int ix = 10; //crosses initialization of '类型 名字'

end: // error: jump to label 'xxx'

ix = 42;

然而C允许

C++认为但凡初始化都是有用的。

141.异常的抛出、捕获与处理:

异常类型是用于在throw和catch之间传递信息的。

runtime\_error是个有obj的标准类型，定义在stdexcept，旨在运行时观察异常，两种字符串之一的任何一种来初始化。定义在此文件里的class还有exception，runtime\_error中又分出：range\_error（内部计算时发生区间错误），overflow\_error，underflow\_error。logic error分出：domain\_error（没有结果的参数），invalid\_argument（不对的参数），length\_error（尝试创造一个超出限定大小的obj），out\_of\_range（参数不再预期范围内。例如在诸如array的容器或字符串string中采用一个错误索引）。他们分别继承自父类，而logic\_error与runtime\_error继承自基类exception。他们都只有一种操作what()返回一个const char\*，具体内容因类型而异，似乎默认初始化的那三个是编译器设定的字符串。

exception这个头文件定义了最通用的异常类exceptioin，这个obj是默认初始化的而不能人工初始化这样的还有new里面的bad\_alloc与typeinfo里面的bad\_cast。其余的都没有默认初始化，要用两种字符串之一去人工初始化。

对catch的寻找是先该函数的，找不到就caller函数的，...一层层向上。如果都没有找到最后移交给terminate函数并终结程序（会导致函数执行一半就被迫返回），这个函数的行为是系统控制的。

代码举例：输入两个数将它们相除：

#include <iostream>

#include <stdexcept>

using std::cin; using std::cout; using std::endl;

using std::runtime\_error;

int main()

{

int a,b;

while(cin >> a >> b) {

try{

if(b == 0) throw std::runtime\_error("b can't be zero!\n");//变量直接转为类的对象

cout << a/b << endl;

}catch(std::runtime\_error err){

cout << err.what()

<< "Please input b again" << endl;

continue;

}

}

return 0;

}

抛出未命名对象(右值)，用以命名对象去获取，得到的字符串就是.what()会返回的。如果抛出的仅仅是字符串，也就是throw "hhh";那么要用指针去接受catch char \*str;。

#include <vector>

int main()

{

std::vector<int> vec;

vec.at(0);

return 0;

}

这个程序编译通过，但在运行时会在控制台（通过命令行启动）发出out\_of\_range（超出范围的使用）。

#include <vector>

int main()

{

std::vector<int> vec;

vec[0];//仔细想想vec[0]的计算过程，无效域有效完全不影响。

return 0;

}

就像数组越界，编译没问题，运行一般也没有问题，大概是因为索引原理是基于地址的。

#include <vector>

int main()

{

std::vector<int> vec;

auto beg =vec.front();

return 0;

}

编译没问题，运行直接over，因为要对这个元素进行实质性的操作。不过应该只是访问操作吧。

145.C++函数：

C++的函数实现允许参数未命名，这也说明它在函数实现中不会被使用，但是你还是要分给它一个实参以做到一一对应。

函数的参数原型可以在任何任何一个scope中，这个同变量定义。函数原型可以多次声明，但是每个参数的默认初始化只能又一次，上个原型的默认初始化会在这个原型中继承。结合就形成了可变的默认参数。函数参数的默认初始化也有要求，只能是最右侧的才可以默认初始化，因为实参的省略是对右侧剩下的而言。注意：默认初始化是在原型之中生效。调用的时候受原型影响再去执行函数内部代码。不过可以方便阅读而在定义部分也加上。

函数的跨文件调用，1.cpp中只要有定义，main.cpp中要有加了extern的函数原型，extern不是必要的但是必须要有函数原型，两个文件链接起来。

函数调用需要的工作：寄存器内数值的保存于return后的重新加载，可能有实参的复制，地址的跳转（这个到很快，毕竟访问内存）。

形参可以是引用，这就相当于为实参取了个别名。

函数重载的时候要求参数类型是不一样的，而高水平const与返回值类型被无视的(因为要实现的仅仅是调用，根据条件根本就不能分辨他们)，因为入选切实可行的函数只有它。函数重载就是编译器选择一个更匹配的函数定义best match。滥用函数重载不利于可读性。结合const\_cast这种强转，可以改变返回值的低水平const。函数寻找的步骤：第一步，从最近的范围开始一步步扩大来找同名字的，找到一个后就在该范围里面找到所有该名字的(即所有该名字的可见函数，不可见的就称隐藏)又称之候选函数，注意的是就是在这个范围里面找哦即使其他范围有更适合的；第二步，在候选函数中选出可以按实参被调用的函数称之为切实可行的函数；第三步，判断类型的合适，选择最合适的(要考虑到参数类型可以转化，不是适不适合转化)：与其他切实可行函数相比，每个参数都不糟糕，且至少有一个自己更好的参数类型。如果第三步没找到就成ambiguous call错误。第一步或第二步没找到就是no match。

第三部匹配详细：

1.直接是一样的、数组或者函数转化为指针、两者参数差的是顶层const

2.差低层const(只能是没有底层的传给有底层const的)

3.差引用关系

4.数值类型之间的转换（包括char转int），指针类型转换（除了第一层的数组与函数类型转指针，还有数字0、nullptr转指针，以及非低水平const指针转void\*或者指针转const void\*）

5.符合类的类型转化（应该是指隐式转化了）。

PS：1.的那三个属于同一级别的。

函数模板在重载只有当完全匹配的函数找不到了就会找模板函数。

void f();

void f(int);

void f(int, int);

void f(double, double = 3.14);//只用在函数原型的时候告诉编译器就可以了，默认初始化只能从后米往前开始，因为使用的时候参数是从前往后对应的。

f(5.6); // calls void f(double, double)。

void manip(long);

void manip(float);

manip(3.14); // error: ambiguous cal

#include <iostream>

using namespace std;

void Print(double);

int main() {

void Print(int);

Print(3.14); //调用的是void Print(int);除非这两个原型在一个scope中

return 0;

}

void Print(double num) {

cout << num << endl;

}

void Print(int num) {

cout << num << endl;

}

实参中数组与函数会被转化为指针，但是直接cout输出函数是数字1。如果要输出一个函数的地址，需要在函数名前加一个强制类型转换——(void\*)func2。函数名并不是地址，不过它可以通过隐式转换为地址来使用。由于C++规定，非静态成员函数的左值不可获得，因此非静态成员函数不存在隐式左值转换，即不存在像常规函数那样的从函数到指针的隐式转换，所以必须在非静态成员函数前使用&操作符才能获得地址。

非main的函数返回的值会初始化一个临时变量。当函数结束时局部obj会变得无效，注意局部obj包括文字常量，所以如果要返回一个引用的话那一定是对人工分配(程序结束或free时释放)或者对实参的引用。和指针一样，。

返回引用的函数是lvalue（代表着那个变量么），一般返回临终值

我返回一个数组指针但是没有在函数的定义中进行return，编译器没有报错。测试的编译器包括vs2010、vs2015、vc++ 6.0、gcc等。编译器在对函数进行编译的时候，并未对函数是否能正常的返回做判断，从而可能导致函数的返回数值是一个不确定的值的逻辑问题：<https://blog.csdn.net/meanong/article/details/80107959>　确实是将eax寄存器中的值直接作为返回值，估计g++中要求string以\0结尾于是直接返回string会导致“segment fault”，但是返回string\*，可以正常输出一个乱码string。

参数是数组还会帮你自动转换为指针，返回值要用数组那种形式那么有点特殊。Type (\*function(parameter\_list))[dimension]，C++11为了简化这种，就有了种利用auto的新形式：把return类型移到函数的末尾，例如：auto func(int i) -> int(\*)[10];(你需要明白这里的\*为什么被括号包着，因为这优先级最高)。那么也就有了decltype：int arr[]={1,2}； decltyp(arr) \*func(int i)所以也就要求arr定义在使用decltype(arr)之前。在我看来三种方法中用复杂返回值用auto最好。C++14对函数的编写形式进一步简化了。auto(decltype) xx。

int (\*func(const int &i))[3] { //其实函数应该写成一个函数指针的形式，这样子就能够用[]表示返回一个数组了

}

一个constexpr函数有如下要求：返回值与每个形参都是文字常量类型，且函数体有且仅有一个return（多个return会被检测出来），不能有其他运行时才执行的语句。因为编译器要去替代。constexpr size\_t scale(size\_t cnt) { return cnt; }cnt是const返回值才是constexpr，如果是nonconst那也不会报错。

函数指针数组。

1. 静态存储方式的量不一定就是静态变量， 例如外部变量虽属于静态存储方式，但不一定是静态变量。有static的才是静态变量

static变量只能用于当前文件中（C语言限定了作用域），只会被创建一次

151.编译器-c选项就是只编译，生成.o文件

153.有些对象是不能被复制的，比如IO

157.指针是迭代器的一种,可以这么认为。指针视为狭义的迭代器。迭代器是具有类似指针行为的class template。也就是重载了operator-> operator\* operator++等操作符的类模板。指针只能用于某些特定的容器。而迭代器可以适用于所有容器。迭代器的设计就是为了数据结构的泛化。所以迭代器的适用范围更广。

159.除了在引用时，编译器不需要数组的第一个维度，大部分写出来有利于可读性。const int (&test)[] = {1,2,3} ; //允许编译器自己去推断

数组引用：int (&a)[3] =数组地址;

1. 对于C++，void main(),由于它不是当前标准强制的一个选项所以可能在某些系统上行不通。C++标准对觉得return 0;烦的人做出了让步，如果main函数没有return结尾就默认return 0;但是在我的g++中，main函数时必须放回一个int值，否则报错： error: '::main' must return 'int'。

161.可变参数数量的函数：

①对于后面参数都一样的情况，C++11的initializer\_list模板在initializer\_list头文件中，与const数组相似的模板类型,initializer\_list对象中的元素永远是常量值，我们无法改变initializer\_list对象中元素的值。其.begin()和.end()都是返回指针，没有.empty()。这个类是一等公民（first-class），由只能用{}这个语法进行初始化，而且只能由编译器来构建。适用于多类型的都是模板。这个应该与列表模板相似。

initializer\_list<T> lst;//empty

initializer\_list<T> lst{a, b, c...};

lst2(lst);

lst2 =lst;//那么{}也可以

.size()

.begin() .end()//本身每一个元素已经是const了

#include <iostream>//如果用""去include话就是在使用绝对路径

#include <initializer\_list>

void f(std::initializer\_list<int> lst);

int main(void) {

int a =1;

f( {1, 2, 3, 4, 5} );

return 0;

}

void f(std::initializer\_list<int> lst) {

for(auto beg =lst.begin();beg != lst.end(); ++beg)

std::cout << \*beg << " ";

std::cout << std::endl;

}

在C++11的时候，大家都想加上一个值列表的东西，就像{ value1, value2, value2... valueN }一样。一种想法是搞出一个新的关键字，给C++增加一个新的build-in类型。但是新的关键字很有可能会导致老的程序无法被编译，如果凑巧老的程序使用了那个关键字做名字。于是C++11的做法是，只是在标准模板库里面增加一个新的模板initializer\_list，然后让编译器遇到{1,2,3,4}这种东西的时候，隐式的转换成一个initializer\_list的对象。下面这段代码输出的是class std::initializer\_list<int>：

auto list = { 1, 2, 3 };

std::cout << typeid(decltype(list)).name() << std::endl;

二. ellipsis,C++ also has a special parameter type, ellipsis, that can be used to pass a varying number of arguments.。作者说他应该只被用在对C函数的接口这就是C里面的varargs。

void foo(parm\_list, ...);

void foo(...);

163.prvalue(pure value纯数值)指代一个临时对象(函数一般类型的返回值)、一个临时对象的子对象或者一个没有分配给任何对象的值。xvalue(expiring value临终值)指代一个对象，但是和lvalue不同，这个对象即将消亡，具体来说，xvalue是包含了右值引用的表达式。左值表示程序中必须有一个特定的名字引用到这个值是个obj，右值就是pvalue+xvalue。glvalue(generalized lvalue泛左值)即lvalue和xvalue的统称。

表达式

/ \

泛左值 右值

/ \ / \

左值 临终值 纯右值

167.EXIT\_SUCCESS这些不用加std::，因为他们是宏。C++可以直接使用诶。

168.递归的终止条件可以用等于号去写也可以用不等号去写。

在递归的函数里面用+1 -1而不是++ --

177.关于一个字节位数，C没要求,C++规定了必须不小于8比特。现在的计算机认为一个字节8位了。前期4bits/6bits/7bits，7位就够用了，但留了个位作扩展，而且8位方便进制间的翻译。

178.C/C++中inline是对编译器的请求，可以被拒绝。这种函数也是可以被放入头文件的。关键字inline 必须与函数定义体放在一起才能使函数成为内联(因为是对定义的作用吧)，仅将inline 放在函数声明前面不起任何作用相当于被无视。定义在类声明之中的成员函数将自动地成为内联函数。内联坏处：复制代码而占用了更多的内存。以下情况不宜使用内联：如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。增加了编译时间。

180.cassert里的assert，用#define NDEBUG或者编译选项 -d NDEBUG去关闭。不过g++不可以（应该指的是参数）。

181.预定义常量(两个下划线)：\_\_FILE\_\_,\_\_func\_\_,\_\_LINE\_\_

编译时期 \_\_DATE\_\_,\_\_TIME\_\_

182.typedef int FUN(int); <=> int Fun(int); typedef decltype(Fun) A; FUN不存在赋值，所以不用auto。<=>using FUN =int(int); 使用：FUN Fun;这样声明了函数

如果要是函数指针的话，用using要这么写:using FUN =int(\*)(int）;

如果说返回的是函数指针，那还可以用auto的尾随return：auto Fun(int)->int (\*)(int\*, int)

185.类：

class/struct A;是个forward declaration，这是类的声明，属于不完整定义，但能讲明A是个类名。所以它的使用也有限：能用指针与引用；能把它放在同样作用的函数原型中。所以我们能在类中又指向它自己的指针，因为已经声明了。(不能用逗号进行多变量声明，extern声明可以)。两者可以混合使用。

用struct或class都可以定义一个类，两者仅在默认访问水平上有个细微的差别：在第一个访问标识符前(包括了没有访问标识符时)，struct是public，class是private(类成员函数可以访问,不能进行外部访问，friend可以访问，C++中用private去封装)。一个private函数的使用：让public函数调用自己，从而衍生出多个版本的同名public函数。

Constructors成员构造函数是用来初始化变量的，名字就是类名于是它可以有多个用于重载，没有必要人为写上返回值，不能使用const去修饰。其中有个default constructor用于默认初始化不需要参数。我们没有写任何构造函数时，编译器自己写的constructor被称为合成的默认构造函数。我们要去写构造函数，但想有的保持默认就Sales\_data() = default;这样与参数列表后使用C++11的=default，后面没有函数体！我们自己设计的默认构造函数，如果有的数据成员没有被包括，那么就类内部初始化。构造参数列表可以为空，然后用函数体去弥补。构造数据列表可以另起一行写，构造函数列表没有代表初始化的顺序，编译器还是按照默认的从前到后来，如果你的构造函数列表的顺序不同于默认顺序，有的编译器会好心给你一个warning。构造函数列表是直接初始化的地方，可以用{}。注意：不要以为构造函数列表是一对一关系，参数列表只是得到这个值，怎么使用都可以。避免使用成员去初始化成员这样子你要是不清楚初始化就顺序就GG。可以结合函数参数默认初始化。

Sales\_data::Sales\_data(const string &s,

unsigned cnt, double price)

{

bookNo = s;

units\_sold = cnt; //对内部变量来讲不是初始化，对类还是一种初始化

revenue = cnt \* price; // 小心重排序？C语言里面不是有顺序点么，应该是没有问题的。

}

默认构造函数使用：//构造函数没有参数

Sales\_data obj2;

Sales\_data obj();//这个不对

有种委托构造函数，先调用自身，再调用所代表的A(对于A而言使用了，不过编译器可以判断出A，所以A的定义可以放在委托构造函数之后)。使用委托构造函数相当于构造函数的缺省参数模式

Sales\_data(): Sales\_data("", 0, 0) {}会先看有没有类似于初始化列表那样的传参再执行函数体再跳过去。

有种转换构造函数，是指那些只有一个参数，而且该参数又不是本类的const引用。如果有转换构造函数的话，那么那一个参数就可以隐式转化为class。创建了一个临时的class obj(这种转换不都是先临时的么)。编译器的自动转换只会转一次，解决办法是你把多余的步用个强转就行了。我们可以用explicit修饰构造函数的声明，去阻止隐式转换，但阻止不了强转。string没有用explicit，所以C风格字符串可以隐式转换。vector用了explicit，所以有如果类型没对上，会有警告。int a=2.1没有警告，int不是类，是基本类型。这种适合用一次的数据，更省内存。

除了构造以外，类的复制（赋值），析构都由编译器默认定义 或许应该是讲由标准默认定义

成员函数的原型必须要在class内，而定义可在外注意要加上class::且与原型完全一样，如果在内则自带inline属性。C++的成员函数如果要存取自身变量的话是要通过一个隐含的形参this指针，当我们调用这个成员函数时，this会被我们调用的class obj地址初始化。所以std::string isbn() const { return bookNo; }<=>std::string isbn() const { return this->bookNo; }。这个const修饰this指针(本身已是高水平)，低水平，使得this也可以被const obj地址初始化。我们也就称这样的成员函数位const成员函数，const成员函数也就不能去改变值喽。这个const会影响函数的重载。class中把函数定义在前，数据定义在后是可以的并不影响this，编译器会先编译数据再是函数声明再考虑函数定义。this就相当于一个隐含的参数。

成员函数默认内联再加上inline不会有错，默认public还可再加上public。因为这只是默认。

注意外部定义的函数成员其返回值在类名::前，所以如果返回的是类成员也要加上类名::。

friend类与函数(他们不是类的成员)可以访问类的非公开部分。这不是个函数声明，所要另外来个函数原型，不过许多编译器不要求这样。如果friend函数被定义在类里面，那么也隐含inline。其使用就是在声明前面加个friend，但不算声明也不是使用。friend类的成员可以存取本类的所有成员(相当于friend朋友类的函数成员)。friend关系认为不可传递。综上，就是认定一种不可以传递的关系。friend的使用破坏了封装性，造成了一定程度上的耦合，我曾经用过friend是因为类的构造后期出现问题，使用friend作为一种简单的弥补办法。swift变量创建之后就算是public，我真的不是很懂那种封装后却又提供了get、set方法。设置类的双元操作符的时候，必须要使用friend从而来操作类的私有成员。

友元类、友元成员函数、友元全局函数

类型别名受scope的限制(C中可能也一样)，在类中而且受到访问控制。friend其实不是声明，但是编译器有时候会当做算是。

有时候我们需要去改变某个数据成员，那么就是用mutable（这样它自身就不会是const，而且即使它在一个const(比如const成员函数)里面也可改)使之可改。

直接初始化要求类所有数据成员是public，常用的是列表初始化，但是要求初始化的参数按照默认顺序，不多不少。

文字类型的类(我想叫他常量类)要求所有的数据成员都是文字类型（包含其他constexpr类），至少有一个(必须要有)constexpr构造函数(合成构造函数与=default就是constexpr，一个constexpr构造函数必须要初始化所有的数据成员，当然也要满足一般constexpr函数的要求，使用constexpr就是为了在编译阶段“提速”，完成一些任务。

类的成员函数如果是constexpr，那么他也是const的。不过对于构造函数而言，不存在const这样，只有constexpr。

static函数成员，由于它与任何obj无关，所以也就没有this指针，也就不能用const修饰，它能操作的是static类型，因为它不需要创建对象就可以直接调用。static是修饰函数原型的，不需要在定义时再来个。static数据成员显然不是class的一个obj，它自己独立存在。当我们创建class时它也不由构造函数初始化(改进的方法是用constexpr去修饰，这样在编译时就把初始化问题解决了，但如果把它用在不能被数值代替的地方，比如因为类型要求，那么还是要外部定义)。

构造函数根据成员里面有没有指针分为浅构造与深构造，后者还要分配一个堆空间来容纳指针型数据所指的内容以此来避免内存中的错误。

类的访问限制：

原则就是范围是从开始到下一个访问限制开始或者}。能被范围限制的，比如说typedef的定义，数据成员，这些都会受到限制。

所以你可以先public，再private，再public。

返回一个\*this返回值类型是类的引用，或者返回this指针。这样子你就可以在一行中连着调用多个成员函数进行操作。

186.对于输入流istream和输出流ostream我们只能去引用，并不能复制。

188.根据C++标准，临时变量或对象的生命期在一个 完整的语句表达式结束后便宣告结束

193.头文件中虽然函数原型省略直接函数定义不影响编译，但不利于可读。不该把定义与原型都放在头文件中。从使用角度讲没有必要放入原型。

196.

struct First {

int memi;

int getMem();

};

struct Second {

int memi;

int getMem();

};

First obj1;

Second obj2 = obj1; // error: obj1 and obj2 have different types

编译器不会帮你去识别这个，所以被当成不同类型就错了，但是毕竟内存构造上相同，所以可以强制cast。可以用于多个异常处理里面。

199.typedef int Ａ;后就不能在同一个scope中在定义A了(有无typedef，加不加static一样的道理，如果允许的话不利于读)。不过有的编译器会不报错，允许再typedef定义类型，因为没有要求它们报错。不过对于using而言，不在同一个scope中也不行。

200.stdio.h中int snprintf(char \*str, size\_t size, const char \*format, ...)

将可变个参数(...)按照format格式化成字符串，然后将其复制到str中。可变参数可以是int，也可以是char数组。

(1) 如果格式化后的字符串长度 < size，则将此字符串全部复制到str中，并给其后添加一个字符串结束符('\0')；

(2) 如果格式化后的字符串长度 >= size，则只将其中的(size-1)个字符复制到str中，并给其后添加一个字符串结束符('\0')，返回值为欲写入的字符串长度。

若成功则返回欲写入的字符串长度，若出错则返回负值。

输出为一位小数： ios表示 IO stream

#include<iomanip>

cout<<setiosflags(ios::fixed)<<setprecision(1)<<sum;

205.直接初始化 与 赋值初始化（一种复制的方式）对应

默认构造函数会进行默认初始化与数值初始化。类需要它，可以是我们写的，也可以是编译器的合成默认构造函数。

默认初始化：直接new并且不带有圆括号或者花括号的初始化

数值初始化(可能被数值所初始化的)：①对数组列表初始化时，没指定的进行0或空数值初始化

②对静态变量进行0或空初始化

③Type(size)会进行0或空初始化等使用了圆括号的地方

int \*pi1=new int;   //默认初始化；\*pi1的值未定义

int \*pi2=new int(); //值初始化为0；\*pi2的值为0

int \*pi=new int(1024);  //pi指向的对象的值为1024

208.临时变量与引用：

事实上，临时变量是可以被作为左值(LValue)并被赋值的，请看下面的代码：

#include <iostream>

using namespace std;

class CComplex {

friend CComplex operator+(const CComplex &cp1, const CComplex &cp2);

friend ostream& operator<<(ostream &os, const CComplex &cp);

private:

int x;

public:

CComplex(){}

CComplex(int x1) {

x = x1;

}

};

CComplex operator+(const CComplex &cp1, const CComplex &cp2) {

CComplex cp3;

cp3.x = cp1.x + cp2.x;

return cp3;

}

ostream& operator<<(ostream &os, const CComplex &cp) {

os << cp.x;

return os;

}

int main()

{

CComplex a(2), b(3), c(4);

cout << (a + b) << endl;

cout << ((a + b) = c) << endl; //临时对象作为左值，666

return 0;

}

上面的程序编译通过，而且运行结果是：

5

4

临时变量确实被赋值，而且成功了。

所以，临时变量不能作为非const引用参数，不是因为他是常量，而是因为c++编译器的一个关于语义的限制。如果一个参数是以非const引用传入，c++编译器就有理由认为程序员会在函数中修改这个值，并且这个被修改的引用在函数返回后要发挥作用。但如果你把一个临时变量当作非const引用参数传进来，由于临时变量的特殊性，程序员并不能操作临时变量，而且临时变量随时可能被释放掉，所以，一般说来，修改一个临时变量是毫无意义的，据此，c++编译器加入了临时变量不能作为非const引用的这个语义限制，意在限制这个非常规用法的潜在错误。

1. 从标准来将=讲，constexpr 一定会是const。不过编译器实现上，从特性上看（比如在编译期间是否被直接替代的这种优化）并不是包含关系。

211.Screen& clear(char = bkground);这样的函数是没问题的。设置了默认参数，但是没有设置形参名。

214.IO类型：

iostream方面的有：istream，wistream；ostream，wostream；iostream，wiostream

fstream方面的有：ifstream，wifstream；ofstream，wofstream；fstream， wfstream

sstream方面的有：istringsteam，wistringstream；ostringstream，wostringstream；stringstream(这里的string是指in-memory string), wstringstream

鉴于cin等，为了满足大范围的字符支持，有了wcin等w系列。

对应的头文件就是上面列举的方面名字。

之所以可以无视不同流的差别是使用了继承。ifstream，istringstream是istream的子类（多态性），后面的同理。不过IO的继承是会引发错误的（继承以后进行输入输出重定向估计就没有问题了），有的系统内部错误，我们并不能修正。

之前讲过不能对IO类型进行复制。虽然限制了直接去复制，但是可以从底层自己创建一个。

由于读写可能改变stream的状态，所以这些类型的引用这些不能是const。 函数中的使用举例：istream& func(istream &is);

IO库里的状态：strm命名空间里的iostate(是个机器决定的一个整数值)：badbit(打断，是系统级错误)，failbit(程序级错误，可以修复)，eofbit，goodbit(值0，不是与badbit相对，而是与之前的这些错误相对)这四个都是constexpr值（应该是指状态位的位置固定），它们带bit说明是位。流的类型在条件表达式那里检查的就是iostate这个总的把。

std::istream::iostate state =std::cin.rdstate();//上面的strm只是代指那些合适的流对象，不过一个stream才有一套4个状态位, 不管什么情况，我们对它们分别进行输出failbit，eofbit，badbit、goodbit是4210。要看位应该用那几个函数。

std::istream::goodbit 似乎<=>cin.goodbit

类里面的成员函数：bad(), fail()（will be true if failbit or badbit was set）, eof(), good(), clear()（重置所有错误数值，返回值类型为void）, clear(flags), setstate(flags)（设置流的状态），sync()（cin.sync()有，没有cout的，目的是让流与数据源同步，也就是丢弃掉缓冲区中还存在的没有处理的数据，可以说是缓冲区刷新），ignore(int i[, char c])（我见过的是cin.ignore( 5, 'c') ，这个明显是提前设置的）从输入流中提取字符直至提取数到5或者遇到c字符。rdstate()（返回当前的strm::iostate值，rd是record的意思）。flags的类型是strm::iostate。

上面函数中flag表示某种状态

cin.clear(cin.rdstate() & ~cin.failbit & ~cin.badbit);是关闭failbit与badbit，clear可以是用来设置位的，不仅仅是清空变0，可以随你设置

把一个字符cin传给一个int是不行的。处理错误的循环输入不仅仅要清除掉错误状态，还要那个一直没被接受的东东拿走。

EOF会导致failbit、eofbit变1；如果把一个字符或者EOF输入给int，那么badbit不会变成1，failbit会。badbit说是系统级别的错误，但是这个例子就不对。

Ctrl+Z在一行中会导致后来的输入不被接受么，这种使用方法并不会对cin的状态造成改变。借此，我们可以把多行输入传给cin >> str;这一条语句。因为回车并没有起到flush的作用，不过在显示上还有换行的作用。就拿一个普通的字符吧，它还是显示了，但是不会到缓冲区。

一旦输入错误，比如说由cin把一个字符传给整型，cin状态位被改变（估计每一次使用cin前，都会自动看一下状态可不可用）要clear，同时错误的输入会留在缓冲区中要用sync()而不是ignore()，用ignore()清除错误输入于缓冲区中是无效的。

1. 每一个输出流都管理着一个buffer(我认为有的没有buff)，除了之前学过的的三个，会导致该缓冲区刷新的有（这两会自动刷新）：①使用unitbuf设置流的内部状态，从而在每次操作流后刷新缓冲区。默认cerr是这么设置的，具体使用：cout <<unitbuf 这样就设置成功了，解除就cout <<　nounitbuf；之所以叫unitbuf，感觉考虑到他的特性，用一个公用buffer就够了②流A联系到输出流B（一般是将输出流绑定到输入流上），则流A的读写操作都会刷新流B的缓冲区。默认cin与cerr是联系在cout上的（相当于cin.tie(cout)、cerr.tie(cout)），调用cin到这个语句的时候就会先去刷新cout在获取输入。注意：程序的非正常结束由于没有刷新缓冲区，那么数据就留在了缓冲区里，造成没有输出等的现象。经过实验，system("pause");也会刷新std::cout。

endl<=>"\n" << flush而ends<=> "\0" <<　flush（\0打印的话是空格）

cout << "hi!" << endl; // writes hi and a newline, then flushes the buffer

cout << "hi!" << flush; // writes hi, then flushes the buffer; adds no data

cout << "hi!" << ends; // writes hi and a null, then flushes the buffer。从打印效果上看，windows命令行下null字符就是一个空格（不可显示字符都是一个空格一样）。

tie（“同步”函数）的用法：返回已经捆绑住的ostream，如果没有捆绑，那就返回空指针。如果有参数(也是流类型的地址或者说指针)，那就是捆绑上新流，如果该流的指针是nullptr则意味着解除原有捆绑，之后返回一个指向原有捆绑的指针。也就是说tie有两种定义。

cin.tie(&cerr);// reading cin flushes cerr, not cout

流的对象是编译器自动生成的，对于操作符重载对流类型的引用就是引用自动生成的对象，一般的函数的对流对象的引用就像对一般对象的引用。

流对象都不支持复制。因此不能存在容器中。可是编译器不报错（已在g++中检验），那看来用到复制功能前都没有问题吧。

C++对于文件的打开关闭等成功失败的判断是看了流的状态，C是看返回值。其实实现上不是语言的锅，是函数实现的原因，但是这里有语言的设计思想。

219.fstream：

不同于iostream，所以fstream 在obj方面上是有更多的操作的，想法上类似于C的文件操作：fstream fstrm([name [,mode]);name没有就是没有连接文件，mode没有就是默认mode。这就是一种初始化，不过在C++11之前是不能用string只能C风格的字符串去指文件；这种类的成员函数有open(name,[mode])，用来连接文件。有open()自然也就有close()。还有检测要连接的文件已处于连接状态或者说打开的is\_open()。

销毁一个fstream obj时，close会自动被调用。

文件打开模式：in，out（以此模式打开文件默认是丢弃原有内容，所以为了保护内容我们还会使用app或者in），app（在写之前到达末尾。只要不是trunc意为truncate)就可设置为这个，一旦设置就处于输出模式），ate（一open就到达末尾）（at end），trunc（清空内容，使用前提是已被指定为out），binary。默认的模式就是in，out。可以结合使用：out|trunc<=>out,in|out...（我这边的模式都没有加上流名字，是不对的，模式可能是流的数据成员）。in和out不是对文件来说的，是对要与文件交换数据的对象说的。

ifstream用out模式也可以编译成功，但是不可以有输出到文件的操作。确实检查上没有java严格。

#include <iostream>

#include <fstream>

int main(void) {

std::ifstream f\_in("C:[\\Users\\Administrator\\Desktop\\Engli.txt](file:///\\Users\\Administrator\\Desktop\\Engli.txt)", std::ifstream::in);//in是ifstream内的，ifstream是std名字空间内的。如果文件使用中文名字，在windows下需要时ASCII编码。

std::string str;

std::cout << std::unitbuf;

while(f\_in >> str) {

std::cout << str << ' ';

}

std::cout << std::nounitbuf;

f\_in.close();

return 0;

}ifstream的>>读取和cin一样无视空白字符

用已连接文件的变量打开另一个文件要求原文件先关闭，否则虽然还是编译运行成功，但会没有输出等一个糟糕的结果。

225.输入还是输出是对要被输入输出的obj而言。而不是对流对象而言。cin也是这样。

226.sstream（用于处理整体行和局部单词，像是一个简化版的队列容器）:

变量可以直接被string初始化，虽然形式上是()直接初始化的形式，但是过程上是复制初始化，被设置成explicit。成员函数：str([s])，没有s就是return一个当前string的复制，有s就和之前讲的一样的赋值。

当一个in-memory string流被读取完毕，会触发end-of-file。

string line, word;

while (getline(cin, line)) { //还有一种 cin.getline

istringstream record(line);

record >> info.name;

while (record >> word)

info.phones.push\_back(word);

}

229.支持随机访问的例如vector，deque（双端队列），array（C++11加入的数组升级版，可能基本上没有太大的区别），string。支持随机访问都说明内存上分配是存在连续的。

不过list与forward\_list（C++11加入，没有size操作符）的随意插入与删除很方便。

大部分容器我们都要提供元素类型，但不是全部？不行，template里面必须要有内容。

所以用vector或者array容器而不是内置array。一般使用的最好都是vector，除非别的容器确实更适合。如果你有许多数值小的元素以及空间开销，那就不要用list和forword\_list。当你真的真的实在要插入或删除一个中间元素时，就先用list，而后再转为vector。

231.STL容器通用（具体允许的操作受类型的约束，我们这里就当作是用基本算数类型吧）：

类型别名：iterator，const\_iterator，difference\_type（C语言里面是ptrdiff\_t），size\_type，value\_type，reference，const\_reference。后面这三个的存在你就可以不需要知道其内部具体的类型

构造：C c; （由此引出一个复制初始化） C c1(c2); The constructors that take a size are valid only for sequential containers; they are not supported for the associative containers. C c(b, e);（array除外，b、e是指iterator） C c{a, b, c....};（有无等于号都可以） 见过用指针初始化vector

int ia[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 55, 89};

std::vector<int> vec(ia, std::end(ia));

赋值与交换：c1=c2; c1={a, b, c.....}; a.swap(b)或者用swap(a, b) swap函数其实就是一种交换，它比一般方法的交换快很多，因为元素并没有被交换，交换的是用于交互的数据结构（array除外）保证在一个固定时间内完成，比如说C中我把两个指向数组的指针值换一下。进行swap交换，那么对于string外的容器而言，迭代器，引用与指针都会跟着变化，都是有效的，但是书上说string的话就是不一定，可能会无效掉，我试了一次是有效的。这种思想可以放到不少函数中使用：string转number后如果只是为了输出的话，那么可以在这时调用的是string自己的输出 assign(b ,e)这种是迭代器复制 assign(i1)i1是初始化列表 assign(n, t)（数组没有assign）

大小：c.size()（forwoard\_list除外） c.max\_size()（返回一个值至少等于那个容器所能容纳下的元素，可能大于） c.empty()

增加元素（数组因定长除外）：这四个无返回值：push\_back(t)与emplace\_back(args) push\_front(t)与emplace\_front(args)相比于push\_back,emplace\_back可以避免额外类的复制和移动操作，所以应该用emplace\_back()。（从目前所知，可能是指emplace就是在原来的内存块上进行修改，push\_back到预留空间备用完的时候就会移动） ！！！insert(p, t)在迭代器p前面插入一个元素t insert(p, n, t)在迭代器p所指向的元素前面插入n个t，并返回插入部分首元素的迭代器，如果n=0，则返回p。insert(p, b ,e)用迭代器去插入元素在迭代器p前面（应该是前向列表的insert函数）！！！经过我验证string成员insert的参数不会是迭代器，就是索引。插入的位置就是索引位置开始。而且我没有试出3个参数的insert函数 vector、deque、string的增加元素有潜在可能会导致迭代器，指针，引用无效，因为对象发生了复制，内存没准重新分配了。而列表就会始终有效。 之所以往前面插，那么插入后就在这个位置了。

insert(p, {...})

elections.emplace\_back("Nelson Mandela", "South Africa", 1994); //没有类的创建

<=> elections.push\_back(President("Franklin Delano Roosevelt", "the USA", 1936));

由于inset是在选中的迭代器前面插入：

用insert在开头插入：

list<string> 1st;

auto iter = 1st.begin();

while (cin >> word)

iter = 1st.insert(iter, word); // same as calling push\_front

用insert在末尾插入：

不需要那个赋值。

list 的iterator不能+1，但是能++。因为函数符重载的支持。

assign的使用举例：

std::list<const char\*> l{"Mooophy", "pezy", "Queeuqueg"};

std::vector<std::string> v;

v.assign(l.cbegin(), l.cend());//只要元素可以转就可以了

删除元素： c.erase(args) c.clear() pop\_back() pop\_front()(pop同时返回出来的值？不，return void)

forward\_list的erase是特殊版的。erase(p)删除迭代器p所指向的元素并返回范围中的下一个迭代器(与它的特殊insert对应吧)，如果删的是off-the-end iterator那么未定义。 erase(b, e)范围性地清除迭代器b到迭代器e，当然不包括e所指的元素，如果b=e就不会造成元素的删除。 clear()就全清了。

删除一个元素，对deque<...>而言，如果删除的是元素不是开头与结尾，那么所有的引用、迭代器、指针都会变得无效；如果删除的是最后一个元素，那么就off-the-end迭代器会变得无效；如果删除的是首元素，那么都不会受到影响 。对于vector<...>，string则是指向被删除元素后面的那些变得无效。还有删除一个元素，编译器不会检查，而超出范围会就必导致运行出错。

erase()后容器obj的地址并没有改变。我在erase函数执行以后马上查看，发现除了被erase的元素出了问题，其他都挺好的。

比较操作符：==和!=是通用的。，剩下的四个对于unordered associative containers除外

直接获取元素，四个都是返回一个引用：back()（forward\_listm没有，或许应该加上万一对最后一个元素访问呢，不过话又说回来，这概率太小了） front() 还有索引[n]与索引函数at(n)。索引仅限顺序存储的容器，at()还可以用来检查部位是否有效，无效就抛出out\_of\_range。

不同于迭代器，对空容器使用back() front() 是超出范围的。迭代器的范围只至少也会存在一个。刚好用empty()进行检查。

back()返回的是引用，所以如果声明变量的话：

auto v2 = c.back(); // v2 is not a reference; it's a copy of c.back()

迭代器：c.begin() c.end() c.cbegin() c.cend()这四个是通用的，下面的反向迭代器是forward\_list除外reverse\_iterator，const\_reverse\_iterator的c.rbegin()，c.rend()，c.crbegin()，c.crend()。forward\_list只能前进不能后退，所以对于能用的iterator也不能用减量运算符来到退。

迭代器的运算：仅限string, vector, deque, and array

大小变更（当然array除外）：resize()

list<int> i\_list(10, 42); // ten ints: each has value 42

i\_list.resize(15); // adds five elements of value 0 to the back of ilist

i\_list.resize(25, -1); // adds ten elements of value -1 to the back of ilist

ilist.resize(5); // erases 20 elements from the back of ilist，如果有迭代器、引用、指针指向被删除的元素，那么他们就会无效。

顺序存储的容器，其实你增加或者删除了元素，只要做好对可能无效的东西的更新就可以了。C++优化了end()，因为它使用的频繁，插入、resize都是对末尾元素操作。

list迭代器的前进不是用++与+=这些，而是std::advance(listname, length);。那就有时候要麻烦点了，advance必然会对输入的iterator造成改变，传给他的参数必须要是个左值，不能直接是begin()的返回值之类的。

list不是顺序存储，所以迭代器没有<这些吧。也不能直接相减。

forward\_list有着不同的增加与删除元素的函数：

假设ABC这样排，因为删除B元素是要把A与C相连，到但是B到不了A。

于是前向列表没有被定义insert, emplace, or erase。取而代之的叫insert\_after（返回插入后的第一个元素的迭代器）, emplace\_after, and erase\_after（返回范围中的下一个迭代器）。我们要删除B，就要erase\_after(A);函数的重载与使用都是一样的，就是加了个after意义变了一下。

那么要对头一个元素操作就要一个off-the-beginning的迭代器before\_begin()，这个仅仅是forward\_list才有的。

forward\_list没有size()，list就有。所以要在forward\_list的末尾插入元素的话，只好把迭代器通过遍历到end()的前一个位置，所以遍历时要一个迭代器在前面看着是否下一步就是end()。

当要插入元素的时候，vector、string的内存分配详情：

就是如果要分配内存的话，就会一次性多分些，多余的部分用作保留以供给之后的分配。vecotor的增长效率常常比list、deque还好。

管理内存分配的函数：shrink\_to\_fit()放弃保留的内存，使真正内存大小就是size()\*一个元素的字节。不过这个函数被认为只是请求，不一定被执行 capacity()返回当前内存分的可以装下元素的大小 reserve(n)告诉容器内存要至少有n个元素的大小了。如果n不大于当前元素的大小，那么就什么都不会发生。 补充说明：resize()管理的是元素数量，这也是为什么要初始化的原因，所以只有当使用resize进行增大的时候可能会需要内存的分配。

std::distance(beg, s.end())

std::distance(oldVal.begin(), oldVal.end())

list与forward\_list操作:

merge(lst1, lst2)对已经按照从小到大的排序的lst2整合到也排好序(应该也是从小到大)的lst1。移完后，lst2为空。

merge(lst1, lst2, pred)增加了断言函数（大概是由于这是比较用的断言函数，同sort，所以就重载）。

sort()

sort(pred)

<algorithm>里的sort不支持list，list自己的成员函数sort已经支持使用pred。从小到大排序是：

int cmp(const int l, const int r) {

return l < r;

}//而不是<string.h>里面的cmp，有小于、相等、大于。

insert(iterator,value\_type)

remove(val)val算是一个predicate把。进行遍历，对是数值val的元素进行那个erase。

remove\_if(pred)使用断言函数，判断函数函数名字加上\_if

unique()

unique(pred)

splice(args)lst才有

splice\_after(args)flst才有

235.begin()与end()既可以返回iterator也可以返回const\_iterator，也就是说它是用了函数重载的。

236.容器：

容器有两种，一种sequential一种association。

一般都用vector<...>

array模板的使用：

array<int, 42>;构造一个数组

初始化方法：列表初始化这个与数组一样可以{0}、把一个同样大小的数组（非模板）给它，模板的当然也可以

同数组，列表只能用来初始化，不能赋值。不过这个是模板，可以被同类型直接赋值。//列表就不能用来赋值

对于除了Array的模板，swap直接交换了数据结构；对于Array，一个个元素交换。查看元素的地址是否变化可以判断。

1. 对于一般的指针,<<输出的是它的指针值（去掉0x的十六进制地址），对于指向字符串的指针输出的是字符串。这个函数对char \*的重载特殊处理了。

243. C语言中的库函数qsort、C++中的sort函数,二者是基于快速排序的函数。Java基本类型也是快排，但是复杂类型只能通过归并排序这种通用办法。

248.全局变量存在.data里面，然后会在连接的时候把目标文件的.data整合在一起再定下全局变量的内存，并向其中装入值。extern int a =2;是不对的，不管是还是C++，要么extern声明，要么就定义。

250.调用数组的迭代器的时候会把数组转换为数组容器，因为只有容器才有迭代器。所以把数组当做参数传递后，不能使用迭代器，用引用传参也不行，因为这时候是指针。

252.利用Win32API函数封装成类，可以进行精准计时。

http://www.cnblogs.com/zhangjing0502/archive/2012/06/25/2561574.html讲的是时间函数

255.#pragma once是编译器相关的，有的编译器支持，有的编译器不支持。在头文件的最开头加入，就能保证只被编译一次。

256.在类里面的public中的enum，若要使用要类名::enum名::具体常量名字

273.64位win7下，gcc环境，long long和unsigned long long和intmax\_t都是8字节的可以处理19位十进制数。别忘了long double类型。

64bits可以表示9开头的19位十进制数。

275.stdafx的英文全称为：Standard Application Framework Extensions（标准应用程序框架的扩展）。Windows和MFC的include文件都非常大，即使有一个快速的处理程序，编译程序也要花费相当长的时间来完成工作。由于每个.CPP文件都包含相同的include文件，为每个.CPP文件都重复处理这些文件就显得很傻了。为避免这种浪费，AppWizard和VisualC++编译程序一起进行工作，如下所示：

◎AppWizard建立了文件stdafx.h，该文件包含了所有当前工程文件需要的MFCinclude文件。且这一文件可以随被选择的选项而变化。

◎AppWizard然后就建立Stdafx.cpp。这个文件通常都是一样的。

◎然后AppWizard就建立起工程文件，这样第一个被编译的文件就是stdafx.cpp。

◎当VisualC++编译stdafx.cpp文件时，它将结果保存在一个名为stdafx.pch的文件里。(扩展名pch表示预编译头文件。)

◎当VisualC++编译随后的每个.cpp文件时，它阅读并使用它刚生成的.pch文件。VisualC++不再分析Windowsinclude文件，除非你又编辑了stdafx.cpp或stdafx.h。

在这个过程中你必须遵守以下规则：

◎你编写的任何.cpp文件都必须首先包含stdafx.h。

◎如果你有工程文件里的大多数.cpp文件需要.h文件，顺便将它们加在stdafx.h(后部)上，然后预编译stdafx.cpp。

◎由于.pch（预编译头文件）文件具有大量的符号信息，它是你的工程文件里最大的文件。

如果你的磁盘空间有限，你就希望能将这个你从没使用过的工程文件中的.pch文件删除。执行程序时并不需要它们，且随着工程文件的重新建立，它们也自动地重新建立。

277.gcc编译的C语言文件也可以不写return语句。int main()。

275.C里面的VLA数组使用也是有限的，我从一个文件中读取数据，并统计得到count，但是就不能用count去声明一个数组了。

276.你把一个小数除以0以后的输出：1.#INF00。如果你这0比较明显的话，而不是我这样还去看文件，那么编译器是会给出警告的。inf表示浮点数超出了能够表示的范围。

0.0/0 输出-nan

整数除以0会直接被中断掉（出现异常）。无论C还是C++。

277.puts(NULL);不会报错，但是是无效的。puts("");是有效的。

279.printf有缓冲区而fprintf没有缓冲区

280.以a、a+模式打开文件，所有的写操作都发生在文件的末尾，使用fseek函数可以使文件指针指向不同的地方，但是当发生写操作时，文件指针总是指向文件的末尾，所以文件的数据是不能覆盖的。

287.windows下的Cygwin使用math.h库不需要动态链接。

Cygwin使用了windows的CRT

mingw对于自带的默认直接连接了，不用我们再去链接。

291.windows下用文本模式打开与linux一样，但是如果是用二进制去打开在fread是就要考虑windows下是\r\n，linux仅仅\n。还有个区别就是中文默认形式，所占的字节不同。

293.::a，调用的是是全局a。如果没有定义全局a，就会在编译时报错。

296.const变量的初始化时间有两种可能：

const int i = Fun(); // ok: initialized at run time，除非这个函数是constexpr const int j = 42; // ok: initialized at compile time

297.

const int &ri = dval;

实际上要经过

const int temp = dval; // create a temporary const int from the double//我认为这个值并不是临时值。

const int &ri = temp;

299.如果你把声明和定义合在一起，那是可以的。修饰函数原型的参数默认初始化、static就要用来修饰定义。

如果你在声明和定义都进行了参数的默认初始化，那么就会报错说你：

main.cpp:17:14: error: default argument given for parameter 1 of 'int f(int)' [-fpermissive]

main.cpp:7:5: error: after previous specification in 'int f(int)' [-fpermissive]

304.指向同一个数组的两个指针之间的距离就是指一个经过改变到另一个的数值。

begin+(begin-end)/2 </=> (begin+end)/2

308.编译器对指向数组的引用检查更加严格，需要检查数组的维数。

int a[4];

int (&b)[4] =a;//编译器会检查[4]。

309.ival++ && ival可以用于判断ival是不是-1或0?不行，++对ival的值造成了副作用。

312.转换关系;

bool flag; char cval;

short sval; unsigned short usval;

int ival; unsigned int uival;

long lval; unsigned long ulval;

float fval; double dval;

3.14159L + 'a'; // 'a' promoted to int, then that int converted to long double

ival = dval; // dval converted (by truncation) to int

flag = dval; // if dval is 0, then flag is false, otherwise true

cval + fval; // cval promoted to int, then that int converted to float

sval + cval; // sval and cval promoted to int

cval + lval; // cval converted to long。之所以不哦也不用通过int，是因为直接符号位扩展就够了。

ival + ulval; // ival converted to unsigned long

usval + ival; // promotion depends on the size of unsigned short and int

uival + lval; // conversion depends on the size of unsigned int and long

不同类型之间都有隐转，尤其容易忘记数组与指针。

313.未定义的行为一般都可以用常理去推测，但是编译器的作者自作多情就尴尬了。

const\_cast后复制给一个指针，用这个指针去改就是个未定义行为。

323.这就是一个函数指针了：

bool (\*pf)(const string &, const string &);//先设置一个函数指针

pf = lengthCompare;

pf = &lengthCompare;//两种让函数指针指向函数的方式，函数在汇编中就是地址

bool b1 = pf("hello", "goodbye");

bool b2 = (\*pf)("hello", "goodbye"); //于是就有两者对应方式（当指针，当函数名）

直接对函数名进行cout输出，都是1。对函数名用了&后也一样。用了函数指针也还是这样。

下面是一个很吊的函数声明： 函数声明可以写成函数指针的形式

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

typedef函数类型：

typedef bool Func(const string&, const string&);

typedef decltype(lengthCompare) Func2;

typedef函数指针类型：

typedef bool(\*FuncP)(const string&, const string&);

typedef decltype(lengthCompare) \*FuncP2;

using F = int(int\*, int); // F is a function type, not a pointer

using PF = int(\*)(int\*, int);

auto f1(int) -> int (\*)(int\*, int);//这个是返回指针

decltype方式

auto(decltype) 自动按照return语句判断函数返回值类型

329.下面这串代码的理由是这两个类有关的仅仅是一个函数

class Screen {

// Window\_mgr::clear must have been declared before class Screen

friend void Window\_mgr::clear(ScreenIndex);//这要求这个函数已经在之前声明过了，因为要求这个类在之前声明过了

// ... rest of the Screen class

};

一个奇葩的友元函数顺序：

#include <iostream>

class TestB;//为了能让之后TestA::Print的声明中使用TestB

class TestA {

public:

static void Print(TestB &testb); //不可以直接定义，因为之前的TestB是不完整声明

};

class TestB {

public:

TestB(int a): num(a) {}

private:

int num;

double dnum;

friend void TestA::Print(TestB &);

}testb(1);

int main(void) {

TestA::Print(testb);

return 0;

}

void TestA::Print(TestB &testb) {//终于可以直接定义了

std::cout << testb.num << testb.dnum << std::endl;

}

335.windows下的IDE工程会自建一个makefile.win

339.fopen创建一个文件用fseek与fwrite去写，没被写的部分的都会是0，而且这些部分是逻辑上的，物理上的应该被优化了。ntfs是真的占有这部分0，ext4只是记录了有效部分。

341.smart pointers and move-enabled containers（和share\_ptr一样）, let us write more sophisticated classes without having to contend with the intricacies of resource management.

342.如果程序有多个异常要抛出，要借助不同表达式类型去区分，比如说自己定义的类

343.类类型转基本类型要使用类型转换函数了

345.内联函数又称内置函数

349.unsigned可以接受负数输入std::cin。支持类型的隐式转换。

352.system()在stdlib这个库中。

356.重提：VLA数组不能被初始化。

int main(void) {

int i;

cin >> i;

int arr[i];

//std::array<int, i>;//这里的i的位置必须要是常量才行

}

369.容器之所以叫容器就是因为它可以存各种类型，也就是模板。

378.标准库提供了三种顺序容器适配器：queue、priority\_queue、stack、deque（双向队列）.适配器是标准库中通用的概念，包括容器适配器、迭代器适配器和函数适配器。本质上，适配器是使一事物的行为类似于另一类事物的行为的一种机制。容器适配器让一种已存在的容器类型采用另一种不同的抽象类型的工作方式实现。例如，stack适配器可使任何一种顺序容器以栈的方式工作。由于一些容器适配器的的操作，所以不能使用array这个固定大小的，也不能使用forward\_list这个不会回退的，上面这两个C++11的是完全不能使用容器适配器的。statck只要求在末端的push\_back, pop\_back, and back，所以剩下的容器都可以转换。queue要求 back, push\_back, front, and pop\_front吧），所以能用的是list与deque。优先队列要求front, push\_back, and pop\_back。

之所叫适配器，是因为支持在实例化模板的时候传入其他类型作为底层实现

一般的数据成员：除了size\_type、value\_type还有container\_type

一般的函数成员：empty() size() swap()实现原理可能与容器的swap一样，不过这里还要考虑把容器的类型也交换了。

A<...> a; A<...> a(container);//会把容器的内容复制到容器适配器中。

支持所有的关系运算符

By default both stack and queue are implemented in terms of deque, and a priority\_queue is implemented on a vector.

当你这样定义时：stack<string> str\_stk; //str\_stk是基于deque容器实现的stack；

当你这样定义时：stack< string,vector<string> > str\_stk; //str\_stk是基于vector<string>实现的stack，其每一个元素类型都是string

Statck的函数：pop() push()复制或移动还可以用emplace()构造一个top element top()

queue、priority\_queue函数：pop()去除最front或者优先级最高的元素 front() queue是back()，priority\_queue是top() push()或者emplace()

C++中优先队列默认底层使用最大堆实现，如果要实现最小堆，最后面的比较函数直接使用greater<int>：



我一直以为在pop的时候就直接由pop出的元素的返回值，没想到被拆开来了。

380.容器的generic algorithms(对于所有，所有容器)：

大多数定义在algorithm，有一些在numeric。

它的实现，应该不仅仅借助了操作相似的迭代器，因为有些迭代器没有<这些并不能满足一些算法。基于迭代器，所以这些算法不能引起容器大小的变化。

find()话说string本身就有一些find了。

#include <vector>

using std::vector;

#include <iostream>

using std::cout;

using std::endl;

#include <algorithm>

using std::find; //就算是函数也是std::名字，不可以加上()。当然其实不用加上，他们不是std独有的。

int main()

{

int val =42;

vector<int> vec(10, 0);

auto f =find(vec.begin(), vec.end(), val);

if(f == vec.end()) { //在这里，如果没有找到就返回off-the-end迭代器，相当于到顶了。其实应该传cend()

cout << "a" << endl;

}

}

find不仅仅适用于正向迭代器，还可以用于反向迭代器，看来进行了函数重载。

count函数参数和find一样，返回的是出现val的次数。

除了find、count这两个只读函数，还有accumulate()、equal(），这个函数定义在numeric中。功能是累加，也是三个参数，前两个用来指出范围，最后一个是起始值。这第三个参数，可以是整数，也可以是浮点数，做了函数重载。不过如果是不能相加的类型，那么就会报错。equal()返回的是bool类型，功能就是至少性的比较，一样就返1，不同就返0，是按照元素一个个比较过来的。

equal(roster1.cbegin(), roster1.cend(), roster2.cbegin());//roster2从begin()返回的迭代器开始，至少做到有roster1

像equal这样，用一个迭代器表示第二个东东，那么会认为第二个东东至少和第一个东东一样大，进行比较的话就是按照第一个东东的参数来。所以如果的那个第二个东东是第一个东东的子串时，那么就会造成迭代器超范围的访问。不过我试验的时候，还是可以正常使用，甚至我传了一个本身就在范围外的迭代器过去，或许是因为只读，不过写的函数超范围在试验中也没有问题，写是确实写了，只是没产生什么影响。

std::cout << std::equal(roster1.cbegin(), roster1.cend()+2, roster1.cbegin());//返回1

std::cout << std::equal(roster1.cbegin(), roster1.cend(), roster1.cbegin()+13);//返回0

有写操作的通用函数：

fill()和find一样，三个参数。功能是把迭代器选中范围里面一切元素用val去写。

fill\_n()参数：pos，n，val。就是把迭代器改成了索引与数量控制。

copy()自然是三个迭代器那样的参数咯。返回值是指向最后一个复制过去的元素的迭代器（因为copy也是从一边到一边）。replace是一种copy，不过函数比较灵活。

int main()

{

vector<string > lst, lst1(3);//unique\_copy是copy之后去除重复元素

lst.assign({"aa", "b", "cz", "aa"});

sort(lst.begin(), lst.end());

unique\_copy(lst.begin(), lst.end(), lst1.begin());

for(auto str : lst1) cout << str <<endl;

cout << lst.size() << endl;

return 0;

}

replace()这里的replace是通用函数，有四个参数，前两个用迭代器表示范围，第三个是被指定的数值，第四个是替换用的数值。这是在原有上改变。

replace\_copy()参数与replace(通用函数一样，只不过就是先copy全部，建立一个新的容器，再replace。

sort()借助<来做到排序。会从小到大排列。

unique()把排序后的容器，删除重复的元素。看来原理就是把元素与该元素后一个比较，如果是，就把后面那个不是的移到这个位置。返回一个指向有效部分范围的off-the-end迭代器。之后要进行erase，使得这个成为真正的off-the-end迭代器？？？

back\_inserter（）返回值可以视为后面插入的迭代器。他会让off-the-end后移一格。这个函数并不是成员函数，但也算是通用函数。这个函数可以与填充函数结合。back\_inserter改变大小并不使是迭代器出现问题。

vector<int> vec;

auto it = back\_inserter(vec); //it被设定为后面插入的迭代器

\*it = 42;\*it=23;//插了两下

有时候我们想要的并不是按照数值大小来排序，可能是string的长度这些别的，所以有一种sort，它有第三个参数，而且被称为断言 predicate。断言就是从表达式中找到一个能够用作条件的值。这个值可能是一元的，也可能要二元（两个参数才能表示出来）.两个参数的sort,predicate就是<。我们用函数来实现自定义的断言，用法到和qsort()中的cmp()一样。只不过参数不限制于void指针，返回值的话，例子中是bool。借助这种新的sort可以实现降序排序，使用predicate为>即可。

stable\_sort()用法与三参数sort一样，但是遇到两个一样的元素，不会对其交换位置。对结构体排序时就与sort不同了。带有stable的函数可保证相等元素的原本相对次序在排序后保持不变

partiton() 也是三个参数，两个表示输入范围，一个是predicate。按照predicate把范围重新排列成true和flase两部分，true部分在前，false部分在后，返回的是true部分的off-the-end迭代器。它还有个stable\_partition()。

find\_if()就是上面的三个参数。下面这个函数不能成为find\_if的断言条件，因为它是双参数函数，而find\_if是一元断言，需要使用bind函数.

bool check\_size(const string &s, string::size\_type sz)

{

return s.size() >= sz;

}

count\_if(）就像find\_if一样。

for\_each()那三个参数，功能是对每一个元素调用第三个参数所指的函数。

class PrintString {

public:

PrintString(ostream &o = cout, char c = ' '):os(o), sep(c) { }

void operator()(const string &s) const { os << s << sep;}

private:

ostream &os; // stream on which to write

char sep; // character to print after each output

};

for\_each(vs.begin(), vs.end(), vs.PrintString(cerr, '\n'));

transform()函数，接受四个参数，前面两个是迭代器指明input range，第三个是一迭代器指明目标地，第四个参数是个predicate就是转换函数。功能：把输入范围里面的每一个元素都经过转换函数，把它传给目的地。高级一点的for\_each。

reverse()两个参数来表示作用范围

reverse\_copy()加一个表示输出目的地的迭代器

remove\_if()两个迭代器参数，既然有if那就要有predicate。

remove\_copy\_if()两个输入迭代器，第三个参数是输出迭代器，第四个参数就是predicate。

replace系列通用函数的使用：

replace(beg, end, old\_val, new\_val);

replace\_if(beg, end, pred, new\_val);

replace\_copy(beg, end, dest, old\_val, new\_val);

replace\_copy\_if(beg, end, dest, pred, new\_val);

PS：数组的迭代器就是指针，所以有时候以指针作为参数传过去。

roster1.end() +=2;改变的是临时变量。

382.vector<int> 与vector<long> 不能比较，虽然int可以转为long。

386.callable objec包含t可调用对象:函数、函数指针、重载了调用操作符的类、Lambda Expressions（Lambda表达式基于数学中的λ演算得名，就是匿名函数，可以实现闭包，由C++11引入）

Lambda Expressions：

结构：[capture list] (parameter list) mutable throw() -> return type { function body} -> return type 可以别省略

使用举例：

auto f = [] { return 42; }//可以有/无那个空着的()。

cout << f() << endl;//上面表明了参数列表为空

默认情况下，即捕获字段为 [] 时，lambda表达式是不能访问任何外部变量的，即表达式的函数体内无法访问当前作用域下的变量。如果要设定表达式能够访问外部变量，可以在 [] 内写入 & 或者 = 加上变量名，其中 & 表示按引用访问，= 表示按值访问（也就是复制一个，但是是只读的），变量之间用逗号分隔，比如 [=factor, &total] 表示按值访问变量 factor，而按引用访问 total。不加变量名时表示设置默认捕获字段，外部变量将按照默认字段获取，后面在书写变量名时不加符号表示按其余变量的设置捕获（默认捕获设置 是“按值复制”），比如下面三条字段都是同一含义。书上说我们只把它用于非静态变量，所以不用与std::cout，明明是因为外面的局部变量里面可以直接用。下面是他使用的三种格式：

[&total, factor]

[&, factor]//factor按值访问，其余都&访问

[=, &total]//这个&不可以没有

使用mutable的话，那个()是必需的。

auto f = [v1] () mutable { return ++v1; };

389.size（)返回值的类型就是size\_type

391.解决在一元断言的函数中使用两个参数的Lanmbda expression：（C++11）

在functional头文件里面定义了一个bind函数，这个函数可以当作一个通用函数适配器。我们借助这个函数传递一个表示参数大小的check\_size。

auto newCallable = bind(callable, arg\_list);有点像构造函数里面的代理构造。当我们调用newCallable的时候就会调用Callable，并且把arg\_list传递过去。

arg\_list：你用\_n(n是整数)，这样表示出来的参数都是来自newCallable的参数。

auto check6 = bind(check\_size, \_1, 6);//\_1就是来自check6的第一个参数。check\_size是两个参数的函数

check6("a");//这就是使用。

auto wc = find\_if(words.begin(), words.end(),

bind(check\_size, \_1, sz));//\_1在placeholders的空间中，placeholders是std内。所以using std::placeholders::\_1;也可以using std::placeholders

在bind中的参数都是copy的，而在bind中要用流的话应该用ref()，比如ref(os);因为如果直接用&os，那么在bind函数里面就直接被转换成os了，就会以os给Callable。如果要用const的话就cref()。

在之前的C++标准中，用bind1st，bind2nd这两个函数来进行固定参数数量的bind。

392.revisting iterator（就是其余种类的iterator）：

Insert iterators、Stream iterators、 Reverse iterators、 Move iterators

Insert iterators有的操作：\* ++ =

auto it=back\_inserter(lst);//lst是一个vector<string>//it始终与最新的off-the-end iterator保持同步。

it="aa"//<=> \*it ="aa" 因为字符串取地址还是自己

除了back\_inserter还有front\_inserter。其实最一般的函数是inserter，这个函数的第二个参数就是指出目标容器的插入位置，最后哦就是插在起那面。使用inserter函数在某一个元素的迭代器前面插入元素，跟据这个函数的习惯，那么这时候还是指向这个元素。除非是使用begin()+3这样的表达式。

我们能用back\_inserter的前提是可以push\_back，用front\_inser的前提是可以push\_front。应该使用emplace()系列。

Stream iterators：虽然流不是容器，但是也有iterator。之所以要有它是为了可以使用通用函数。在iterator头文件中。感觉就像迭代器那样使用流。

istream\_iterator操作：istream\_iterator<T> in(is); istream\_iterator<T> end; in1==in2（同样的类型，都是off-the-end iterator或者连接的流有同样的输入） in1!=in2 \*in in->term in++,++in

ostream\_iterator操作：ostream\_iterator<T> out(os); ostream\_iterator<T> out(os, C-sytle string);然后再输出后面就更上这个字符串，比较常用的是在后面加一个"\n" out << val \*,++对out而言什么也不做，因为给cout的东西都会直接被刷新输出，但是还是有这个操作符。输入的话到被认为是一个序列。

ifstream in("afile");

istream\_iterator<string> str\_it(in); // reads strings from "a file

istream\_iterator<int> in\_iter(cin);

istream\_iterator<int> eof; //默认表示eof

while (in\_iter != eof)

vec.push\_back(\*in\_iter++);

istream\_iterator<int> in(cin), eof;

cout << accumulate(in, eof, 0) << endl;

istream\_ierator直接读取流中的内容，只能确定是在使用之前。一般是不会出问题的，举个出问题的情况：假设还没有读取流中内容，我们就直接销毁了这个迭代器。而这会影响之后输入的读取。

Reverse interator:

注意下这种迭代器的end()是off-the-end也就是说从正向来看的话就是在第一个元素的前面。

396.类的补充（11章有必要重新看一遍）：

static修饰的那个静态成员是不能同时用类的内部初始化的。ISO C++ forbids in-class initialization of non-const static member 'Test::a'，从提示来看，还要加上const（静态成员应该在运行前就初始化了）

class Test{

static const int a =0;

};

类的初始化列表里面可以是bedroom(bedroom)这样子同名的，因为括号外面的被指定为参数，括号里面的被指定为类的成员。如果有的人觉得影响可读性的话，就可以bedroom(bedr)。

自己写那些构造函数，都加个default版本

struct A {

int a,b;

A(): a(1) {}

};

int main()

{

A a;

cout << a.b;//b不会被初始化为0，因为已经没有synthesis默认构造函数。

return 0;

}

Classes control these actions through special member functions: the copy constructor, move constructor, copy-assignment tor, move-assignment operator, and destructor。如果一个类已经定义了复制构造函数（浅复制）、复制赋值操作、析构函数。那么就不会再定义synthesis移动构造函数与移动赋值函数。这时候，复制构造就会取代移动。只有当编译器没有定义任何复制操作的东东，非静态成员可以移动，才会有synthesis的移动系列。

Copy constructor的参数应该是这样：

class Foo {

public:

Foo(); // default constructor

Foo(const Foo&); // copy constructor

// ...

}; 第一个参数必须是引用，如果不是引用就是一个死递归，而且基本上都是const引用。其他参数都要有参数初始化。如果程序员没有自定义复制初始化，就会有一个synthesized 版本的copy构造函数，是把类的非静态成员一个个复制到创建的对象中；否则被定义为=delete。

string null\_book("9-999-99999-9"); //跳过复制构造函数

</=>string null\_book = "9-999-99999-9";//这个比上面应该是多了一些操作。先用复制构造函数建立一个临时对象，再用重载的=把临时对象传给新的对象。

下面这个代码有6处复制：

Point global;

Point foo\_bar(Point arg) // 1

{

Point local = arg, \*heap = new Point(global); // 2, 3

\*heap = local;

Point pa[ 4 ] = { local, \*heap }; // 4, 5

return \*heap; // 6

}

复制赋值操作:

需要判断是否是自己赋值给自己。

Object& operator=(Object& object){

if (&object != this){

\_resource = new string(\*object.\_resource);

}

return \*this;

}

deleted functions：

有的类不允许复制操作，比如说流对象，不然就是多个对象对同一缓冲区进行操作。我们就在函数声明后面加上=delete

=delete不是=default，它可以用于任何函数。这个东东告诉编译器，被声明的函数，我们并不会去定义。

不能在用了default后再用delete。也就是两个不能同时使用。

如果你的析构函数是delete的或者不可访问的，那么synthesis默认构造函数，copy构造函数（也就是C++11之前的所有复制操作中的构造函数）会被定义为delete。

Destructor：

数据成员是按照初始化的倒序进行析构。

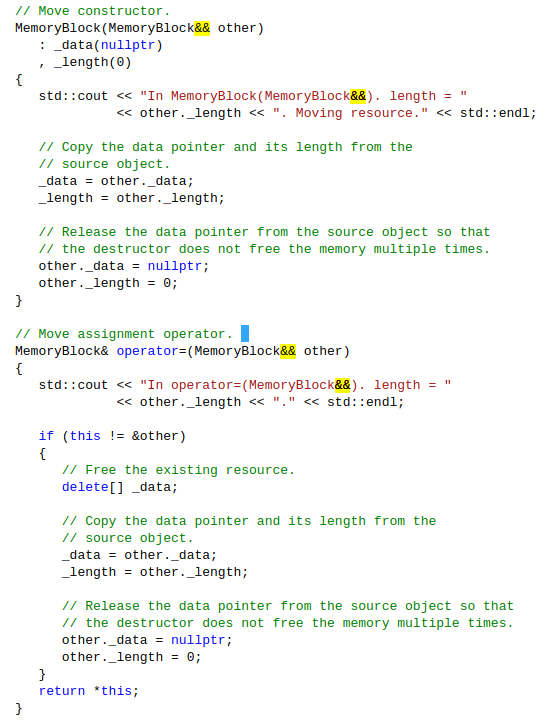
synthesized destructor：

~Sales\_data() { }//也可以不要{}用=default，那些构造函数也可以这样使用，因为他们可以是synthesis。

move constructor and move-assignment operator是C++11加入的类基本操作。这两个操作是基于、库函数。X x, x2 = std::move(x); // uses the synthesized move constructor。

在声明类的move构造函数时，第一个参数虽然是引用但是是右值引用，这就告诉编译器我们除了会对这东东进行赋值与析构以外不会再干什么。所以使用了move以后，我们不能对值做任何有效的假设。在移动后，我们要让原对象处于可安全析构的状态（指针什么的都赋值为空）。

move构造函数并不会引起内存的分配，他会接管一切。由于不会去分配，所以move构造函数不太可能抛出异常。但是我们如果不显示表明的话，编译器就认为可能抛出异常。书上说我们应该给这个构造函数的声明与定义都加上noexcept，这样相当于确定不会（不再存在可能让）move构造函数抛出异常。对于要分配新内存空间的这种移动或者复制：在构造时出现异常，复制构造的话抛出异常还可以救，但是移动构造时抛出异常，对于vector，不能满足自身保持不变的特性（这与为什么会有这种要求，没人给我解释...）。于是，你如果不是用noexcept，那么编译器就会一直使用复制构造函数，因为它才符合特性。



template<class T>

void swap(T& a, T& b)

{

T temp(std::move(a));

a = std::move(b);

b = std::move(temp);

}

移动系列synthesis被delete：

当定义了复制系列时或者没有定义复制系列但是移动系列编译器无法自己生成。（包括synthesisi的复制系列）

类成员是const或者引用但是没有被初始化。（从值的给出与接收）

如果已经被初始化，还能被移动系列改变么？

析构函数是delete或者无法访问。

当一个类里面既定义了复制系列，也定义了移动系列，那么到底调用哪个看右操作数是什么：我们在operator=的实现中一直使用类的拷贝，那么这个重载对复制系列是适用的，对移动系列也是适用的，进行了2合1，由于函数重载，复制系列管左值的拷贝，移动系列管右值的拷贝，或者对左值使用了move函数。

396.5 emplace底层使用placement-new构建元素。

397.内存操作的这些全在memory头文件中

为了更合适地使用动态分配的内存，C++有两种智能指针的方法。智能指针会保证动态分配的内存得到自动释放。两种智能指针分别是shared\_ptr、unique\_ptr，还有一个weak\_ptr是一个weak 引用向被shared\_ptr管理的指针，本身没有重载指针的\*、->操作运算符，但是提供了一些成员函数，用于协助share\_ptr处理。

当异常发生的时候，用一般指针分配内存的话，可能会导致内存泄漏，比如说下面这个函数：我们并没有外面的指针可以处理里面的内存分配，而smart poniter在退出域的时候就会导致变量的销毁，然后内存自动释放。

void f()

{

int \*ip = new int(42); // dynamically allocate a new object

// code that throws an exception that is not caught inside f

delete ip; // free the memory before exiting

}

shared\_ptr与unique\_ptr除了一般的指针操作外还有的共同操作：

成员函数swap() get()返回指向p的一般指针（p的地址），要注意这个地址所指向的部分没有被自动删除。

shared\_ptr还有的操作：（q是指内建的指针类型）

初始化操作：shared\_ptr<T> p(q)q是一个来自new的指针

shared\_ptr<T> p(u) u是unique\_ptr类型,p会接管u所指的区域，然后u就变成了null pointer

shared\_ptr<T> p(q, d)q所指的内存被p接管，q一定要能被转换成T\* 类型，d就是用于删除的可调用对象。

shared\_ptr<T> p(p2, d) p是p2的复制，d是用于删除p的可调用对象。

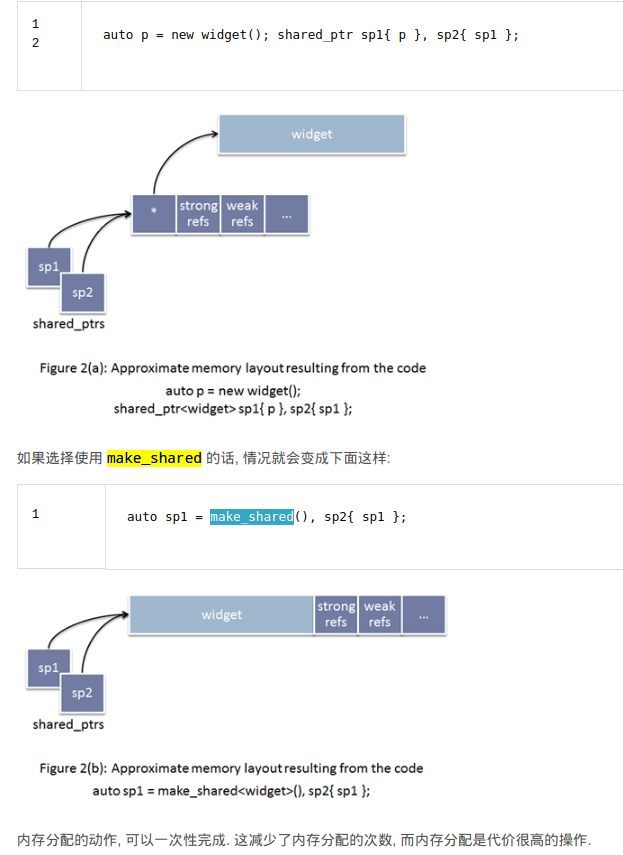
成员函数：unique()如果p.use\_count()是1就返回true use\_count()返回当前共享这指针的object数目，这个用于调试时使用，它的使用可能比较慢。

swap(p, q)<=>q.swap(q) p.reset() p.reset(q) p.reset(q, d) 首先p指针的值会被改变所以如果p是唯一一个指向那部分内存的指针，那么内存就会释放。加上一个q后，那么p就会复制q的值，有个d的话，那么就会调用d释放q。小心使用。

一般指针到shared\_ptr指针的转化仅限于显示构造，它的构造函数是explicit。不建议让一般指针指向智能指针的区域，因为我们不会去思考智能指针什么时候会放掉这块内存，而且我们要注意不能去把那个一般指针释放掉。

最安全地分配和使用动态内存的方法是make\_shared()。就像calloc一样，这个函数会分配并初始化内存，返回一个shared\_ptr，同时在内存分配上有点特殊：

shared\_ptr<string> p4 = make\_shared<string>(10, '9');//默认初始化



对基本类型使用share\_ptr；

int a = 10;

std::shared\_ptr<int> ptra = std::make\_shared<int>(a);

动态管理内存的方法还有：

new操作符格式如下：

（1） 指针变量＝new 类型；

（2） 指针变量＝new 类型(初值参数)；如果没有初值参数，就是数值初始化吧。

（3） 指针变量＝new 类型[数组元素数]；

new delete关键字

string \*ps1 = new string; // default initialized to the empty string

string \*ps = new string(); // value initialized to the empty string

int \*pi1 = new int; // default initialized; \*pi1 is undefined

int \*pi2 = new int(); // value initialized to 0; \*pi2 is 0

auto p1 = new auto(obj);

当new分配内存失败就会抛出异常bad\_alloc。不过你可以用另外一种形式的new从而不抛出，返回空指针：

int \*p1 = new int; // if allocation fails, new throws std::bad\_alloc

int \*p2 = new (nothrow) int; // if allocation fails, new returns a null

delete对同一指针所指的内存块进行两次，第一次释放内存，第二次没准就是破坏数据了（这内存又被什么什么使用了呢）

再被delete后，这个指针的值还是那个地址块，我们称这个指针为dangling pointer悬挂指针。为了避免误操作，我们最好马上地把它变成空指针。

delete 操作符的格式为：

delete 指针变量; // 释放指针变量所指向的动态内存

delete []指针变量;// 释放为数组分配的动态内存

例如：

delete p;

delete []p1;//删除的是内存，指针变量并不属于他管理的范围

345.move库函数定义在utility头文件中：

std::move接受一个参数返回该参数的右值引用。同时相当于保证之后不会使用这个参数。

345.5 右值引用只能绑定到临时对象上，下方代码绑定到表达是临时的计算结果上

int i =4;

int &&rr = i\*24;

347.在旧的标准中，我们没有办法阻止对右值的赋值，于是在新的标准中为了向后兼容，也就没有认为这是错的。

string s1,s2;

s1+s2="wowo";

为了阻止这种写法，可以使用引用限定符（reference qualifier）。书上的例子是原型和定义的函数头都加上了引用限定符。不过看样子引用限定符并没有被windwos下的cygwin支持。

struct A{

A(const A& r): a(r.a) {}

A(const A&& r): a(r.a) {}

A& operator=(const A r) &;//表示使用=的对象一定是左值，还有个引用限定词是&&

private:

int a;

};

A& A::operator=(const A r) & {

a=r.a;

return \*this;

}

struct Foo{

Foo sorted() &&;//对自己操作？？

Foo sorted() &;

private:

vector<int> data;

};

//因为是右值，所以可以对原址进行排序

Foo Foo::sorted() && {

sort(data.begin(), data.end());

return \*this;

}

//因为是左值，所以不能对原址进行排序

Foo Foo::sorted() & {

Foo ret(\*this);//进行一次复制

sort(ret.begin(), ret.end());

return ret;

}

348.重载运算符函数参数的数量与该运算符的运算数量要一样。不过如果重载运算符是成员函数时，this作为左操作数，于是参数就会比运算对象少一个。

除了函数调用运算符以外其他运算符的重载函数都不能参数初始化。

除了::、.、\*、?:这四个以外都可以重载（包括new、delete），这四个特殊，不用重载。去除了?:那么要么一元要么二元。->、&没有被禁止重载，觉得设计不合理

有些重载运算符是成套的，有==就该有!=。有+就该有+=。一套加减乘除运算系列等等。

赋值、下标、调用、成员访问->这几个运算符应该是必须是成员函数，如果不是他就会报错说你这不是非静态成员函数。大概因为符号左侧必须要是类的对象。有对称性的符号应该是外部函数，虽然我们也可以把它定义成成员函数，但是左侧对象必须是那个类的对象。>>和<<也不能是成员函数，因为他们的第一个操作数不可能是那个类。其余基本上都是成员函数了。

下标运算符要重载的话就要有两个版本：返回一般引用，常量引用

举递增运算符为例:

++、--的操作要注意是否超过有效范围，所以要进行检查。加减乘除所带来的改变。

CLASS& operator++ ()//前置++

CLASS& operator++ (int)//后面++，int这个形参不会被命名。如果显示调用的话，那么只用在()里面加入一个int，根据重载规则，那么就会选中它。只能是int。

箭头运算符的重载：

X& operator->() { return \*this; } //用起来真的不方便

重载函数调用操作符：

struct absInt {

int operator()(int val) const {

return val < 0 ? -val : val;

}

}absobj;

int a =absobj(-42);//这些运算符都是给对象使用的。而本例中，他们就像函数一样，是可调用对象。

大概是基于函数重载，库在算法，关系比较、位运算上有一些可函数调用的类：

Arithmetic（Algorithm）：plus<Type> minux<...> multiplies<...> divides<...> modulus<...> negate<...>

Realtional:equal\_to<...> not\_euqal\_to<...> greater<...> greater\_equal<...> less<...> less\_equal<...>

logic:logical\_and<...> logical\_or<...> logical\_not<...>

使用：

sort(svec.begin(), svec.end(), greater<string>());//第三个参数里面没有传参

类类型转换class-type conversions：

又称为user-defined conversions。其实就是重载了类型转换运算符。

operator int(） const;//写法格式，由于是对类的操控的重载运算符，必须是成员函数。话说我定义了这个类类型转换函数就可以和int进行愉快的加法了（利用隐式转换），比写operator+方便。不过单单靠类类型转换是会出错的，istream定义了bool的类类型转换但是没有定义<<（假设istream实现地如此粗略），所以cin<< 42;这表达式就是一个对常量的左移位操作。为了防止这种情况的发生，可以使用C++11的expcilit conversion operator，这样以后必须显式操作才能实现。显式的类型转换就是那些cast，新老版本的(type)、type()。不过这个expcilit是有个方便的规定例外的，即使如果被对象用在了使用条件的地方，不需要显式指出就会转成bool。

如果一个类中有多个类类型转换，我们要保证向一个类型转化只存在一种可能。否则我们的代码就会具有二义性：一.A类定义了B类转成A的转换构造函数，B定义了一个转A的类类型转换函数；这时候要避免就要去显式使用定义的类型转化，A(b)表示使用A中定义的那个(选中这个我猜测是因为从左到右进行读取)，要想使用B中定义的那个就要b.operator A()二.定义多个转换规则，转换出来的类型之间有联系，比如说一个int()一个double()甚至是long()。其实一和二碰到了也没有关系，只要能做到没有二义性就可以。二义性引发的是编译时函数重载错误。所以非要用时解决办法就是解决重载匹配，long对应int与double这样子就会ambiguous，都要经过类型转换被认为是一样好，但是先对long进行int或者double类型的转换，那么编译器就知道在编译时选择哪个函数了。当我们对同一个类提供了转换目标是算术类型的类型转换，也提供了重载的运算符，那么就会遇到重载运算符与内置运算符的二义性问题。

内存分配失败并不会中断程序（除了make\_shared的抛出异常），但是如果你基于此继续操作的话就会步入陷阱。

351.函数调用使用map打包：

map<string, function<int(int, int)>> binops = {//前面这个string就像是一个用于寻找的key

{"+", add}, // function pointer

{"-", std::minus<int>()}, // library function object

{"/", div()}, // user-defined function object

{"\*", [](int i, int j) { return i \* j; }}, // unnamed lambda

{"%", mod} }; //一般函数

cout <<　binops["+"](10, 5);

　　一种C++11的库类型function，是模板，使用就像是函数指针，定义在functional头文件：

function<int(int, int)> f; function<int(int, int)> f(nullptr);

function<int(int, int)> f(obj);

std::function<int(int, int)> f( [](int a, int b) {return a+b;} );//可以=赋值，就把它当作一个类型来看待。匿名Lamndba倒像是一个量可以赋给别人。

function与unary\_function、binary\_function没有关联，后面两个已经被bind代替了

插入：map.insert(pair<type1,type2>(xx,xx))

如果通过[] 运算符来c插入数值，区别是当map中已经存在一个同key元素会更新值

map<int, int> iMap;

std::cout << iMap[42] << std::endl; 一旦使用了一个原本不存在元素，就会自动创建一个经过初始化的元素，所以输出0 。

352.OOP（object-oriented programming）：

核心思想是封装、继承、多态（动态多态是虚函数，静态是函数重载）。动态绑定是忽略相似类型的差距，而以统一的方式去使用它们的对象。

继承Inheritance：

class 派生类名 : 继承方式 基类名1，继承方式 基类名2， ……

{

...；

} // public表示继承不变的访问控制，private表示继承的全都变成private。如果不用访问修饰符，那么struct默认public继承，class默认private继承。

友元关系不能继承

基类通常要定义vitural destructor。即使没有用到，书上说也还是要该定义的。否则，delete一个指向派生类对象的基类指针将产生未定义的行为。虚析构函数将会阻止synthesis版本的move系列操作的生成，于基类和派生类中。因为不然的话，由于动态类型与静态类型的不对齐会导致，一个派生类的动态类型但是其中的基类部分被move告诉编译器我们只会对其进行赋值与析构，那么不就废了？！析构函数的动态重载是特殊的。基类要有移动操作，派生类才可能有移动操作。

如果没有设置vitural function，是不可以被覆盖的。override。要覆盖的函数通常定义为虚函数。当我们使用指针或者引用调用虚函数的时候，该调用将被动态绑定，函数的解析过程发生在运行时期，否则解析过程发生在编译时期。

继承过来的虚函数还是虚函数，只有纯虚函数不能定义。

父类的构造函数会比子类的先进入，并不是进入子类的构造函数然后再计入父类的构造哈数

final的位置在类名后或者函数名后，inline、constexpr要用就要恰好放在返回值类型前面这个位置。

虚表是每个类一个，而不是每个对象一个。对虚表中内容的改变需要借助指针或者引用。虚表的内容替换？如果又有正常使用，又有多态使用怎么办？但是析构函数作为虚函数的时候，基类与子类名字不同？补充调用p->mem():的流程？

虚函数不会是静态的，因为静态成员函数独立于对象存在,没有this指针。

虚函数也不能是内联函数，因为内联函数是在编译时确定的。也就是说声明了virtual就不会自动再声明inline。

虚函数不能是构造函数，因为虚函数是需要虚表的，但是虚表却要构造函数先分配好内存空间。

今天写虚函数发现基类中的虚函数必须被定义，也就是必须要有{}。否则报的错就是 undefined reference to `vtable for 类名 `。不定义的话，只要不使用（比如说派生类中不去覆盖），那么就可以编译通过(这与一个类的定义它的编译顺序有关)。因为函数原型怎么被virtual 覆盖！如果给一个纯虚函数在内部定义的话就会说pure-specifier on function-definition。

调用p->mem():

首先确定p的静态类型

在静态类型所对应的类里面查找mem，找不到就在直接基类中找。最终找不到，报错

一旦找到了mem，就进行常规的类型检查

如果调用合法，如果不是虚函数的就使用编译时代码。

352.5. 泛型编程是在编译时获知类型。

359.模板与泛型编程

模板函数：作用域仅限于这个函数。模板的作用域，不能用block进行扩大（编译器解析的时候会出现问题），虽然模板类似乎是这么做的

template <typename T, typename F = less<T>>//F这个类型的声明和using相似

int compare(const T &v1, const T &v2, F f = F())//=F（）是默认初始化

{

if (f(v1, v2)) return -1;

if (f(v2, v1)) return 1;

return 0;

}

从C++的观点来看，模板参数列表中的class与typename是一样的。大家更喜欢用typename因为class看起来就像是class类型。不过在有几个场合中typename不同于class。

模板的实例化要在使用时才可以确定，而多个文件中实例化相同模板的开销非常严重。所以C++11让我们可以显式实例化：那你这样无非就是把写好的模板代码一贴，然后把类型给搜索替换掉，从而达到可读性上的一致，又减少开支吗。

extern template declaration;

比如extern template class Blob<string>; //这还只是声明，要去掉extern来个定义

template declaration;

比如template int compare(const int&, const int&);//这直接是声明了，这个倒没有必要使用模板函数。

模板的实现原理？居然是运行时识别类型？

template <typename T1, typename T2, typename T3>

T1 sum(T2, T3);

auto val3 = sum<long long>(i, lng);//返回值T1就是long long，T2、T3是要推断出来的

template <typename T1, typename T2, typename T3>

T3 sum(T1 a, T2 2) {

return a+b;

}

sum<float>(1,2) float是对T1的（因为T1是第一个）.（1，2）是对T1、T2的，所以T3并没有得到类型。

sum<int, int, float>(1, 2)这样T3才得到float类型

我们希望编写一个函数，接受表示序列的一对迭代器和返回序列中的一个引用：

template <typename It>

decltype(\*beg) fcn(It beg, It end)

{

return \*beg;

}//由于编译器到达参数列表之前，beg都是不存在的，所以应该换一种形式将返回值的类型挪到beg类型后

template <typename It>

auto fcn(It beg, It end) -> decltype(\*beg)

{

return \*beg;

}

类模板的派生：

template <class Type>

class TLinkList

{

//基类类模板定义

};

template<class Type>//由此看来模板的作用域的确是只有一block

class TOrderedLinkList: public TLinkList<Type>

{

//派生的类模板定义

};

364.对象数组的每个元素一般调用无参构造函数或所有参数都带有默认值的构造函数：

CPoint \*p = new CPoint[6];

// 只能调用无参构造函数或所有参数都带有默认值的构造函数

CPoint a[] = {CPoint(1, 1), CPoint(2, 2), CPoint(3, 3)};

// 调用带参构造函数，不使用参数默认值，这是惟一例外情况。也可以使用对象的复制或移动嘛

370.继承中的二义性：

class Bed

{

public:

Bed(float); // 基类构造函数

void SetWeight(float); // 设置重量

float GetWeight() const; // 访问重量

void Sleep(); // 睡觉

protected:

float weight; // 重量

};

class Sofa

{

public:

Sofa(float); // 基类构造函数

void SetWeight(float); // 设置重量

float GetWeight() const; // 访问重量

void WatchTV() ; // 看电视

protected:

float weight; // 重量

};

class SofaBed : public Sofa, public Bed // 多继承

{

// sb.SetWeight(58.2); // 有歧义

sb.Sofa::SetWeight(58.2); // 调用Sofa的同名函数

sb.Bed::SetWeight(58.2); // 调用Bed的同名函数

//也是用直接指定来消除歧义，不过可以使用虚基类去解决问题。

371.使用虚基类的方法，使得在继承间接共同基类时只保留一份成员副本，这样将不会出现多义性问题（不同的类名就是不同的类型，所以继承自统一基类的两个类的某个函数就会被认为是不同的函数，而使用了虚继承以后就知道函数来自的是那个基类）。格式：class 派生类名: virtual 继承方式 基类名

class Furniture // 声明公共基类

{

public:

Furniture (float w): weight(w){}

void SetWeight(float w) { weight=w;} //设置重量

float GetWeight() const { return weight; } //得到重量

protected:

float weight; //重量

};

class Bed: virtual public Furniture //将公共基类声明为虚基类

{

public:

Bed(float w): Furniture(w) {} //初始化公共基类的数据成员

void Sleep() { cout<<"开始睡觉"<<endl; } //睡觉

};

class Sofa: virtual public Furniture //将公共基类声明为虚基类

{

public:

Sofa(float w): Furniture(w) {} //初始化公共基类的数据成员

void WatchTV() { cout<<"正在看电视"<<endl; }//看电视

};

class SofaBed : public Sofa, public Bed

{

public:

SofaBed(float w): Furniture(w), Sofa(w), Bed(w) {} //对公共基类与中间基类的初始化

void FoldOut() { cout<<"可以折叠和打开"<<endl; } //折叠

};

虚基类的初始化在语法上和一般基类一样，只是调用顺序略有不同，它遵循以下顺序：

虚基类的构造函数在非虚基类之前调用

若同一层次中包含多个虚基类，则按它们在派生类中声明虚基类时被指定的顺序调用

若虚基类由非虚基类派生而来，要先调用更高级别基类的构造函数，再遵循上述1和2的顺序

例如：

class A;

class B;

class C : public A, virtual B

{

};

将产生如下的调用次序：

B( )

A( )

C( )

例如有基类B，从B派生出C和D，然后类F又同时继承了C和D，现在类F的一个对象里面包含了两个基类B的对象，如果F访问自己的从基类B那里继承过来的的数据成员或者函数成员那么编译器就不知道你指的到底是从C那里继承过来的B对象呢还是从D那里继承过来的B对象。于是虚基类诞生了，将C和D的继承方式改为虚继承，那么F访问自己从B那里继承过来的成员就不会有二义性问题了，也就是将F对象里的B对象统一为一个，只有一个基类B对象，下面是一段代码说明了对虚基类的使用。

379.流的补充：

fstrema是由ostream派生而来，并不是由ifstream与ofstream合并而来。

输出单个字符可以使用put函数。put(ch)就会把ch表示的字符输出到标准输出文件。

输入单个字符可以使用get函数，get(ch)就会从标准输入文件中读取一个字符给ch。遇到文件结尾符，就返回EOF。

putback()就是把一个字符返回到输入流当中。

观测输入流对象中的当前字符peek()。输入流对象.peek()。不过我觉的它的原理就是取出来了，看一下，然后又放回原流去了。

文件流对象.read(字符指针buffer, 长度len);

文件流对象.write(字符指针buffer, 长度len);//可以一次将一个数组写入文件

文件从用户的角度上看可以分为设备文件与普通文件。

tellg() 返回输入文件指针的当前位置

seekg(位置) 将输入文件中指针移到指定的位置

seekg(位移量, 参照位置) 以参照位置为基础移动若干个字节

tellp() 返回输出文件指针当前的位置

seekp(位置) 将输出文件中指针移到指定的位置

seekp(位移量, 参照位置) 以参照位置为基础移动若干个字节

函数名的最后一个字母g用于输入文件的函数（g是get的第一个字母），p的是用于输出文件的函数（p是put的第一个字母），如果是既可输入又可输出的文件，则任意用tellg()与tellp()，seekg()与seekp()都是等价的。

ios::beg 文件开头,这是默认值。

ios::cur 当前的位置。

ios::end 文件末尾。

382.命名空间用于减少同一个名字的冲突，用特殊的域去限制；区分不同人员写的代码

命名空间需要先定义，后使用，命名空间一般定义格式如下：

namespace 命名空间名//如果没有命名空间的名字，那么就会当作全局来使用

{

//各种名字的声明或定义，甚至可以是函数。

}

它的定义方法和一个类，它不可以使用访问控制，因为它本身就是要求外部可以访问的，不必做的就是不要做的。

404.下面这两个符号重载函数已经可以让+运算完整了：

COperator operator=(int a);

COperator operator+(COperator ob);

416.VC6 Debug模式下默认对局部变量初始化0xcccccccc，当遇到初始化语句中的赋值，他才变成那个值。VC6.0未进行复制的堆空间值为0xCDCDCDCD，也就是说野指针的值会是这个

在变量内存快块的前后会加上4个FD。

释放的堆空间会被赋值为EE FE。

以上对堆的管理方法都是在Debug模式下。

这些赋值是判断问题的好技巧

417.使用new申请的堆空间是通过双向链表进行链式管理，最后一个节点的0x0000000表示链表的结尾。

419.math库的pow函数不要传给它int参数，我把一个int结果的表达式（表达式内全是int的加减运算）传过去结果居然出现偏差。但是换成int常量直接代替表达式就又计算正常了。mdzz，反正这种要隐式转换的即便是int转double都要小心。否则可能会引起计算偏差。 问题已经无法复现？！

424.

#include <iostream>

int main(void)

{

const int i =0;

int \*j =(int \*)&i;

\*j =1;

std::cout << i << \*j << std::endl;

}

这一段代码，在gcc下编译就是11，在g++下编译就是01。之所以会这样是因为那时候编译器看到const语句认为是常量，之后再看到i就一直用0替换，跟#define比较类似。

491.左值与右值？

a=b+25;//a是左值

b+25=a;//此时a是右值

494.使用下标和指针：指针的效率至少和下标一样，如果正确使用的话。

两者的算法都要使用到乘法运算距离，再然后加法。不过区别再具体实现上:

int array[10],a, \*ap;

for(a =0; a< 10; a+=1)

array[a] =0;

for(ap=array;ap < array+10;++ap)

\*ap =0;

使用指针的循环每一步的计算所用的一切都是一样的，所以在可以优化成为第一次使用了乘法之后就保留了乘法得到的那个结果，于是以后就用加法了。

DP思想：

for(i =0;i <size; ++i)

x[i] =y[i];//这里的下标运算符是消耗时间的主力

for(p1 =x,p2 =y;p1-x < size;)

\*p1++=\*p2++;//下标运算符得到了改进，但是循环的判断部分变得糟糕

for(i =0,p1 =x, p2 =y;i < size;i++)//判断部分得到了改善，如果将i、p1、p2声明为寄存器变量的话，那么没准可以再少掉一些数值复制的步骤。

\*p1++ =\*p2++;

forL(p1 =x,p2 =y;p1 <&x[size];)//少掉了计数器i的运算，x[size]其实可以早早就算好。p1、p2是寄存器变量。

\*p1++ =\*p2++;

495.现在许多编译器比程序员更加懂得怎么样分配寄存器，对于这类编译器，使用register反而会降低了效率，请检查编译器的文档看看他是否会执行自己的寄存器分配策略。

496.多维数组中只要第一维才可以缺省。其实其他维度编译器也是可以推断出来的。编译器需要的只是其下层每个单元的大小。所以int a[3][3][][3]也是没有问题的！但是他就是不这么做，估计是嫌编译器的编写者要编写更多的代码吧。

498.

可能某个文件需要include的头文件，在另一个子文件夹里面。这个时候就需要在include的时候，使用..\(上级目录)Utilities\CMsgStation.h、.\(当前目录)DirectShow\Device\CVideoDevices.h等等。这样比较麻烦，有一个解决方法就是添加工程的包含（include）目录，比如设置为：$(ProjectDir)XXX，其中$(ProjectDir)是vc编译器的预定义宏，代表当前工程目录。这样include XXX目录下的头文件yyy.h时，就可以直接include yyy.h了。

筛选器就是编程时右上角的那个解决方案资源管理器。

501.不过在VS下的使用winsock2.h以及WS2tcpip.h都是会报错error LNK2019：无法解析的外部符号...。网上说是库没有被链接进去。我试了下：#gragma comment(lib, "ws2\_32")但是没什么软用。最后在项目-》MyPing属性-》链接器-》输入-》附加依赖项，在其中加入了ws2\_32.lib才成功了。

506.singal.h在C++里面自然就是csignal。两个函数：signal() raise()

signal()函数用于设置程序信号的处理机制：到有点像异常

static int run = 1;

void handle\_signal(int s)

{

run = 0;

}

signal(SIGINT, handle\_signal);

signal(SIGTERM, handle\_signal);

SIG\_DFL,SIG\_IGN 分别表示无返回值的函数指针，是头文件中定义的handle\_signal，指针值分别是0和1，这两个指针值逻辑上讲是实际程序中不可能出现的函数地址值。  
SIG\_DFL：默认信号处理程序  
SIG\_IGN：忽略信号的处理程序

/\*

signal()函数中的消息。

SIGABRT

异常终止，如调用abort()。

SIGFPE

算术运算出错，如除数为0或溢出。

SIGILL

非法函数映象，如非法指令。

SIGINT

交互式信号，如中断。windows中C-C结束程序

SIGSEGV

非法访问存储器，如访问不存在的内存单元。

SIGTERM

发送给本程序的终止请求信号。

SIGCLD

当子进程退出时，会给父进程发送一个SIGCLD信号

\*/

signal()用于确定以后当信号sig出现时的处理方法。如果handler的值是SIG\_DFL，那么就采用实现定义的缺省行为；如果handler的值是SIG\_IGN，那么就忽略该信号；否则，调用handler所指向的函数(参数为信号类型)。

int raise(int sig);向程序发送信号sig。如果发送不成功，就返回一个非0值。

当一个已经设置了信号处理函数的信号发生时，为了防止出现信号处理函数的无限递归问题，如果信号时异步的也就是不是调用abort和raise引起的，信号处理函数就不应该调用除了signal以外的所有库函数，因为在这种情况下结果是未定义的。

507.5

**\_exit()**函数的作用最为简单：直接使进程停止运行，清除其使用的内存空间，并销毁其在内核中的各种数据结构；exit() 函数则在这些基础上作了一些包装，在执行退出之前加了若干道工序，也是因为这个原因，有些人认为exit已经不能算是纯粹的系统调用。  
  
**exit()**函数与\_exit()函数最大的区别就在于exit()函数在调用exit系统调用之前要检查文件的打开情况，把文件缓冲区中的内容写回文件，就是"清理I/O缓冲"。 但是不能释放僵尸进程的数据结构。

508.fwrite和fread函数的表大小参数被拆城了第二个表大小的参数与第三个表数量的参数，因为是阻塞函数，且不提供非阻塞。返回值是块的读写次数，不是字节数。

509.EOF详解：

feof()最大的用处就就是防止while((c=fgetc(fp)) != EOF)把0xFFFFFFFF这样的不是文件结尾的数据却因为EOF被宏定义为-1误判。

#define EOF (-1)

#define \_IOEOF 0x0010

#define feof(\_stream) ((\_stream)->\_flag & \_IOEOF)

当文件位置指针(fp－>\_ptr)到了文件末尾，然后再发生读/写操作时，标志位(fp->\_flag)才会被置为含有\_IOEOF。再调用函数就会返回EOF了。

26号ASC码，被称为Substitute character（代替符，置换符）。表示无效的，错误的，或者不能在设备上表示的。在ASCII以及UNicode字符集中都为26，在键盘上的对应按键就是Ctrl+Z。这个其实是一种softEOF，不是真正上的文件结尾，有时用它来影藏后面的数据，因为文本模式或控制台下程序读到这里就停止了。据说有些文件格式在头部中加入这个字符就是这个作用，例如PNG、GIF。有些文件格式以推荐在末尾插入这个字符（但是绝不是在编辑文本时候按Ctrl+Z,文本识别成Undo操作）。不过在二进制模式下or using the File System Object to progress beyond it，这个字符文件尾的作用就失效了。

相似的:Number 4 is called End-of-Transmission and is used by Unix to mark or end of input similar to no 26 in DOS. Number 28 which is called "File Separator" has also been used for similar purposes.

glibc 的 getchar 定义是:

int

getchar(void)

{

int result;

\_IO\_acquire\_lock (\_IO\_stdin);

result = \_IO\_getc\_unlocked (\_IO\_stdin);

\_IO\_release\_lock (\_IO\_stdin);

return result;

}

\_

IO\_getc\_unlocked 是个宏，定义是：

#define \_IO\_getc\_unlocked(\_fp) \

(\_IO\_BE ((\_fp)->\_IO\_read\_ptr >= (\_fp)->\_IO\_read\_end, 0) \

? \_\_uflow (\_fp) : \*(unsigned char \*) (\_fp)->\_IO\_read\_ptr++)

也就是说：在正常状态下，返回的永远都是 unsigned char 范围内的 int，否则返回一个「下溢出」值，一般就是 EOF（-1）。

511.全局变量只能被constant初始化么，因为它的建立太早了，初始化的话只能用constant，不能是函数。

515.strlen返回的是size\_t这在stddef.h中定义。

516.标准库有些时候使用汇编语言来实现，目的就是为了充分使用默写及其所提供的特殊的字符串操控指令。从而追求最大限度的速度。

519.补充string操作函数：

strspn(str, char \*group)它将返回str所指向的字符串起始部分中与group中字符匹配的数目。其相反的函数就是strcspn，就是数group中没有的字符。

字符串按照指定的字符分隔：最新的Linux内核2.6.29，说明了strtok()已经不再使用，由速度更快的strsep()代替

520.C语言中内存操作函数：

已知的有memcpy，memcmp，memmove，memset函数。

void \*memchr(void const \*a, int ch, size\_t length);就是加了第三个参数用于限制搜索长度的strchr函数。

526.什么时候你应该向函数传递一个结构而不是结构体指针？只有当结构体比较小的时候。传输效率就是与字节数有关的。

527.位段成员必须声明int、signed int或者unsigned int类型。

struct BITSEG {

signed int ch: 7;

signed int ch2: 6;

signed int ch3: 19;//我记得之前似乎有似乎有不能跨字节的要求，现在我只知道不要超过系统的一个字，否则某些编译器报错。

};

sizeof的输出都是以字节为单位的，所以它测不出来位段的大小？不，在我的SSD里面它是4字节对齐的。一般都是4字节对齐，而且可以在代码里面进行设置。位段涉及很多不确定因素，位段不跨平台，可移植性差。

529.联合的初始化：

union {

int a;

folat b;

}x={ 5 };我们只能按照联合的第一个成员进行初始化，不能对浮点进行初始化。

531.scanf获取字符串的时候别忘了你的字符缓冲区是有大小的，所以%ns。

532.int f()[]是一个函数返回一个数组，但是声明非法，因为不能返回数组。

int f[]()是一个函数数组，但是数组各个元素大小要一样，不同函数却大小不同。所以非法。

int (\*f[])()借助指针来统一数组各个元素的大小。（函数指针数组）

534.字符串是可以加上整数的：

"xyz"+1 尼玛，别忘记字符创常量能转换成地址，所以如果打印输出是yz。

536.#if defined(xxx就是#ifdef(xxx

#line number "string" number修改了\_\_LINE\_\_, string修改了\_\_FILE\_\_

540.scanf限定符：

对于浮点数，有l来修饰表示是double，L来修饰表示是long double。

541.printf的格式标志：

- 一般是左对齐，这个用于要求右对齐· +

0 使用右对齐的时候缺省情况下是使用空格来表示左边为使用的列。使用了这个就是用零来填充。所以很明显要与长度限制共同使用才会有效果

+ 当格式化有符号值的时候，如果值非负就多输出一个+号在前面。

空格 当格式化有符号值的时候，如果值非负就多输出一个空格在前面，如果和+同时出现，这个就会无效。

# 后面跟着格式。用于o，x，X，e，E，f，g，G这几个前面。对于o，x，X就是进行对应进制格式的输出。对于e，E，f浮点数就是保证有一个小数。对于g，G除了有浮点数的作用还额外保证缀尾的0不会被自动去除。

%l64d g++支持不好，对于long long应该使用%lld。

printf("\32\40");中间的是按照8进制进行处理而不是十进制。因为就是八进制把最前面的0给省略掉。

你要读取wchar\_t（C语言没有char16\_t那是C++11的）。要用wscanf()，还是%c就好。

542.两个缓冲区设置函数：

将缓冲区设定为一个自动数组会不好的，因为当数组被撤销以后流还会继续使用这个地址。

void setbuf(FILE \*stream, char \*buf);

如果缓冲区是NULL，就会关闭所有的缓冲区。

int setvbuf(FILE \*stream, char \*buf, int mode, size\_t size);size修饰buff，如果buff是NULL，那么size必须是0。mode有三种：\_IOFBF，\_IONBF，\_IOLBF。

543.对流的操作：

int ferror(FILE \*stream)报告流的错误状态

void clearerr(FILE \*stream)函数对指定流的错误标志进行重置

544.临时文件：

FILE \*tmpfile( void );建立临时文件来临时保存数据

char \*tmpnam( char \*name );更改建立的临时文件名字，这样子你可以去尝试抹去句柄，然后可以对文件进行复制保存一份等等。

545.文件操作函数：

int remove(char const \*filename);

int rename( char const\*oldname, char const \*newname);

546.补充几个math操作函数：

int abs(itn value);

logn labs(long int value);

div\_t div(int numerator, int denominator);

ldv\_t div(long int number, long int denom);

int quot;//商

int rem;//余数

三角函数：

double sin(double angle)

double cos(double angle)

double tan(double angle)

double asin(double value)

double acos(double value)

double atan(double value)

double atan2(double x, double y)类比div这些函数明显是第二个参数除以第一个参数

双曲函数：

double sinh( double x );

double cosh( double x );

double tanh( double x );

指数的相关计算函数：

fraction\*2^exp =value

double frexp( double value, int \*exponent );//求fraction

double ldexp( double fraction, int \*exponent );//求value

上面这两个函数用于创建于机器无关的浮点数。

floor，ceil，fabs，fmod

fmod自然是double fmod( double x,double y )

549.对于字符串转数字函数，一旦超出方位就会在error中存储ERANGE这个值。对于函数，如果值太大就会返回HUGE\_VAL，如果值太小就会返回0。

550.time.h

如果涉及时间统计最好是使用系统提供的API，一般提供可以精确到微秒，但是确实是存在到纳秒的层次。但是直接计算时间还不如统计时钟周期来的准确，但是即便是使用C语言统计时钟周期，还是不是很准确，如果一定要这种东西，建议配置一个良好的环境，多次运行。纯DOS，X86实模式下或许才能得到较准确的指令周期数。

clock\_t clock(void);返回程序从开始执行起处理器所消耗的处理器周期，要转化为秒就要除以CLOCKS\_PER\_SEC。注意这个值可能是近似值。你可以从main函数刚开始执行的时候调用这个函数，然后以后调用就减一下。如果机器无法提供处理器时间，或者如果时间值太大，就无法用clock\_t表示，函数就会返回-1。

time\_t time( time\_t \*returned\_value );如果参数是非NULL指针，时间值就会通过当前指针进行存储。如果机器无法提供时间或者时间太大就会返回-1。time\_t是32位的话就会在2038年溢出。标准并没要求这个函数的返回值用秒来表示，所以使用这个函数有风险。解决办法是difftime()，这个函数会计算参数1与参数2的差，并将时间转化为秒。

char \*ctime( time\_t const time\_value);输入time\_t时间，返回时间的字符串表示形式。

struct tm \*gtime( time\_t const \*time\_value);

struct tm \*localtime( time const \*time\_value);

都是把time\_t转为tm结构体，前者是返回格林尼治时间，后者是当地时间。

tm结构体中的成员：都是int

tm\_sec,tm\_min,tm\_hour,tm\_mday,tm\_mon,tm\_year,tm\_wday（星期天后的天数）,tm\_yday（1月1日后的天数）

据说ANSI委员会考虑的很周详，有闰秒机制从而适应地球旋转的细微变慢现象。

551.非本地跳转：<setjmp.h>

int setjmp( jump\_buf state );

void longjmp( jump\_buf statee, int value);

提供一种类似于goto的机制，但是不局限于一个函数的作用域。你声明一个jmp\_buf变量，并用setjmp的返回值为零。setjmp把程序当前的状态信息保存到跳转缓冲区。以后在任何地方使用longjmp函数就会进行状态的恢复，longjmp函数会使执行在setjmp中流通。于是他自己的返回值也就要靠第二次setjmp来返回了。

set(restart);//记录位置

switch( set(restart) ) {//也可以通过错误号来查看

case 0://正常

case 1://小错误

default://致命错误

}

553.环境：stdlib.h

环境就是一个由编译器定义的键值对列表，它由操作系统维护。使用getenv这个函数在列表中查找一个特定的名字，如果找到，返回一个指向其对应值的指针。

BUFSIZ定义在stdlib.h中，大小为8192。

system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string字符串所代表的命令，此命令执行完后随即返回原调用的进程。在调用system()期间SIGCHLD 信号会被暂时搁置，SIGINT和SIGQUIT 信号则会被忽略。如果给system一个无效命令会返回0；如果system()在调用/bin/sh时失败则返回127，其他失败原因返回-1。若参数string为空指针(NULL)，则返回非零值。如果system()调用成功则最后会返回执行shell命令后的返回值，但是此返回值也有可能为system()调用/bin/sh失败所返回的127，因此最好能再检查errno 来确认执行成功。  
   
附加说明  在编写具有SUID/SGID（Super user/group identity）权限的程序时请勿使用system()，system()会继承环境变量，通过环境变量可能会造成系统安全的问题。

555.排序与查找：

void qsort( void \*base, szie\_t n\_lements, size\_t el\_size, int (\*compare)(void const \*, void cosnt \*) ); 快排函数

void bsearch(void const \*key, void const \*base, size\_t n\_elements, size\_t el\_size, int (\*compare)(void const \*, void const\*)); 二分查找函数

qsort只是实现了快速排序，而STL库中的sort实现了快排，堆排，归并排等多种排序，并且针对数据量的不同做了诸多优化，如果最终效率不及qsort。在debug模式下，qsort快了很多；release模式下，sort更加快。

557.有些时候while循环比for好看而且还少了一个变量。

对于传递指向数组节点的指针

while(len > 0) {

...

len--;

}

560.malloc一次性最大能申请多大内存空间？理论上空间上限受用户态内存地址空间的限制。64 位系统下分配几个 T 不成问题。我分配内存是可以达到1.7GB，这个似乎刚好与我内存的剩余可用大小一样windows 10中，分配2G内存就会导致函数返回NULL，他们却普遍说windows 32位是可以分配2G内存给用户，实际可以使用有1.9G。malloc的机制是：分配时先从进程线性地址空间分配一块线性地址返回给用户进程，直到你读写这块地址才触发真正的缺页请求，从内存申请真正的内存块给用户。也就是说你请求了比如100M，但你只用过一次这块内存50k，那么实际上内核真正给你用的可能很就是128k。

但是真正快速的内存分配策略是需要自己实现，不能使用这种API。注意内存中字节对齐问题。

562.直接在程序中使用printf而没有头文件，使用gcc的话就会给一个Warning，G++是给你一个error。gcc给了一个Warning之后就会自己去链接（申明一个函数原型可以消除warning），所以程序还是可以正常执行的。

566.int a[] ={1, 2, 3, 3,};这个数组不会因为最后面那个逗号而多出一个0。总共四个元素。

576.C语言的main函数被启动函数调用。main函数的返回值会给一个叫mainret的int型变量，所以main的返回值应该是int。最后传给系统，作为程序的返回状态码。

577. gets\_s()，这是微软自己的东西，不具备移值性。后面带的那个\_s是安全的意思，号称是更安全的函数。

578.linux中g++一个.h文件会变成xxx.h.gch。有一次gcc傻逼了，编译会报错，但是g++成功，于是就用g++来生成中间文件。

580.数组按照列存储还是行存储与编程语言有关。但是考虑到读取的时候会缓存附近的内容，一般为了性能考虑应该使用按内存存储来读取。

585.gcc编译cpp文件与c文件，会得到不同的错误警告。gcc也是可以编译cpp文件的。

586.在C语言里可以使用微软提供的appid来打开edge这类。

587. 交叉编译ARM平台程序，把gcc换成arm-linux-gcc

588. 栈，但是根据调用协定，可能参数使从左往右压入的，也有可能是从右往左。\_\_cdecl\\_\_stdcall（从右往左压入栈）和\_\_pascal不一样

\_\_fastcall快速调用约定（calling convention)规定将前两个参数由[寄存器](https://baike.baidu.com/item/%E5%AF%84%E5%AD%98%E5%99%A8)ecx和edx来传递，其余参数还是通过[堆栈](https://baike.baidu.com/item/%E5%A0%86%E6%A0%88)传递（从右到左）。不过不同编译器规定的寄存器不同。所以跨编译器的时候需要注意。

589.发生段错误的时候，操作系统会向程序发送SIGSEGV消息，然后程序停止。不过在调试状态下，程序不会被关闭，可以拦住以方便明白那里错了

597.使用内存文件由映射共享文件来共享内存

//输入参数不能把.exe省略了

#include <stdio.h>

#include <string.h>

#include <windows.h>

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

{

if (argv[1] == 0) { //如果是作为主进程运行

HANDLE hShmem;

char param[1024];

STARTUPINFOA si;

PROCESS\_INFORMATION pi;

char\* pstr;

hShmem = CreateFileMapping(INVALID\_HANDLE\_VALUE, 0, PAGE\_READWRITE, 0, 256, 0); //创建一个新的文件映射内核对象共享内存。

SetHandleInformation(hShmem, HANDLE\_FLAG\_INHERIT, HANDLE\_FLAG\_INHERIT); //使得句柄可以继承到子进程，用于映射

//子进程映射共享缓冲区以后写入"Hello~!"

sprintf(param, "\"%s\" %x", argv[0], hShmem);

ZeroMemory(&si, sizeof(si));

si.cb = sizeof(si);

if( CreateProcessA(argv[0], param, 0, 0, TRUE, 0, 0, 0, &si, &pi) ==0 ) {

puts("create process error");

return 1;

}

WaitForSingleObject(pi.hProcess, INFINITE); //等待子进程运行结束

//父进程读出并

pstr = (char\*) MapViewOfFile(hShmem, FILE\_MAP\_READ, 0, 0, 0); //将共享内存对象中的内存块映射到当前进程

puts(pstr);

UnmapViewOfFile(pstr);

CloseHandle(hShmem);

return 0;

} else { //如果是作为子进程运行

Sleep(1000);//对应linux中的usleep()

HANDLE hShmem;

char\* pstr;

sscanf(argv[1], "%x", &hShmem);

pstr = (char\*) MapViewOfFile(hShmem, FILE\_MAP\_WRITE, 0, 0, 0); //将共享内存对象中的对象映射到当前进程

strcpy(pstr, "Hello~!"); //往共享内存里写入字符串

UnmapViewOfFile(hShmem);

CloseHandle(hShmem);

return 0;

}

}

/\*

MapViewOfFile用于将存储于磁盘的文件放进一个进程的虚拟地址空间，并在该进程的虚拟地址空间中产生一个用于"存放"该文件的区域，这个空间就叫做File View。系统同时产生一个存放在物理内存中的File Mapping Object用于维护这种映射关系。这时当多个进程需要读写那个文件数据时，它们的File View其实对用的都是同一个File Mapping Object。这样做可以节省内存和保持数据的同步性，并达到数据共享的目的。

FILE\_MAP\_WRITE

读和写的访问权限。hFileMappingObject参数创建时，有PAGE\_READWRITE保护属性。

FILE\_MAP\_READ

写的访问权限。hFileMappingObject参数创建时，有PAGE\_READWRITE或者PAGE\_READONLY保护属性。

FILE\_MAP\_ALL\_ACCESS

和FILE\_MAP\_WRITE一样

FILE\_MAP\_COPY

写复制访问权限。如果创建映射文件对象时，有PAGE\_WRITECOPY和此参数（FILE\_MAP\_COPY），会接收执行文件的视图。

\*/

600.C语言判断文件或文件夹存在：

方法一：access函数判断文件夹或者文件是否存在

函数原型： int access(const char \*filename, int mode);

1. 所属头文件：io.h （linux是#include <unistd.h>  ）

filename：可以填写文件夹路径或者文件路径

mode：0 （F\_OK） 只判断是否存在

           2 （R\_OK） 判断写入权限

           4 （W\_OK） 判断读取权限

           6 （X\_OK） 判断执行权限

用于判断文件夹是否存在的时候，mode取0，判断文件是否存在的时候，mode可以取0、2、4、6。 若存在且具有权限，返回值为0；不存在或者无权限，返回值为-1。

错误代码

EACCESS 参数pathname 所指定的文件不符合所要求[**测试**](http://lib.csdn.net/base/softwaretest)的权限。

EROFS 欲测试写入权限的文件存在于只读文件系统内。

EFAULT 参数pathname指针超出可存取内存空间。

EINVAL 参数mode 不正确。

ENAMETOOLONG 参数pathname太长。

ENOTDIR 参数pathname为一目录。

ENOMEM 核心内存不足

ELOOP 参数pathname有过多符号连接问题。

EIO I/O 存取错误。

特别提醒：使用access()作用户认证方面的判断要特别小心，例如在access()后再做open()的空文件可能会造成系统安全上的问题。

 实例：

#include <stdio.h>

#include <io.h>

int main(void)

{

    if ( !access("C://windows",0) )

         puts("C://windows EXISITS!");

    else

         puts("C://windows DOESN'T EXISIT!");

    return 0;

}

方法二：fopen函数判断文件是否存在

函数原型：FILE \*fopen (char \*filename, char \*type);

filename：文件路径

type：打开文件的方式（有r、w、r+、w+、a、rb、wb等等）

用于判断文件是否存在可以使用 r 或者 rb ，因为使用 其它方式的话，可能会自动建立文件。 返回值为NULL（打不开）和正数（能打开）。

特别提醒：用这种方法做出的判断是不完全正确的，因为有的文件存在，但是可能不可读。

方法三：DWORD d = GetFileAttributesA(const char\* filename); #include <windows.h>

     if (ftyp == INVALID\_FILE\_ATTRIBUTES)

         return false;  //something is wrong with your path!



     if (ftyp & FILE\_ATTRIBUTE\_DIRECTORY)

         return true;   // this is a directory!

C++方式：

#include <iostream>  
#include <fstream>  
using namespace std;  
#define FILENAME "stat.dat"  
int main()  
{  
     fstream \_file;  
     \_file.open(FILENAME,ios::in);  
     if(!\_file)  
     {  
         cout<<FILENAME<<"没有被创建";  
      }  
      else  
      {  
          cout<<FILENAME<<"已经存在";  
      }  
      return 0;  
}

602.发现read确实只是将一定的字节放到缓冲区，不会将缓冲区后面的其他部分置0

#include <stdio.h>

#include <fcntl.h>//open

#include <sys/stat.h>//S\_IWRITE|S\_IREAD

#include <unistd.h>

int main(void) {

int handle =open("/home/battery/Desktop/read\_this", O\_RDONLY, S\_IWRITE|S\_IREAD);

char buf[10] ={"321456798"};

read(handle, buf, 5);

puts(buf);

}

open是UNIX系统（包括LINUX、Mac等）的系统调用函数，区别于C语言库函数fopen。

|  |  |
| --- | --- |
|  | #include<fcntl.h>  int open(constchar\*pathname,intflags);  int open(constchar\*pathname,intflags,mode\_tmode);  返回值：成功则返回文件描述符，否则返回-1 |

对于open函数来说，第三个参数仅当创建新文件时（即 使用了O\_CREAT 时）才使用，用于指定文件的访问权限位（access permission bits）。pathname 是待[打开](http://baike.baidu.com/view/72184.htm)/创建文件的POSIX路径名（如/home/user/a.cpp）；flags 用于指定文件的打开/创建模式，这个参数可由以下[常量](http://baike.baidu.com/view/346799.htm)（定义于[fcntl.h](http://baike.baidu.com/view/3522799.htm)）通过逻辑位或逻辑构成。

|  |  |
| --- | --- |
|  | O\_RDONLY只读模式  O\_WRONLY只写模式  O\_RDWR读写模式 |

打开/创建文件时，至少得使用上述三个常量中的一个。以下常量是选用的：

|  |  |
| --- | --- |
|  | O\_APPEND每次写操作都写入文件的末尾  O\_CREAT如果指定文件不存在，则创建这个文件  O\_EXCL如果要创建的文件已存在，则返回-1，并且修改errno的值  O\_TRUNC如果文件存在，并且以只写/读写方式打开，则清空文件全部内容(即将其长度截短为0)  O\_NOCTTY如果路径名指向终端设备，不要把这个设备用作控制终端。  O\_NONBLOCK如果路径名指向FIFO/块文件/字符文件，则把文件的打开和后继I/O |

设置为非阻塞模式

|  |  |
| --- | --- |
|  | （nonblockingmode） |

以下三个常量同样是选用的，它们用于同步输入输出

|  |
| --- |
| O\_DSYNC等待物理I/O结束后再write。在不影响读取新写入的数据的  前提下，不等待  文件属性  更新。 |
| O\_RSYNCread等待所有写入同一区域的写操作完成后再进行  O\_SYNC等待物理I/O结束后再write，包括更新文件属性的I/O |

open和fopen的区别：  
1,fread是带缓冲的,read不带缓冲.    
2,fopen是标准c里定义的,open是POSIX中定义的.    
3,fread可以读一个结构.read在linux/unix中读二进制与普通文件没有区别.    
4,fopen不能指定要创建文件的权限.open可以指定权限.    
5,fopen返回文件指针,open返回文件描述符(整数).    
6,linux/unix中任何设备都是文件,都可以用open,read.

fopen和open只是为了获取句柄，并不会查看内容，所以文件再大也不会卡。综合来讲，这种读取方式不会卡。

603.布尔类型，C++里面默认有的是bool，C99开始支持布尔类型对应 \_Bool，需要加上stdbool.h头文件。

606. 仿图形界面就是老版BIOS那样。用编程实现的字符界面。

608.

const int INO\_BUFSIZE =(sizeof(struct inotify\_event)+MAX\_JPG\_NAME+4)\*MAX\_INO\_EVENTS;

char strJpgBuf[60\*1024];//error: initializer element is not computable at load time

610.常量全局变量只能用常量去赋值，不能使用常量表达式。全局中下面的会报错：

Const int EVENTS =30;

const int BUFSIZE =EENTS; 会报错

611.调用另外一个exe并向其传递参数：

方一、使用argc，argv

方二、使用标准输入函数与popen（程序调用+管道）

或者调用后其他的进程通信方式

612. 标准没有规定FILE有什么样的成员，我们也不必关心，因为这属于实现细节，不同的编译器可以有不同实现。所以不要让你的代码借助FILE结构的成员，比如说似乎linux和vc6下面的file结构体就不一样。这样没有移植性。

Raspberry里面我查看了/use/lib/stdio.h：

typedef struct \_IO\_FILE FILE;借助这句话百度了一下

然后在libio.h中定义了\_IO\_FILE。这里面定义了IO锁，fd在这里的别名是\_fileno

。然后借助这个成功实现了非阻塞fwrite。

613.sizeof对于char数组和对char指针是不一样滴：类型都不一样

#include <stdio.h>

int main(void) {

char buf[1000];

printf("%u", sizeof(buf));

}

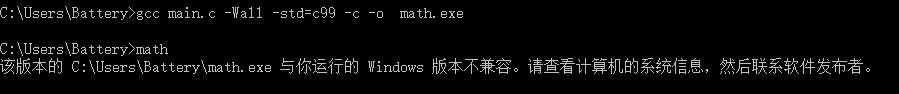
//output: 1000

615.手动调试的时候给每个函数加一个 fprintf(stderr, "handleInotify\n\n");而先暂时不使用syslog。

617. #include <assert.h>  
void assert( int expression );

　　assert的作用是现计算表达式 expression ，如果其值为假（即为0），那么它先向stderr打印一条出错信息，然后通过调用 abort 来终止程序运行

619.



这是因为我使用-c参数（进行编译和汇编），可能是cygwin里面gcc的问题，这就会导致math.exe虽产生了但是不能被执行。

620.fread、fwrite是阻塞的

Read、write是阻塞的

fread/fwrite相对于read/write效率高的原因是在用户层多设了一个缓冲区

文件读写主要牵涉到了如下五个操作：打开、关闭、读、写、定位。在Linux系统中，提供了两套API，一套是C标准API：fopen、fclose、fread、fwrite、fseek，另一套则是POSIX定义的系统API：open、close、read、write、seek。

其中POSIX定义的API是系统API，而C标准API是基于系统API的封装，并且提供了额外的缓冲的功能。因此也可以把它们叫做缓冲I/O函数和非缓冲I/O函数。**当用fopen打开文件时，除了分配文件句柄外，还额外申请了一个缓冲区。**PS：严格来讲，就算是POSIX的I/O函数，仍然是有内核I/O缓冲的，所以write也不一定是直接写到文件的，也可能写到内核I/O缓冲区中，至于究竟写到了文件中还是内核缓冲区中对于进程来说是没有太大差别的，我们不用太关注这一点。

要使用非阻塞IO操作，需要在open的时候制定O\_NONBLOCK标志。这样，如果设备暂时没有数据可读就返回-1，调用者应该试着再读一次（again）。这种行为方式称为轮询（Poll），调用者只是查询一下，而不是阻塞在这里死等，这样可以同时监视多个设备

fd = open("/dev/tty", O\_RDONLY|O\_NONBLOCK);

621.格式字符串错误就是指在格式化字符串的时候出现错误，比如C语言中的printf系列函数。

622. 用C++做的通用浏览器插件框架，这东西有些像Flash，只不过Flash用的AS脚本开发运行swf，我们用的C++开发运行dll

623．我们包含头文件有时候只是因为为了使用其中的声明所以如果使用一些编译器可以自己链接的函数而选择不包含头文件，我们自己加上申明就好，不知道这会不会让我们最后产生的可执行文件小一些。

625.

inFile.get();没有这个函数的话，没想到会继续往按理收已经没有内容的地方继续，然后后部分乱码

while (inFile)

{

inFile >> thread\_info[n\_thread].tid;

inFile >> thread\_info[n\_thread].role;

inFile >> thread\_info[n\_thread].delay;

inFile.get();读取了换行符

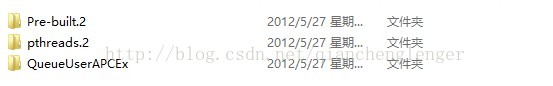
printf("%d %c %f", thread\_info[n\_thread].tid, thread\_info[n\_thread].role, thread\_info[n\_thread].delay);

n\_thread++;

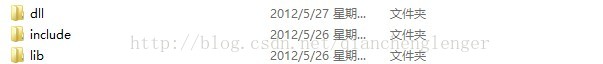
}

626.在VS中配置linux中的pthread.h就需要；

下载pthread包pthreads-w32-2-9-1，不知道是专门放在windows下还是别人从linux中抠出来了。



我们用到的主要是Pre-built.2这个文件夹下的三个文件夹，分别是动态链接库、头文件、静态链接库



之后就是需要把这几个文件放到对应的目录下面：

include文件夹中的三个文件直接拷贝到Visual Studio安装目录下VC->include文件夹下，例如我将include中文件拷贝到的位置是E:\Program Files\Microsoft Visual Studio 11.0\VC\include

 同样的办法与原因，我们也可以把lib文件夹下的内容拷贝到Visual Studio安装目录下默认的lib寻找路径中，即VC->lib中，例如我将lib文件夹下的x64与x86两个文件直接拷贝到E:\Program Files\Microsoft Visual Studio 11.0\VC\lib的下面。但是其实使用的就是x86文件夹下面的文件，所以可以直接将x86文件夹下面的文件放到lib中。

下面dll的存放我一直觉得可能反了，但是我还是照这个来了

把dll下的x64文件夹下的两个文件，即pthreadGC2.dll与pthreadVC2.dll拷贝到C:\Windows\System32下（用于64位程序的运行）

把dll下的x86文件夹下的五个文件，拷贝到C:\Windows\SysWOW64下（用于32位程序的运行），注意一下，千万不能将这些文件拷贝反位置，否则，程序运行时会提示说找不到对应的dll文件。这些在网上的很多文章中都被忽略掉了，所以我们特别提出。

最后在VS中加上#pragma comment(lib,"x86/pthreadVC2.lib")，如果上面没有放到x86文件夹下而是拿了出来，那么就不需在路径中加上x86。

库的使用方式要么是设置好环境变量要么就是直接丢到IDE的搜索路径之中。之后就需要配置ide，让它自己来找或者你在代码中引入lib文件。

627.5. pcap.h是C的抓包库

628.libcurl: #include <curl/curl.h>

libcurl是一个跨平台的网络协议库，支持http, https, ftp, gopher, telnet, dict, file, 和ldap 协议。libcurl同样支持HTTPS证书授权，HTTP POST, HTTP PUT, FTP 上传, HTTP基本表单上传，代理，cookies,和用户认证。

利用libcurl完成传输任务的流程：

1.调用**curl\_global\_init()**初始化libcurl  
2.调用**curl\_easy\_init()**函数得到 easy interface型指针  
3.调用**curl\_easy\_setopt()**设置传输选项  
4.根据**curl\_easy\_setopt()**设置的传输选项，实现回调函数以完成用户特定任务  
5.调用**curl\_easy\_perform()**函数完成传输任务  
6.调用**curl\_easy\_cleanup()**释放内存  
在整过过程中设置**curl\_easy\_setopt(**)参数是最关键的，几乎所有的libcurl程序都要使用它。

**CURLcode curl\_global\_init(long flags);**

这个函数只能用一次。(其实在调用curl\_global\_cleanup 函数后仍然可再用)，如果这个函数在curl\_easy\_init函数调用时还没调用，它讲由libcurl库自动调用。虽然libcurl是线程安全的，但curl\_global\_init是不能保证线程安全的，所以不要在每个线程中都调用curl\_global\_init，应该将该函数的调用放在主线程中.（比如说在多个线程中直接使用curl\_easy\_init）

参数：

CURL\_GLOBAL\_ALL                      //初始化所有的可能的调用。  
CURL\_GLOBAL\_SSL                      //初始化支持 安全套接字层。  
CURL\_GLOBAL\_WIN32            //初始化win32套接字库。  
CURL\_GLOBAL\_NOTHING         //没有额外的初始化。

**void curl\_global\_cleanup(void);**

**char \*curl\_version( );** //当前libcurl库的版本

**CURL \*curl\_easy\_init( );**

**void curl\_easy\_cleanup(CURL \*handle);**

**CURLcode curl\_easy\_setopt(CURL \*handle, CURLoption option, parameter);**

参数：

1 CURL类型的指针  
2 各种CURLoption类型的选项.(都在curl.h库里有定义,man 也可以查看到)  
3 parameter 这个参数 既可以是个函数的指针,也可以是某个对象的指针,也可以是个long型的变量.它用什么这取决于第二个参数.  
CURLoption 这个参数的取值很多.具体的可以查看man手册.

1.CURLOPT\_URL   
设置访问URL。直接执行就会在stdout中输出  
2.CURLOPT\_WRITEFUNCTION，CURLOPT\_WRITEDATA  
回调函数原型为：**size\_t function( void \*ptr, size\_t size, size\_t nmemb, void \*stream);**函数将在libcurl接收到数据后被调用，因此函数多做数据保存的功能，如处理下载文件。CURLOPT\_WRITEDATA 用于表明CURLOPT\_WRITEFUNCTION函数中的stream指针的来源。

如果你没有通过CURLOPT\_WRITEFUNCTION属性给easy handle设置回调函数，libcurl会提供一个默认的回调函数，它只是简单的将接收到的数据打印到标准输出。你也可以通过 CURLOPT\_WRITEDATA属性给默认回调函数传递一个已经打开的文件指针，用于将数据输出到文件里。

fp是个文件可写指针

curl\_easy\_setopt(curl, CURLOPT\_WRITEDATA, fp); //返回的http头

curl\_easy\_setopt(curl, CURLOPT\_HEADERDATA, fp); //返回的html主体数据  
3.CURLOPT\_HEADERFUNCTION，CURLOPT\_HEADERDATA  
回调函数原型为 size\_t function( void \*ptr, size\_t size,size\_t nmemb, void \*stream); libcurl一旦接收到http 头部数据后将调用该函数。CURLOPT\_WRITEDATA 传递指针给libcurl，该指针表明CURLOPT\_HEADERFUNCTION 函数的stream指针的来源。  
4.CURLOPT\_READFUNCTION CURLOPT\_READDATA  
libCurl需要读取数据传递给远程主机时将调用CURLOPT\_READFUNCTION指定的函数，函数原型是：size\_t function(void \*ptr, size\_t size, size\_t nmemb,void \*stream). CURLOPT\_READDATA 表明CURLOPT\_READFUNCTION函数原型中的stream指针来源。  
5.CURLOPT\_NOPROGRESS，CURLOPT\_PROGRESSFUNCTION，CURLOPT\_PROGRESSDATA  
跟数据传输进度相关的参数。CURLOPT\_PROGRESSFUNCTION 指定的函数正常情况下每秒被libcurl调用一次，为了使CURLOPT\_PROGRESSFUNCTION被调用，CURLOPT\_NOPROGRESS必须被设置为false，CURLOPT\_PROGRESSDATA指定的参数将作为CURLOPT\_PROGRESSFUNCTION指定函数的第一个参数  
6.CURLOPT\_TIMEOUT，CURLOPT\_CONNECTIONTIMEOUT:  
CURLOPT\_TIMEOUT 由于设置传输时间，CURLOPT\_CONNECTIONTIMEOUT 设置连接等待时间  
7.CURLOPT\_FOLLOWLOCATION  
设置重定位URL  
8.CURLOPT\_RANGE: CURLOPT\_RESUME\_FROM:  
断点续传相关设置。CURLOPT\_RANGE 指定char \*参数传递给libcurl，用于指明http域的RANGE头域，例如：  
表示头500个字节：bytes=0-499  
表示第二个500字节：bytes=500-999  
表示最后500个字节：bytes=-500  
表示500字节以后的范围：bytes=500-  
第一个和最后一个字节：bytes=0-0,-1  
同时指定几个范围：bytes=500-600,601-999  
CURLOPT\_RESUME\_FROM 传递一个long参数给libcurl，指定你希望开始传递的 偏移量。

**CURLcode curl\_easy\_perform(CURL \*handle);**

描述:这个函数在初始化CURL类型的指针 以及curl\_easy\_setopt完成后调用. 就像字面的意思所说perform就像是个舞台.让我们设置的  
option 运作起来.参数:  
CURL类型的指针

返回值：

1.    CURLE\_OK   
    任务完成一切都好  
2     CURLE\_UNSUPPORTED\_PROTOCOL  
    不支持的协议，由URL的头部指定  
3     CURLE\_COULDNT\_CONNECT  
    不能连接到remote 主机或者代理  
4     CURLE\_REMOTE\_ACCESS\_DENIED  
    访问被拒绝  
5     CURLE\_HTTP\_RETURNED\_ERROR  
    Http返回错误  
6     CURLE\_READ\_ERROR

当使用libcurl发送http请求时，它会自动添加一些http头。我们可以通过CURLOPT\_HTTPHEADER属性手动替换、添加或删除相应 的HTTP消息头。  
    Host  
    http1.1（大部分http1.0)版本都要求客户端请求提供这个信息头。  
    Pragma  
    "no-cache"。表示不要缓冲数据。  
    Accept  
    "\*/\*"。表示允许接收任何类型的数据。  
    Expect  
    以POST的方式向HTTP服务器提交请求时，libcurl会设置该消息头为"100-continue"，它要求服务器在正式处理该请求之前，返回一 个"OK"消息。如果POST的数据很小，libcurl可能不会设置该消息头

struct curl\_slist \*headers=NULL; /\* init to NULL is important \*/

增加消息头：

headers = curl\_slist\_append(headers, "Hey-server-hey: how are you?");

headers = curl\_slist\_append(headers, "X-silly-content: yes");

/\* pass our list of custom made headers \*/

curl\_easy\_setopt(easyhandle, CURLOPT\_HTTPHEADER, headers);

curl\_easy\_perform(easyhandle); /\* transfer http \*/

curl\_slist\_free\_all(headers); /\* free the header list \*/

对于已经存在的消息头，可以重新设置它的值：

headers = curl\_slist\_append(headers, "Accept: Agent-007");

headers = curl\_slist\_append(headers, "Host: munged.host.line")

删除消息头:  
对于一个已经存在的消息头，设置它的内容为空，libcurl在发送请求时就不会同时提交该消息头：headers = curl\_slist\_append(headers, "Accept:");

获取http应答头信息

**CURLcode curl\_easy\_getinfo(CURL \*curl, CURLINFO info, ... );**   
info参数就是我们需要获取的内容，下面是一些参数值:  
1.CURLINFO\_RESPONSE\_CODE  
获取应答码  
2.CURLINFO\_HEADER\_SIZE  
头大小  
3.CURLINFO\_COOKIELIST  
cookies列表

关于密码：

  一些协议支持在URL中直接指定用户名和密码，类似于： protocol://user:password@example.com/path/。libcurl能正确的识别这种URL中的用户名与密码并执行 相应的操作。如果你提供的用户名和密码中有特殊字符，首先应该对其进行URL编码。

也可以通过CURLOPT\_USERPWD属性来设置用户名与密码。参数是格式如 user:password 的字符串：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_USERPWD, "user\_name:password")**;   
    有时候在访问代理服务器的时候，可能时时要求提供用户名和密码进行用户身份验证。这种情况下，libcurl提供了另 一个属性CURLOPT\_PROXYUSERPWD：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_PROXYUSERPWD, "user\_name:password");**

在使用SSL时，可能需要提供一个私钥用于数据安全传输，通过CURLOPT\_KEYPASSWD来设置私钥：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_KEYPASSWD, "keypassword");**

    当前版本的libcurl支持的验证方法有：basic, Digest, NTLM, Negotiate, GSS-Negotiate and SPNEGO。（译者感叹：搞Web这么多年，尽然不知道这些Http的验证方式，实在惭愧。）可以通过CURLOPT\_HTTPAUTH属性来设置具体 的验证方式：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_HTTPAUTH, CURLAUTH\_DIGEST);**  
    向代理服务器发送验证信息时，可以通过CURLOPT\_PROXYAUTH设置验证方式：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_PROXYAUTH, CURLAUTH\_NTLM);**  
    也可以同时设置多种验证方式（通过按位与）， 使用‘CURLAUTH\_ANY‘将允许libcurl可以选择任何它所支持的验证方式。通过CURLOPT\_HTTPAUTH或 CURLOPT\_PROXYAUTH属性设置的多种验证方式，libcurl会在运行时选择一种它认为是最好的方式与服务器通信：  
    **curl\_easy\_setopt(easy\_handle, CURLOPT\_HTTPAUTH, CURLAUTH\_DIGEST|CURLAUTH\_BASIC);**

多线程问题：

首先一个基本原则就是：绝对不应该在线程之间共享同一个libcurl handle(CURL \*对象)，不管是easy handle还是multi handle（本文只介绍easy\_handle）。libcurl是线程安全的，但有两点例外：信号(signals)和SSL/TLS handler。信号用于超时失效名字解析(timing out name resolves)。libcurl依赖其他的库来支持SSL/STL，所以用多线程的方式访问HTTPS或FTPS的URL时，应该满足这些库对多线程 操作的一些要求。

#include <stdio.h>

#include <curl/curl.h>

#include <stdlib.h>

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

{

CURL \*curl; //定义CURL类型的指针

CURLcode res; //定义CURLcode类型的变量，保存返回状态码

if(argc!=2)

{

printf("Usage : file <url>;\n");

exit(1);

}

curl = curl\_easy\_init(); //初始化一个CURL类型的指针

if(curl!=NULL)

{

//设置curl选项. 其中CURLOPT\_URL是让用户指 定url. argv[1]中存放的命令行传进来的网址

curl\_easy\_setopt(curl, CURLOPT\_URL, argv[1]);

//调用curl\_easy\_perform 执行我们的设置.并进行相关的操作. 在这 里只在屏幕上显示出来.

res = curl\_easy\_perform(curl);

//清除curl操作.

curl\_easy\_cleanup(curl);

}

return 0;

}

显示下载进度条：

#include <stdio.h>

#include <gtk/gtk.h>

#include <curl/curl.h>

#include <curl/types.h> /\* new for v7 \*/

#include <curl/easy.h> /\* new for v7 \*/

GtkWidget \*Bar;

////这个函数是为了符合CURLOPT\_WRITEFUNCTION而构造的

//完成数据保存功能

size\_t my\_write\_func(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

{

return fwrite(ptr, size, nmemb, stream);

}

//这个函数是为了符合CURLOPT\_READFUNCTION而构造的

//数据上传时使用

size\_t my\_read\_func(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

{

return fread(ptr, size, nmemb, stream);

}

//这个函数是为了符合CURLOPT\_PROGRESSFUNCTION而构造的

//显示文件传输进度，t代表文件大小，d代表传 输已经完成部分

int my\_progress\_func(GtkWidget \*bar,

double t, /\* dltotal \*/

double d, /\* dlnow \*/

double ultotal,

double ulnow)

{

/\* printf("%d / %d (%g %%)\n", d, t, d\*100.0/t);\*/

gdk\_threads\_enter();

gtk\_progress\_set\_value(GTK\_PROGRESS(bar), d\*100.0/t);

gdk\_threads\_leave();

return 0;

}

void \*my\_thread(void \*ptr)

{

CURL \*curl;

CURLcode res;

FILE \*outfile;

gchar \*url = ptr;

curl = curl\_easy\_init();

if(curl)

{

outfile = fopen("test.curl", "w");

curl\_easy\_setopt(curl, CURLOPT\_URL, url);

curl\_easy\_setopt(curl, CURLOPT\_WRITEDATA, outfile);

curl\_easy\_setopt(curl, CURLOPT\_WRITEFUNCTION, my\_write\_func);

curl\_easy\_setopt(curl, CURLOPT\_READFUNCTION, my\_read\_func);

curl\_easy\_setopt(curl, CURLOPT\_NOPROGRESS, 0L);

curl\_easy\_setopt(curl, CURLOPT\_PROGRESSFUNCTION, my\_progress\_func);

curl\_easy\_setopt(curl, CURLOPT\_PROGRESSDATA, Bar);

res = curl\_easy\_perform(curl);

fclose(outfile);

/\* always cleanup \*/

curl\_easy\_cleanup(curl);

}

return NULL;

}

而Main函数里面过多涉及到对gtk的控制，我就不添加了。

断点续传：

curl\_easy\_setopt(handle,CURLOPT\_RESUME\_FROM\_LARGE,use\_resume?local\_file\_len:0); 第三个参数是文件大小（已下载为例，打开文件还要是a+模式）

curl\_easy\_setopt(curlhandle, CURLOPT\_WRITEDATA, f);

629. rand()的使用：

#include <stdlib.h>

#include <windows.h>

#include <stdio.h>

#include <time.h>

int main(void) {

srand((unsigned)time(0));

printf("%llu",RAND\_MAX);

for(int i1 =0; i1<3 ;++i1){

printf("%ld", (1+(int)(100\*rand()/(RAND\_MAX+1.0)))%2);

}

return 0;

}

RAND\_MAX真的很大很大。。。。标准规定RAND\_MAX 的值应至少为32767

windows下32767（使用%u或者%lu输出） ，使用18446744065119649791 (%llu输出)，但是看样子是32767，不清楚%llu的输出怎么并没有向下兼容。

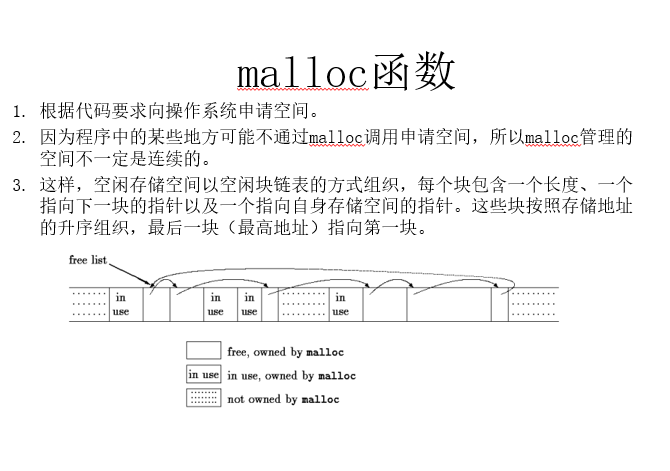
linux下2147483647

因为标准，导致RAND\_MAX可能成为可移植性的阻碍。

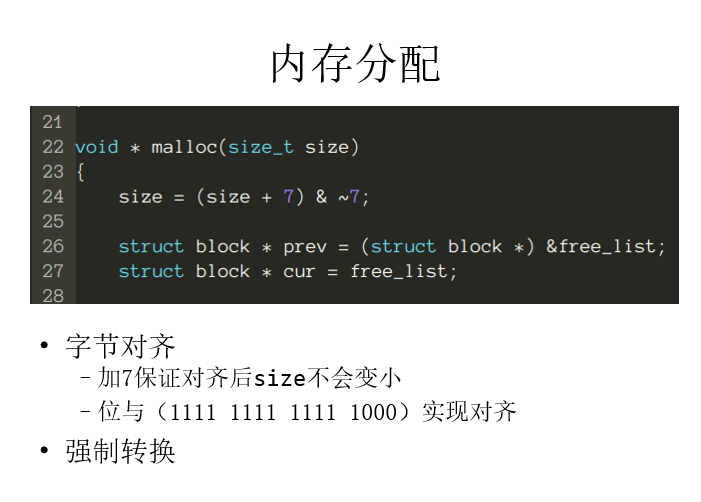
631.将VS安装在其他盘，C盘还是一下子少了11个G。

632. iterator是消耗型的，即每一个值被使用过后，就消失了。因此，你可以将以上的操作2理解成pop。对iterator进行遍历之后，其就变成了一个空的容器了。我们对一个iterable用for ... in ...进行迭代时，实际是先通过调用iter()方法得到一个iterator，假设叫做X。然后循环地调用X的next()方法取得每一次的值，直到iterator为空，返回的StopIteration作为循环结束的标志。for ... in ...会自动处理StopIteration异常，从而避免了抛出异常而使程序中断。

634. 一般的用户习惯，前面有"m\_"的前缀，表示类成员变量；有"g\_"的前缀，表示全局变量



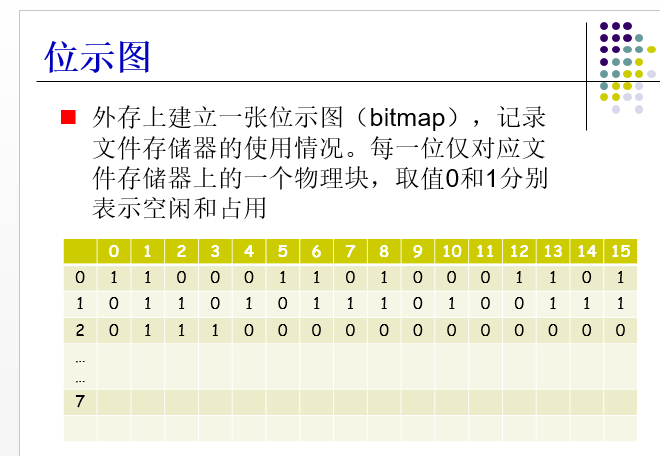
如果是按照8字节进行对齐自己编写malloc算法：



Size+7就会导致最后3位可能会向前面进一。

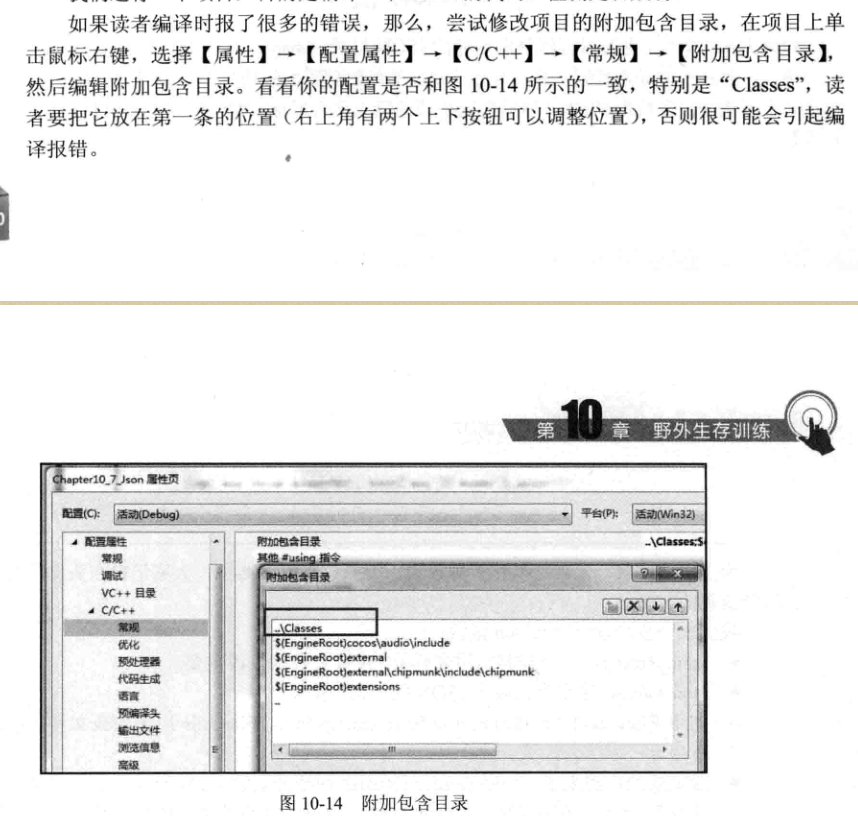
已分配内存与未分配内存都是断的，所以估摸着都是用链表进行实现。

不过在外存上，这种不连续的情况是使用了，用于查看指定位置是否存有数据：



636.jsoncpp库是专门用来处理json的。

在VS中创建筛选器（一个虚拟文件夹），然后右键添加现有项中选中所有的json包中文件。



640. \_CrtDumpMemoryLeaks()就是检测从程序开始到执行该函数进程的堆使用情况，通过使用\_CrtDumpMemoryLeaks()我们可以进行简单的内存泄露检测，调用这个函数会在stdout中输出监测到内存泄露信息。



linux+gcc用valgrind，windows+msvc用vs调试功能。

编译通过后，用cppcheck跑一遍，能解决不少代码中明显的内存泄漏；运行的时候用valgrind跑，可以检查运行时候存在的内存泄漏。

valgrind是一套工具集：

        memcheck ------> 这是valgrind应用最广泛的工具，一个重量级的内存检查器，能够发现开发中绝大多数内存错误使用情况，比如：使用未初始化的内存，使用已经释放了的内存，内存访问越界等。

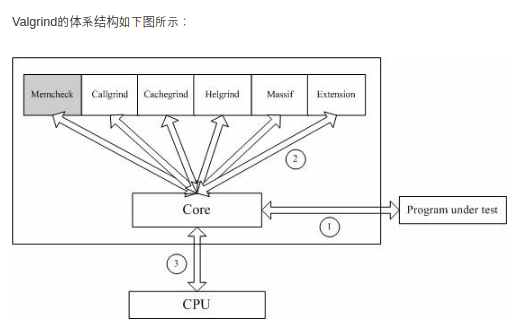
        callgrind ------> 它主要用来检查程序中函数调用过程中出现的问题。

        cachegrind ------> 它主要用来检查程序中缓存使用出现的问题。

        helgrind ------> 它主要用来检查多线程程序中出现的竞争问题。

        massif ------> 它主要用来检查程序中堆栈使用中出现的问题。

        extension ------> 可以利用core提供的功能，自己编写特定的内存调试工具



通过bitArray就能实现下面大部分功能：

使用未初始化的内存(Use of uninitialised memory)  
使用已经释放了的内存(Reading/writing memory after it has been free'd)  
使用超过malloc分配的内存空间(Reading/writing off the end of malloc'd blocks)  
对堆栈的非法访问(Reading/writing inappropriate areas on the stack)  
申请的空间是否有释放(Memory leaks – where pointers to malloc'd blocks are lost forever)

检测内存泄露:valgrind --tool=memcheck --leak-check=full ./val

definitely lost：内存没有被释放，且没有任何指针指向这里。肯定泄漏了。报告给出的堆栈是内存被分配时的调用堆栈，它可以基本明确内存是由什么业务逻辑创建的。   
still reachable：是说内存没有被释放，尽管如此仍有指针指向，内存仍在使用中，这可以不算泄露。  
possibly lost：是说可能有泄漏，一般是有二级指针（指针的指针）等复杂情况不易于追踪时出现。   
suppressed：统计了使用valgrind的某些参数取消了特定库的某些错误，会被归结到这里

还有看程序执行完一次功能之后（可能就是一次执行完）那些增加的内存。

Helgrind 寻找内存中被多个线程访问，而又没有一贯加锁的区域

检测原理：

使程序运行一遍，之后对比内存使用情况，在程序结束之后多出来的，不正常的内存即内存泄露部分。这个过程中可以直接维护一个list。其中如果要检测的是堆内存的泄露，只需要hook住malloc/realloc/free和new/delete就可以了（其实new/delete最终也是用malloc/free的，所以只要截获前面一组即可）；要检测BSTR的泄漏，就需要截获SysAllocString/SysFreeString；要检测HMENU的泄漏，就需要截获CreateMenu/ DestroyMenu。如果程序是循环运行的，只要支持用户能够在程序运行过程中在合适的位置下一个断点，并将内容输出，用户就能够判断是否是内存泄露。

643.

cannot declare member function `static int  
   CTest::testEtrance(short unsigned int)' to have static linkage

错误的原因就是在实现的地方不需要static了, 只在声明的地方需要static

644.内联汇编：

VC中对内联汇编的使用

\_\_asm {

xxxx

xxxx

}

Cygwin或者Mingw 中GCC的内联汇编使用：

\_\_asm\_\_ \_\_volatile\_\_(

"xorps %%xmm0,%%xmm1\n\t"

"1:\n\t"

"movdqa %%xmm0,%%xmm1\n\t"

"pcmpgtd (%1),%%xmm1\n\t"

"movl %%eax,%0\n\t"

:"=m"(mx)

:"r"(a),"r"((long long)N),"r"(tmp)

:"eax");

一共包含4个部分,之间使用:分隔.

* instruction指令.每条指令之后最好使用"\n\t"结尾,这样在gcc产生汇编格式比较好.
* output operand输出.每个输出部分使用,分隔."="作为修饰符,"m"表示存放位置/约束符,()里面表示对应C程序值.
* input operand输入.这个部分和输出是一样的.
* clobber.这个部分是告诉gcc在这条指令里面我们会修改什么值.

汇编扩展以\_\_asm\_\_开头表示后面部分为汇编,\_\_volatile\_\_严禁将此处的汇编语句和其他语句进行重组优化。

寄存器使用%%开头。

直接数使用$开头的十进制。

%0,%1,%2用来引用输出和输入的内容.

645. 你一直new delete，时间久了会产生内存碎片。这个是性能问题

646.内存基本没有增长，但是因为句柄的不断占有而使得程序崩溃。比如说Qt里面的QNetworkAccessManager的get、post方法。

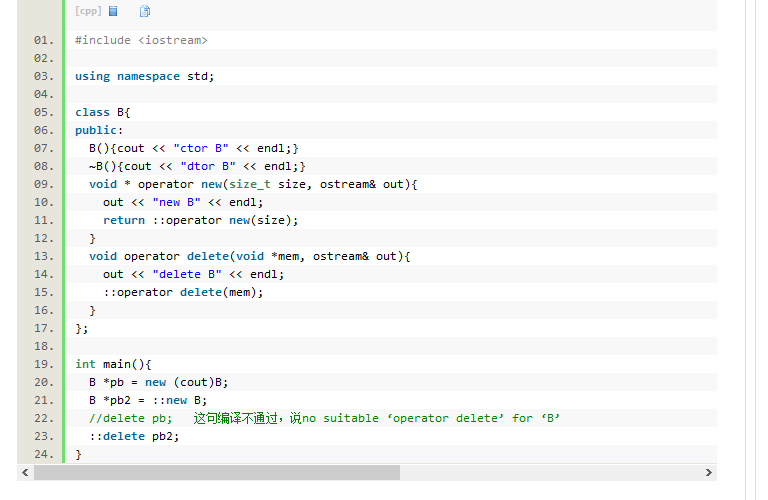
650. 首先我们区分下几个容易混淆的关键词：new、operator new、placement new

new和delete操作符我们应该都用过，它们是对堆中的内存进行申请和释放，而这两个都是不能被重载的。要实现不同的内存分配行为，需要重载operator new，而不是new和delete。

placement new是operator new的一个重载版本，增加了函数参数，只是我们很少用到它。如果你想在已经分配的内存中创建一个对象，使用new是不行的。也就是说placement new允许你在一个已经分配好的内存中（栈或堆中）构造一个新的对象。

char \* ps =new char[40];

xxx xxname=new(ps) xxx 比如下面：



上图中 :: operator new(size);创建的是指定大小的内存，:: operator不能被省略。

注释语句的错误是因为placement delete是编译器在placement new出错的时候进行调用，会直接释放内存，而不是调用析构函数. 自己析构内存的时候，只能是调用析构函数，或者调用operator delete这个函数名字，据说是这样。

Point\* p = new (arena) Point;

delete p; // Wrong. You release the arena memory

p = new (arena) Point;

// correct method

p->~P();

p = new (arena) Point;

主要是因为placement new的时候C++是认不出来的内存上类型的分配。

651. 字节对齐：

如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。

字节对齐的目的是加快寻址速度：需要字节对齐的根本原因在于CPU访问数据的效率问题，有的CPU或者说一般的CPU并不是支持直接访问指定的内存地址，下面这个例子就很恰当。假设上面4字节整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。从汇编就可以知道CPU访问数据的类型与字节。

通过sizeof可以进行结构体对齐字节数的判断

有一些系统（比如sparc系统）对对齐要求很严格，不然会出现错误（程序运行的时候出现段错误）。

VC和GCC默认的都是4字节对齐，编程中可以使用#pragma pack(n)指定对齐模数，通过#pragma pack ()取消自己的字节对齐设置。出现以上差异的原因在于，VC和GCC中对于double类型的对齐方式不同。  
   
  
 651.5.

使用GCC编译器：

char c;

double b;

double a;

c的地址最大，b，a按照字节对齐的方式地址逐渐减小。

652.据说VS2017的安装速度比2015快很多。

653.firefox是使用C以及gtk进行编写。C++他们说不可控，就是不知道编译器替你做了什么。

654. vs插件  很多涉密公司都不让安装的，传说盗取代码，也不知道真的假的



655. 看来并不是等价，我是打算在VS中通过for循环来输出vector<string>



这个是真的神奇

658. 由于编译器的优化，循环体中的局部变量再分配的时候实际上可能只存在一次内存分配。

659. pair<string, unsigned> a[100];

a[1].first() string

a[1].second() unsigned

也有以断言为参数的sort

660. C-C也会被当做继续执行的信号，除非你一直按着C-C，使得这个信号被程序当做普通的输入接收：

while(1) {

system("pause");

}

661.qq将协议放到一个rtf文件中，在安装的时候会进行显示

662.

#include <stdio.h>

int main(void) {

int \*p =3;

printf("%x", &\*p); //直接&3会出错，依旧存在warning：initialization makes pointer from integer without a cast

return 0;

}

即便是C语言中，不能对左值使用&。所以想了一招p。

663. 有时候我们需要利用代码将指定路径(folderpath)的文件夹打开（效果相当于我们双击了此文件夹），为了方便初次遇到这个问题的朋友，以下我罗列了几种可用的方法：

         方法一：System("start     folderpath"); 但是没想到

if (GetFileAttributesA("storage")&FILE\_ATTRIBUTE\_DIRECTORY)

system("start storage");

if判断的是相对路径，而system里面完全不知道是哪里打开，不是相对路径，也不是cmd里面的默认路径。所以需要加上绝对路径进行束缚。

         方法二：System("explorer.exe     folderpath");

         方法三：ShellExecute(null,"open",null,null,folderpath,SW\_SHOWNORMAL);

666. 在 ASCII 里本来就有的数字、标点、字母都统统重新编了两个字节长的编码，这就是常说的”全角”字符，而原来在127号以下的那些就叫”半角”字符了。 中国人民看到这样很不错，于是就把这种汉字方案叫做 “GB2312“。GB2312 是对 ASCII 的中文扩展。

UTF-8就是在互联网上使用最广的一种unicode的实现方式，这是为传输而设计的编码（编码并不是定长），并使编码无国界(可以有足够长的位数)，这样就可以显示全世界上所有文化的字符了。Unicode是定长编码。

有时候要向终端中运行的程序进行很繁琐的输入，可以借助记事本和管道运算符。

linux下一个中文三个字节UTF编码，Windows下GB2312编码一个中文两字节

#include <stdio.h>  
  
**int** main(**void**) {  
 **int** c = '你 ';  
 printf("%x,%c%c%c",c,0xe4,0xbd,0xa0);  
 **return** 0;  
}

667. char\*str1="\x1203456" 后面的数字全被解读了，被当做超范围解读了直到读到一个不是十六进制的。用strstr在jpeg流中找\xFF\xD8是有问题的，只能找到开头的，不如自己的一个循环。这个函数还是用在对一般可见字符串的查找比较好。

而且在C++里面，如果要用指针指向字符创常量需要加上一个const。

668. warning: pointer/integer type mismatch in conditional expression

isEndFind ? isEndFind=0:memcpy(dataBuf+imgSize, buf, size), imgSize+=size;

条件表达式限制了放入其中的表达式类型。

这种很危险，因为一旦条件表达式无效之后，程序的逻辑就会有大问题。

670.环境变量操作：

环境变量按优先级有系统环境变量-》用户环境变量-》程序环境变量（目前只接触过服务器）。

char \*value = getenv( ENV[ i ].c\_str() );   获取环境变量的值

671. SEH 基于线程，每一个线程都有自己的SEH 函数，这点从线程信息块可以知道

当编译器的SEH支持 层把原始的操作系统SEH的复杂性封装起来的时候，它同时也把原始的操作系统SEH的细节隐藏了起来。在VC++或者Borland的运行时库的源码中有对异常处理。

**当一个线程出现错误时，操作系统（内核态处理）调用用户定义的一个回调函数。**这个回调函数可以做它想做的一切。例如它可以修复错误，或者它也可以播放一段音乐。无论回调函数做什么，它最后都要返回一个值来告诉系统下一步做什么。

1.操作系统通过**EXCEPTION\_REGISTRATION**结构（其实这个结构是异常处理链表中的一部分）找到这个异常处理的回调函数，由于SHE未公开的原因，读者可以使用内核调试器，如KD或SoftICE来查看这个结构的定义。而这个结构体就被线程信息块中的一个成员指向。EXCEPTION\_REGISTRATION结构创建在堆栈上。

        push handler // handler函数的地址  
        push FS:[0] // 前一个handler函数的地址  
        mov FS:[0],ESP // 安装新的EXECEPTION\_REGISTRATION结构

fs:[0]地址的所指向的一块数据区，它其实就是用来保存当前进程关键信息的进程控制块数据结构，在内核态环境下，它表示的是进行控制块KPCB结构体，在用户态环境下，它的表示形式是线程控制块的用户态形式TEB。

**当 异常发生时，系统遍历EXCEPTION\_REGISTRATION结构链表，直到它找到一个处理这个异常的处理程序。一旦找到，系统就再次遍历这个链表，直到处理这个异常的结点为止。在这第二次遍历中，系统将再次调用每个异常处理函数。关键的区别是，在第二次调用中，异常标志被设置为2。这个值被定义 为EH\_UNWINDING。**

2.下面就是这个回调函数：

EXCEPTION\_DISPOSITION  
\_\_cdecl \_except\_handler( struct \_EXCEPTION\_RECORD \*ExceptionRecord,  
                        void \* EstablisherFrame,  
                        struct \_CONTEXT \*ContextRecord, 上下文  
                        void \* DispatcherContext);

第一个结构体参数里面是

typedef struct \_EXCEPTION\_RECORD {  
   DWORD ExceptionCode; 异常的代码表示，比如内存访问异常是0xc0000005

通过在WINNT.H中搜索以STATUS\_开头的＃define定义  
   DWORD ExceptionFlags;  
   struct \_EXCEPTION\_RECORD \*ExceptionRecord;  
   PVOID ExceptionAddress; 代表异常发生的地址

   DWORD NumberParameters;  
   DWORD ExceptionInformation[EXCEPTION\_MAXIMUM\_PARAMETERS];  
} EXCEPTION\_RECORD;

**当一个异常处理程序拒绝处理某个异常时，它实际上也就拒绝决定流程最终将从何处恢复。只有处理某个异常的异常处理程序才能决定待所有异常处理代码执行完毕之后流程将从何处恢复。**

当操作系统看到返回值为ExceptionContinueExecution时，它将其理解为你已经修复了问题，而引起错误的那条指令应该被重新执行

1. volatile修饰函数据说能够帮助编译器优化，告诉编译器这个函数不会返回，于是调用后不会保存返回地址。对于oom()和do\_exit()函数而言，它们是永远都不会返回的，如果还将调用它们的函数的返回地址保存在堆栈上的话，是没有任何意义的。还有作用是修饰变量、汇编块（\_\_volatile\_\_）。

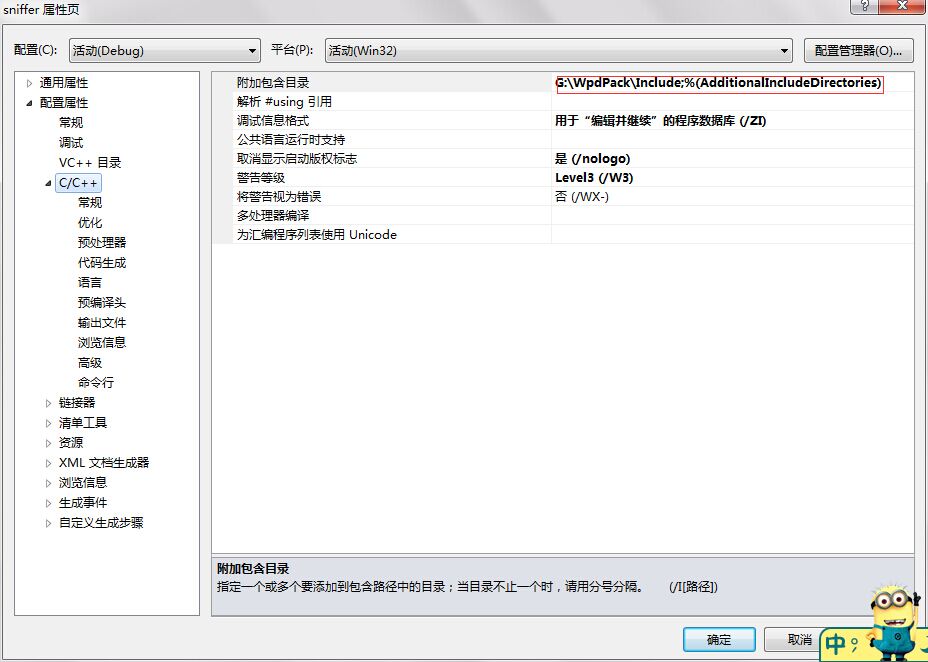
对变量使用，避免编译器将从内存读取优化成为从寄存器读取（就是前面指令的结果，或者读取的数据）。行为上有点接近java的volatile的保证，但是只是接近。

674. 现今的跨平台实现方法主要有两种，一种是“一次编译，到处运行”（解释型语言），还有一种是“一次编码，到处编译”（但是不能使用系统api），QT、cocos2d-x等等。Windows、Linux、Unix目前都已经遵循了POSIX接口规范，这就使得他们中大部分的常用接口都是近似的，所以在跨平台框架里，这大部分的代码是可以共用的

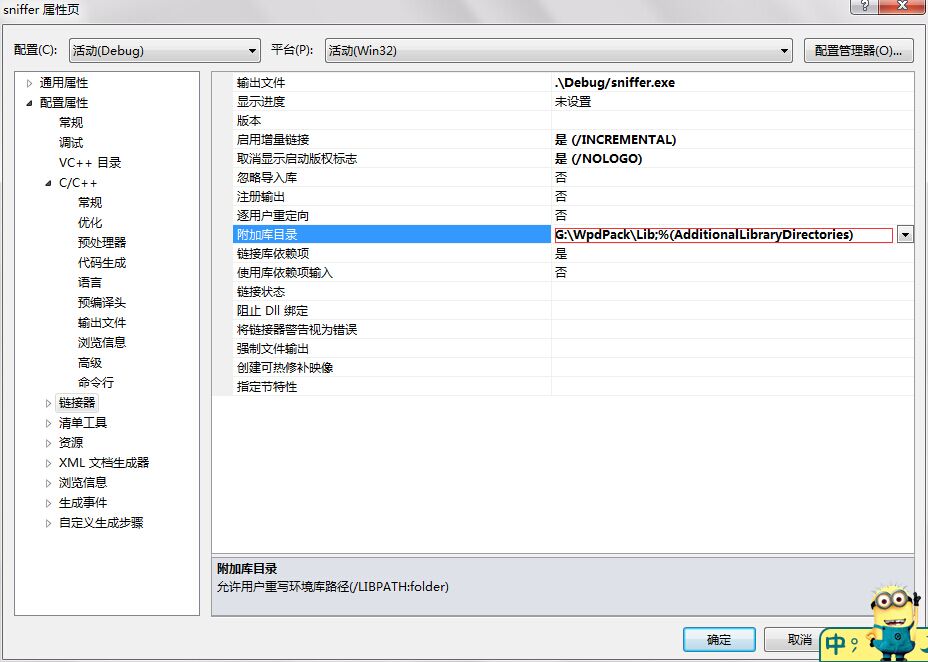
675.VS中使用WinPcap：

在C/C++中导入wpdpack包：

然后再在链接器中导入Wpdpack包：



环境搭建完成。



677.

class test

{

public:

enum type{One,Two, Three}; //sizeof为1

private:

};

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

{

cout << sizeof(test)<<endl; // 1 其实enum的申明并不占有类产生对象的内存空间，所以输出1是因为什么都没有的时候sizeof一个类就是1 。

cout << sizeof(test::type)<<endl; // 4

return 0;

}

682.

枚举类型不具备名字空间的作用。在一个作用域中在，如果两个枚举类型的值相同，则会编译出错。以下就是一个出错的例子：

struct CEType

{

enum EType1 { e1, e2 };

enum EType2 { e1, e2 };

}

同一枚举中不同的名字可以有相同的值：enum uu{id=1,name=1};

1. 字符串比较：

数组是数组，指针是指针，数组不能直接用==参与字符串比较。

#include <stdio.h>

#include <string.h>

int main(void) {

char \*a ="abc";

const char \*b = "abc";

const char c[] = "abc";

if(a==b) printf("a==b\n");

if(a==c) printf("a==c\n");

if(c=="abc") printf("c==\"abc\"\n");

return 0;

}

输出：

a==b

数组的相等：

#include <stdio.h>

#include <string.h>

int main(void) {

const int arr[] = {123};

int a = arr;

unsigned b =arr;

long c =&arr;

if(a==arr) printf("a==arr\n");

if(b==arr) printf("b==arr\n");

if(c==arr) printf("c==arr\n");

printf("%d %d\n",sizeof(arr),sizeof(&arr));

return 0;

}

c==arr

4 8

如果将a的类型改为long，那么a==arr

数组名字参与==比较或者=赋值的时候就是以数组头地址去参与。但是将数组名作为sizeof参数，查到的是数组元素的大小。

1. 关于变量与堆栈的地址关系

#include <stdio.h>

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

{

long l;

int i1;

char buf[32]; //这里分配到的大小比32大,会多出4个字节或者8个字节

int i1;

printf("%p %p %p %p %p %p",&l,&i1,&buf,&i2,&argv,&argc);

//argc 16字节

//argv 12字节 gcc从右到左压入栈

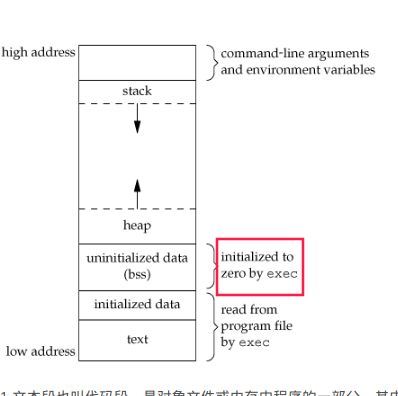
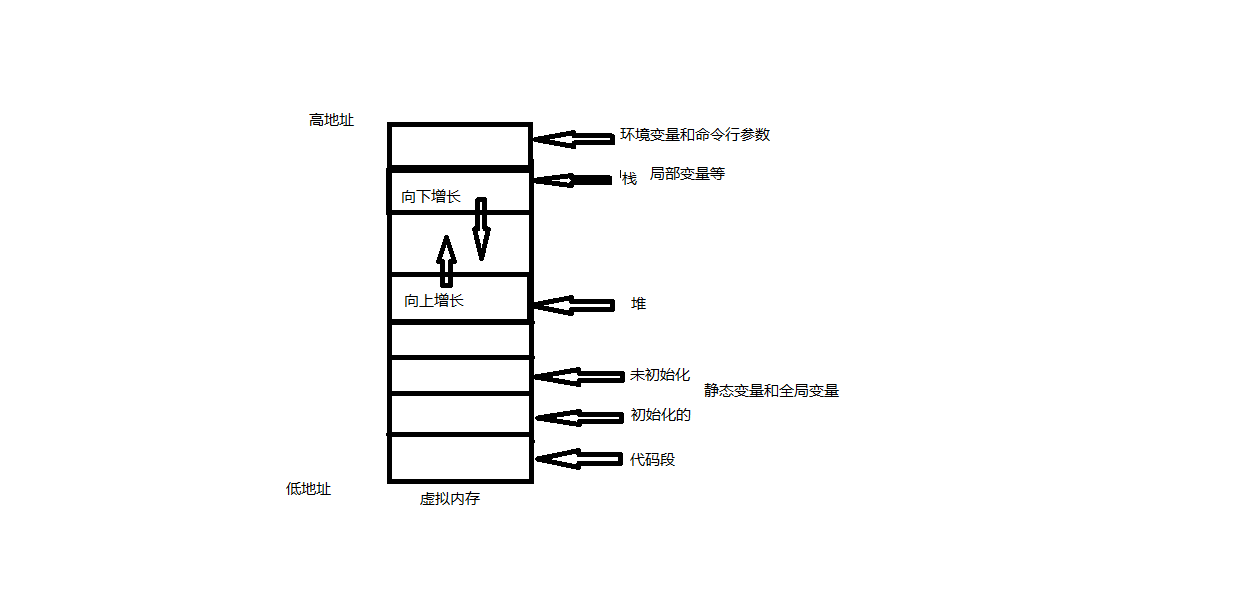
}

0x7fff988578a8 0x7fff988578a4 0x7fff98857880 0x7fff9885787c 0x7fff98857860 0x7fff9885786c

1. 静态全局变量不能跨文件，被static修饰了以后，被extern修饰也没用，除非在另外一个文件中定义一个同名非static全局变量。
2. fputs用于文本模式，windows下你写入的字符串里面会有\n，fputs会自动在\n前插入\r。
3. 函数跨文件执行，两个文件中都定义了静态变量，函数会对静态变量操作，我想知道函数会改变的是定义文件还是执行文件？肯定是定义文件中的，因为可以假定使用静态函数变量。
4. 如果要对RELOP用到sizeof，需要在extern里面传入具体的数字大小，我还以为链接能够解决这个问题。

extern char \*RELOP[6];

1. 因为构造函数没有返回值，所以Huffuman 的bool buildTree(string fInput)不适合和构造函数合并。我写的是UI程序，又不能直接退出程序，只能通过判断返回值然后提醒用户。没有问题的，就当构造函数执行失败了，一个弹框或者按钮边上的文字提醒就行。
   1. 693.一个由C/C++编译的程序占用的内存分为以下几个部分   
      1、栈区（stack）— 由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其   
      操作方式类似于数据结构中的栈。   
      2、堆区（heap） — 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回   
      收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表，呵呵。   
      3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的   
      全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另   
      一块区域。 - 程序结束后由系统释放。   
      4、文字常量区 —常量字符串就是放在这里的。 程序结束后由系统释放
2. 程序代码区—存放函数体的二进制代码。



1. 枚举排列：

C++ STL提供next\_permutation()（algorithm头文件），支持从最小字典序开始不断枚举下一个排列，从而枚举所有排列。

#include <iostream>

#include <algorithm>

using std::cin;

using std::cout; using std::endl;

int main(void) {

int n,p[10];

cin >> n;

for(int i1=0;i1<n;++i1) cin >> p[i1];

std::sort(p,p+n);

do {

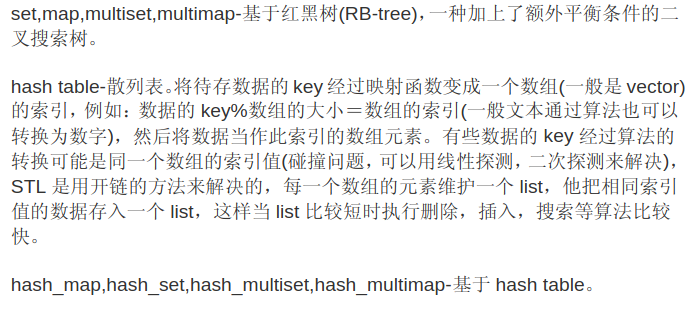
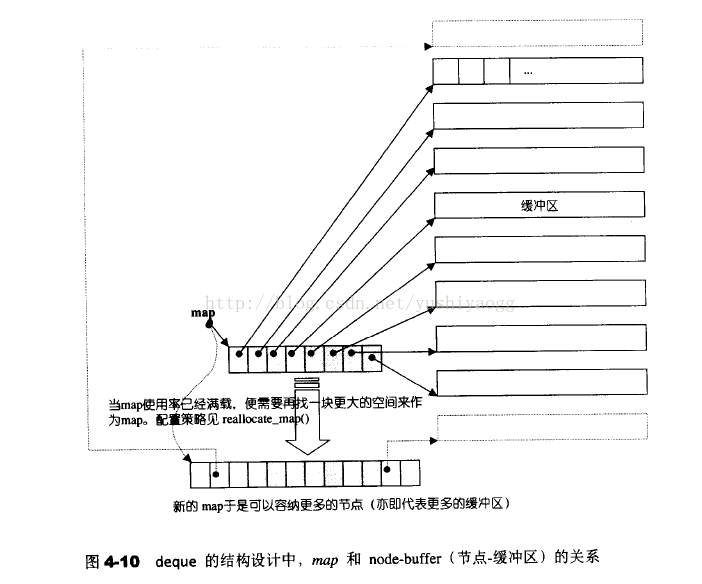
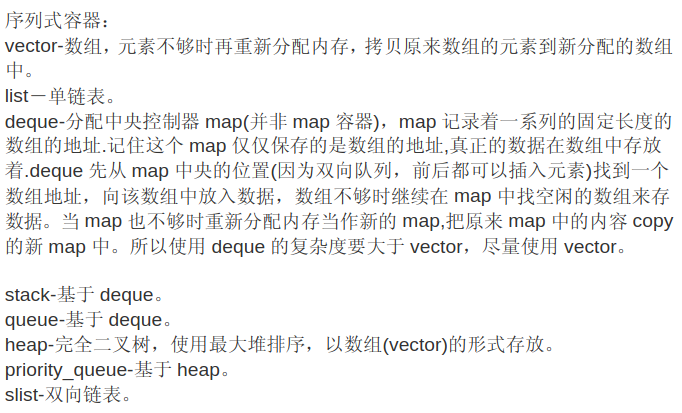
for(int i1=0;i1<n;++i1) cout << p[i1] << " ";

cout << endl;

}while(std::next\_permutation(p,p+n));

}

1. STL共有六大组件 ： 1、容器。2、算法。3、迭代器。4、仿函数。6、适配器。



1. 语言分为编译型还是解释型，看主流处理方式，因为C是存在解释器的。

699.typeid

cout << typeid(1.1f).name() << endl;

对非引用类型，typeid是在编译时期识别的，只有引用类型才会在运行时识别。 输出float指针的name()即”Pf”

701. const std::fpos（对应C语言文件的fpos\_t，表示一个流中的位置）

std::fpos也是个模板，要输入模型参数，比如说：std::fpos<std::string>。我这是把它拎出来使用看错误才知道的（不是在另一个代码文件中），不然错误提示一大堆。Defined in header <ios>

1. 引用本身通过指针实现。
2. 很明显不能对this指针赋值
3. 内联函数 补充：

使用inline的坏处：

(1)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。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。

(2)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，不像 non-inline 可以直接链接。

(3)是否内联，程序员不可控。内联函数只是对编译器的建议，是否对函数内联，决定权在于编译器。

虚函数可以是内联函数，内联是可以修饰虚函数的，但是当虚函数表现多态性的时候不能内联。

1. extern "C" （可以用于修饰函数与变量）的作用是让 C++ 编译器将 extern "C" 声明的代码当作 C 语言代码处理，可以避免 C++ 因符号修饰导致代码不能和C语言库中的符号进行链接的问题。后面可以接代码块或者表达式。

这特性需要编译器支持

1. 将流重定向到某一个文件

dup2（文件描述符）、freopen（文件路径）

1. malloc函数内部实现：

在Linux中，每个进程可访问的虚拟内存空间为3G，但在程序编译时，不可能也没必要为程序分配这么大的空间，只分配并不大的数据段空间，程序中动态分配的空间就是从这一块分配的。如果这块空间不够，malloc函数族（realloc，calloc等）就调用sbrk函数将数据段的尾部移动（brk、sbrk函数会改变数据段的长度），sbrk函数在内核的管理下将虚拟地址空间映射到内存，供malloc函数使用。注意，sbrk不是系统调用，是C库函数。

#include <unistd.h>

int brk(void \*addr);

void \*sbrk(intptr\_t increment);

sbrk传入0的时候，返回的指针就是当前数据段末尾终止后第一个位置

栈的内存分配速度比堆快，是因为

1. 操作只涉及到几条专门的指令 pop、push，以及栈操作设计寄存器
2. 堆如果内存不足的时候，还需要调用brk、sbrk函数扩大数据段
3. 顺序点：

1.每个完整表达式结束后，即分号后面   
2.&&，||，三木运算符（？：），以及逗号表达式的每一个运算对象计算之后   
3.函数调用中对所有实际参数的求值完成之后（进入函数体之前）