Static\_cast、dynamic\_cast、const\_cast和reinterpret\_cast总结：

static\_cast：用于非多态类型之间的转换，不提供运行时的检查来确保安全性，但可以在编译时期进行类型检查（这是和强转的区别）。用在以下情况：

\*类层次结构中，把子类指针或引用转化为父类表示，或父转子（不安全）

\*用于基本数据类型之间的转换，比如int-char，int-enum，自己担风险。

\*把void指针转化为目标类型指针，非常不安全

截取原理大约是，4bit转2bit，高位截取。2bit转4bit，高位补0

dynamic\_cast<type>(exp)：type必须是类的指针、类的引用或void\*。如果type是一个指针类型，exp必须也为指针类型。如果type是引用类型，exp也必须为引用类型。用于类层次间上下行转化，子转父和static\_cast一样，父转子有类型检查功能会更安全。

\*继承关系之间的指针类型与引用类型的转换。

\*如果exp是type的父类（父转子），若不能转返回nullptr；

const\_cast<type>(exp)：用来改变表达式里面的常量性(const)与易变形(volatile)

\*用于常量指针被转化为非常量指针，并任然指向原来的对象。

\*常量引用被转化为非常量引用，并且任然指向原来的对象。

Reinterpret\_cast<type>(exp)：用来处理无关类型之间的转换，它会产生一个新值，新值会与原始参数具有相同的比特位（搞不懂）

c++定义静态变量，要在外面实现，否则会无法解析外部符号。

**Assert、#error和static\_assert的区别：**

assert先计算表达式，如果为假则打印错误，并用abort终止程序运行，为动态断言。

static\_asert在编辑期间实现断言，对于调试模板很有用

#error在与预编译期

**Sizeof用法：**

\*计算栈中分配的内存大小

\*不计算static变量占用的内存

\*32位指针4字节，64位指针4字节，不管用的什么指针类型

**手动查看一个类中的内存分布：**

找到vs20xx开发人员命令提示符，切换到项目工作目录，cl /d1reportSingleClassLayout[className] [cppFile Name]

**Const和constexpr的区别**

Const:

\*定义常量和修饰变量，定义了之后就不能修改了。

\*修饰变量，表示该变量的值不能修改了

Const char \*p 表示指针p指向的内容不能修改

Char \* const p 把p声明为常量指针，它的地址不能修改

Constexpr基本包括const的用法：

例子：

template<int N>

class list{

public:

void Print() const

{

cout << N << endl;

}

};

constexpr int hh(int a){

return a \* a;

}

Main():{

Const int z=2;

List<hh(z)> b;

b.Print()---------->4

const成员函数：

如果创建了const对象，而成员函数不是const则不能调用：

class NewFoo{

public:

int x;

NewFoo(int v){

x = v;

}

int getX(){

return x;

}

};

const NewFoo f(100);

f.getX();//错误，无法调用

同名函数可以指定不同的const修饰：

Int Get(){}

Int& Get(){}

Const int& Get() const {}

这三种都可以成功，当然第三种是用在const对象身上的。

Stl其实都不是用来继承的，会引发许多问题，例如析构问题

lamdba表达式

结构：[闭包捕获变量](参数){函数体}

*** []：默认不捕获任何变量；***

*** [=]：默认以值捕获所有变量；***

*** [&]：默认以引用捕获所有变量；***

*** [x]：仅以值捕获x，其它变量不捕获；***

*** [&x]：仅以引用捕获x，其它变量不捕获；***

*** [=, &x]：默认以值捕获所有变量，但是x是例外，通过引用捕获；***

*** [&, x]：默认以引用捕获所有变量，但是x是例外，通过值捕获；***

*** [this]：通过引用捕获当前对象（其实是复制指针）；***

*** [\*this]：通过传值方式捕获当前对象；***

在上面的捕获方式中，注意最好不要使用[=]和[&]默认捕获所有变量。首先说默认引用捕获所有变量，你有很大可能会出现悬挂引用（Dangling references），因为引用捕获不会延长引用的变量的声明周期：

最好不用=或&来捕获全局变量，例如有个临时值类型变量，本来马上就可以销毁，但用了&之后修改变他的生命周期

Int a=100;

例子： auto lb1=[](){cout<<””’}不捕获

Auto lb2=[a](){cout<<a}捕获

Auto lb3=[&a](){a+=10;cout<<””;}引用捕获,修改变量后，源变量也会修改

Auto lb4=[a]()mutable{a+=10;cout<<a} 复制捕获，修改变量源变量不会修改

--按值类型捕获并可以修改

int x{0};

auto f1 = [=]() mutable {x = 42;}; // okay, 创建了一个函数类型的实例

Typename用法：

\*声明模板参数，这里class和typename作用一样

\*声明一个类型：

Ex:

Template<typename T>

Void Print(const T& con){

If(con.size()>=2){

C::const\_iterator it(con.begin());

++it;

Int value=\*it;

Cout<<value;

}

}

上面这个代码会报错，因为编译器不知道T::const\_iterator是一个类型，万一是个变量呢。所以需要对这个类型进行申明。

Typename T::const\_iterator it(con.begin())

\*

**模板学习：**

\*类模板和函数模板都要编写定义再。h文件中

\*模板的实例化类型确定是在编译期进行的。

\*模板写好，一般编译器不会出什么错，出错一般在实例化编译之后

\*用到的模板会编译，没用到的不会编译

\*函数模板只能全特化不能偏特化，类模板都可以。

\*模板的匹配优先匹配特化版本，且有限匹配全特化，然后是偏特化。按照参数的明确程度进行匹配。

\*函数模板可以参数推导，类模板不能。

函数模板：

\*成员函数的模板函数为啥不能为virtual？：

因为c++编译器在创建类的时候要确定虚表的大小，例如又n歌虚函数，按字节算虚表大小就是4n。但是由于成员模板函数，他本质上会编译出许多个变体用来接收不同类型的参数，所以虚表的大小就不能确定。

\*成员函数的模板函数为啥不能有函数模板的默认模板参数？：

打个问号？在我实验中是可以用的。

**\*模板全特化与偏特化：**

意思就是把泛化的东西变得特化一些，例如没有特化的可以对任何类型做相同处理，特化后分别对不同的类型分情况处理。

特化分为全特化与偏特化，全特化就是模板参数全部确定，不需要编译时期确定。偏特化是参数部分确定，还有部分需要编译时期确定。

\*函数模板的特化（全特化），如下，参数全部为确定类型：

template<typename T>

bool eq(T a, T b)

{

return a == b;

}

template<>

bool eq(float a,float b)

{

return abs(a - b) <= 0.0001;

}

\*类模板函数的特化（全特化），如下，参数全部为确定类型：

template<typename T>

class Fclass

{

public:

bool eq(T a, T b);

};

template<typename T>

bool Fclass<T>::eq(T a, T b)

{

return a == b;

}

template<>

class Fclass<float>

{

public:

bool eq(float a, float b);

};

bool Fclass<float>::eq(float a, float b)

{

return abs(a-b)<=0.001;

}

\*模板的参数匹配优先级：

模板的匹配优先匹配特化版本，且有限匹配全特化，然后是偏特化。按照参数的明确程度进行匹配。

template<typename T,typename U=int>

class fzy

{

public:

fzy()

{

cout << "morencanshu" << endl;

}

};

template<typename T>

class fzy<T,int>

{

public:

fzy()

{

cout << "偏特化版本" << endl;

}

};

template<>

class fzy<int,int>

{

public:

fzy()

{

cout << "quantehua" << endl;

}

};

会发现上面总是会优先匹配全特化的版本，如果没有全特化版本再匹配偏特化的，最后才匹配默认版本。

**\*类模板中**

\*每个不同类型的类模板中都有独立的静态成员

\*类模板中泛型成员函数：

template<class T>

class fzy

{

public:

template<class U>

void haha(U aa);

};

template<class T>//注意在外边有两层模板，不能卸载一起

template<class U>

void fzy<T>::haha(U u)

{

cout << u << endl;

}

**\*模板的模板参数**

\*typedef和using都可以给类型定义别名，但区别在于using专门用来模板的别名。

template<typename T>

using VV = std::vector<T, std::allocator<Test>>;

typedef double haha;

\*实践：模板的参数模板：

//下面例子建立给类来接收vector泛型，需要用到模板别名

template<typename T>

using VV = std::vector<T, std::allocator<Test>>;

//声明类：

template<typename T,

template<typename ELEM> class CONT> //ok

class ContainerWrapper {

public:

CONT<T> elems;

};

main---------：

ContainerWrapper<int, VV>zz;

zz.elems.push\_back(100);

zz.elems.push\_back(200);

for (auto c : zz.elems)

cout << c << endl;

\*举例：不是模板模板参数：

template<typename T, typename Sequence = list<T>>

class stack

{

private:

Sequence c;

};

Main-----

Stack<int> aa1;

Stack<int,vector<int>>aa2;

上面这种情况就不是模板模板参数，因为模板模板参数的第一个参数和第二个参数是独立的。但是上面这种情况一但第一个参数确定了第二个参数也就确定了，公用一个T。

**\*模板的非类型参数：**

\*浮点数、字符串、类对象都不能作为非类型模板的参数

template<int NN>

int feileixing(int a){

return a + NN;

}

如上就没啥问题，但如果改为float、double、string、类对象就不可以。浮点数有历史原因，string是内部链接对象不可以。

**\*模板类与友元：**

\*可变参数模板：

//没有多余参数的版本：

// 用来终止递归并处理包中最后一个元素

template<typename T>

void FPrint(string str, T t)

{

cout<<str<<endl;

cout << "wuhu" << endl;

}

template<typename T,class... V>

void FPrint(string str,const T& firstArgs, const V& ... args)

{

cout << sizeof...(args)<<" "<<firstArgs << endl;

FPrint(str,args...);

}

**内部链接和外部链接的区别：**

链接类型会直接影响我如何将一个给定的逻辑结果合并到物理设计中。

链接一般分为两部：1对各个编译单元生成根文件cpp->.o,，2通过链接程序将各种符号链接起来，生成可执行文件。

**\*内部链接：**

定义：

意味着这个对象的访问受到当前编译单元的限制。也就是一个有内部链接的定义对于任何其他任何编译单元都是不可见的。例如：

Static int x; enum haha{NO,YES}

具体类型有：

静态全局变量的定义，静态自由函数的定义，静态友元函数的定义

类的定义

内联函数的定义

Union共同体的定义

Const常量定义

枚举类型定义

所有的声明

用内部链接定义的一个重要例子就是类的定义。他不能在同一作用域的编译单元内重复定义。在单个编译单元外部使用的类必须在一个头文件中定义。

class Point{

int d\_x;

int d\_y;

public:

Point(int x,int y):d\_x(x),d\_y(y){} //internal linkage

int x() const{return d\_x;} //internal linkage

int y() const{return d\_y;} //internal linkage

};

inline int ..() {}; //internal linkage

**\*外部链接：**

定义：

意味着这个定义不局限于单个的编译单元。在.o文件中，具有外部链接的定义产生外部符号，这些外部符号可以被所有其他单元访问，来解析其他单元中的未定义符号。

具体类型有：

类的非内联函数（包括成员函数和静态类成员函数）的定义

类的静态成员变量的定义

命名空间或全局非静态自由函数，非静态变量，非友元函数.

所以拥有外部链接的实体如果被声明在头文件并且多个cpp包含，就可以能有链接冲突错误，有多个编译单元链接时，面对多个相同名字，无法正常链接到对象。

**\*最后再给出一个C++编程建议，慎重考虑在头文件中定义有链接的实体**

一，如果头文件是像int a=1;这样的定义，被包含在多个.cpp文件后肯定会报出链接错误。

二，如果是想static int a = 2;这样的定义就会在所有包含他的.cpp文件中生成一个副本，如果被大量源文件include的话，就会占据大量的空间，造成内存浪费。

**函数指针模板参数：**

template<typename T,void (\*FP)(T& v)>

void fForeach(T arr[], int size){

for (int i = 0;i < size;i++)

FP(arr[i]);

}

template<typename T>

void Print(T& v){

cout << v << " ";

}

//main

int a[5] = { 1,2,3,4,5 };

fForeach<int, Print>(a, 5);

**指针数组和数组指针详解：**

数组指针：可以说成是“指针的数组”，首先这个变量是一个数组，数组中的每一个元素都是指针类型，在32位中每个元素占4字节，64位每个元素占8字节。可以作为函数参数传递

char\* arr[4] = { "asdfas","fasdfa","asdfasdf","dfadf" };

int \*p1;int \*p2;int\* p3; int\* arr2[3] = { p1,p2,p3 };

指针数组：可以说成是“数组的指针”，首先这个变量是一个指针，指针指向数组的首地址，一般用作参数传递。

void printRes(int(\*rs)) //传递一个一维数组

{

for (int i = 0;i < 3;i++)

{

cout << rs[i];

}

}

void printRes(int(\*rs)[2])//传递一个二维数组

{

for (int i = 0;i < 3;i++)

{

cout << rs[i];

}

}

引用数组和数组引用详解：

引用数组：好像没有这种东西。！！！！！！！！

数组引用：一般用作参数传递,可以声明为变量。一般配合模板使用

作为参数传递：

Void newTest(int (\*arr)[4]){} //表示arr是一个数组，类型位int&类型，个数为四个。

int arr1[4] = { 1,2,3,4 };

printRes(arr1);

作为变量使用：

int array[2] = { 1, 2 };

int(&a)[2] = array;

Std::boolalpha和std::noboolalpha:

用于输入输出流。用于cout的一开头：

True a=1,b=1,c=0;

Cout<<std::boolalpha<<a<<b<<c<<endl;

Res: true,true,false

如果不加的话默认是用1和0来输出的。

c++占位符：

Using namespace std::placholders; 内置有\_1,\_2....\_20占位符，多用于bind函数使用，填充bind的参数。

C++bind函数：

返回一个函数对象，

Char\* s,char s[4],char s[]长的计算：

Strlen不统计\0这个字符的长度,下面答案刮开查看

const char\* s1 = "adsfasf";

cout << s1 << endl;

cout << strlen(s1) << endl;

cout << sizeof(s1) << endl;

结果：7，4

char s2[4] = { '1','2','3','\0' };

cout << s2 << endl;

cout << strlen(s2) << endl;

cout << sizeof(s2) << endl;

结果：3，4

char s3[] = { "adsfasf" };

cout << s3 << endl;

cout << strlen(s3)<<endl;

cout << sizeof(s3) << endl;

结果：7，8

**Move移动语义：**

移动语义功能上和拷贝相似，但效率要比拷贝搞得多了。

Move得主要功能有：以极高得效率实现复制、转移所有权。

Ex：

String a=”lll”;strign b=”zzzz”;

A=std::move(b);

//这时候b的所有权已经转移到a了。b已经可能没有值了，但还可以用它，但用它的可能得不到什么太大用了。

**智能指针：**

unique\_ptr：

他不能被多个实例共享的内存管理，仅有一个实例拥有内存所有权。

申明：

Unique\_ptr<type> f1(new Test(1,2));

Auto f2=make\_unique<type>(1,2);

如果有unique\_ptr<type>f3; f3=f1这是不允许的 因为unique\_Ptr返回的是个右值。正确的需要需要使用move来操作:f3=std::move(f1)。这时候f3有值，但f1就没有了，因为一个仅有一个实例有内存所有权。

shared\_ptr：

原理是引用计数，只有最后一个对象指向该内存的shared\_ptr不需要再指向该对象时，才会析构这个对象。计数的作用是每增加一个shared\_ptr count+1，到count为0时就会释放内存。

shared\_ptr指针常用操作

**use\_count()：**返回多少个智能指针指向某个对象，主要用于调试目的

**unique()：**是否该智能指针独占某个指向的对象，如果是，返回True

**reset()：**复位/重置

不带参数时

若该指针是唯一指向该对象的指针，那么释放该对象，并将该指针置空(nullptr)

若该指针不是唯一指向该对象的指针，那么不释放该对象，但计数减一，该指针置空(nullptr)

带参数时(一般是new出来的指针)

若该指针是唯一指向该对象的指针，那么释放该对象，并将该指针指向新对象

若该指针不是唯一指向该对象的指针，那么不释放该对象，但计数减一，该指针指向新对象

**\*解应用**：获得指针指向的对象

**get()：**返回保存的指针(裸指针)。用于有些函数的参数需要的是一个内置裸指针而不是智能指针

**swap()：**交换两个智能指针所指向的对象

**=nullptr**

将所指向的对象引用计数减一，若引用计数变为0，则释放该指针指向的对象

将智能指针置空

指定删除器以及数组问题

可以指定自己的删除器函数取代系统的默认删除器

管理动态数组时

weak\_ptr的引出是为了解决循环引用的问题

**例如：**

Class A{

Shared\_ptr<B> \_b;}

Class B{

Shared\_ptr<A> \_a;}

Auto b=make\_shared<B>();

Auto a=make\_shared<A>();

b->\_a=a;

a->\_b=b;

这时候就会引起循环引用问题。解决办法很简单，将A类或B类的对象随便一个的shared\_ptr改为weak\_Ptr就可以了。

但是：

Weak指针不持有对象，不能通过weak指针去访问对象成员。

Lock（）返回一个shared\_Ptr指针对象，如果已经被析构返回nullptr；

use\_Count（）判断引用计数是否为0

Expired（）判断对象是否被析构

可变参数：

void PrintFloats (int n, ...){

double val;

printf ("Printing floats:");

va\_list vl;

va\_start(vl,n);

for (int i=0;i<n;i++)

{

val=va\_arg(vl,double);

printf (" [%.2f]",val);

}

va\_end(vl);

printf ("\n");

}

**c++函数指针：**

声明：

int(\*haha)(); 返回值（\*名称）（参数，参数）

int(\*hehe)(int);

调用：

cout << (\*haha)() << endl;

cout << (\*hehe)(1000) << endl;

**Delete和default的使用**：

Default用来指定默认的构造函数

Delete用来让已存在默认的默认的构造函数不能使用（例如拷贝构造函数）

Ex：

Class cv{

Cv(const cv&)=delete;

Cv& operator=(const cv&)=delete;}

Cv cc;

Cv ccc;

Cc=ccc;//会报错，不可以这样

Cv cccc=cv(cc)//不可以，拷贝构造函数被删除

让移动语义move函数生效可以：

Class cv{

Cv(const cv &&);

Cv& operator =(const cv&&);

Cv cc;

Cv ccc;

Cc=std:move(ccc);

Cv cccc=cv(move(ccc);

拷贝构造函数的调用时机：

\*把对象以值传递的方式传入函数参数：

Gl\_test(cv cc){

Cout<<”111”<<endl;}

\*当把对象以值传递得方式从函数返回

Cv Gl\_Test(){

Cv cc;

Return cc;

}

\*一个对象赋值给另一个对象：

Cv cc;

Cv xx(cc); //或 xx=cc;

Floor，ceil，round的区别：

Ceil向上舍入：1.26-》2 2.1-》3

Floor向下舍入：1.26-》1 2.1-》2

Round：四舍五入：1.4-》1 1.5-》2



左值右值，左值引用右值引用，引用的概念：

**左值右值：**

左值一定在内存中，右值可能在内存中也有可能在寄存器中（也可以说左值是可赋值的，有地址的）

&&表示右值引用。右值引用可以绑定到右值（但不能绑定到左值！）：

Ex:

int &&f()

{

return 100;

}

int i = 100;

int & x1 = i;//正确，可以讲x1绑定到i（一个左值上）

int & x2 = f();//错误，f()返回右值，不能绑定

int &&x3 = f();//正确，可以将x3绑定到f()上（右值引用）

int &&x4 = i;//错误，无法将x4绑定到左值上。

int a=5 a在内存中是左值，5是右值，在寄存器中

Int b=a b在内存中是左值，a也在内存中是右值

---------------------

int b=a+1 此时a+1是右值，在寄存器中

Int \*p=&a 此时&a是右值，在寄存器中

**引用：**

Int xx=100;

Int &a=xx;

Cout<<a<<” “<<&a<<” “&xx<<endl;

a和xx的地址是一样的，说明引用其实可以理解为对指针的封装吧。

左值引用：

**右值引用：**

一般只用于参数，设计时考虑我真的需要右值引用或不关心说明类型只是个没存在感的中间层用右值引用。

为c++11新特性，实现语义转移和精确传递。为了消除两个对象交互时不必要的对象拷贝，节省运算存储资源提高效率；更简洁明确的定义泛型。

广义可分为三种右值引用：

\*无名右值引用：

---将表达式t转换为T类型的无名右值引用。

int aaa = 1;

int bbb = 2;

int &x1 = aaa;

int &&x2 = static\_cast<int&&>(aaa);

int &&x3 = static\_cast<int&&>(x1);

---使用move来实现，标准库中的类型基本都实现了移动构造和移动赋值，都可以用move。

\*具名右值引用：

初始化时候设置

int&& b = static\_cast<int&&>(a);

int&& c = std::move(a);

//static\_cast<X&&>(a) 和 std::move(a) 是无名右值引用，是右值

//b 和 c 是具名右值引用，是左值

\*转发型引用：

如果一个变量被声明为T&&类型，而且T需要经过推导才能确定，那么这个变量或参数就是一个转发型引用。

转发型引用由以下两种语法形式产生：

\*如果一个变量为auto&&，那么就是

\*函数模板中，参数为T&&，并且T需要推导才能确定类型，那么是。

特点：

\*转发型引用不稳定，他的实际类型由它所绑定的值来确定

\*转发型引用既可以绑定左值，又可以绑定右值。

\*如果绑定左值，就成了左值引用。如果绑定右值，成了右值引用。

Ex：

struct X {};

X&& var1 = X(); // var1是右值引用，只能绑定右值X() auto&& var2 = var1; // var2是转发型引用，可以绑定左值var1

// var2的实际类型等同于左值var1，即X& auto&& var3 = X(); // var3是转发型引用，可以绑定右值X()

// var3的实际类型等同于右值X()，即X&& template<typename T> void g(std::vector<typename T>&& param1); // param1是右值引用 template<typename T> void f(T&& param2); // param2是转发型引用

X a; f(a); // 模板函数f()的形参param2是转发型引用，可以绑定左值a

// 在此次调用中模板参数T将被推导为引用类型X&

// 而形参param2的实际类型将等同于左值a，即X& f(X()); // 模板函数f()的形参param2是转发型引用，可以绑定右值X()

// 在此次调用中模板参数T将被推导为非引用类型X

// 而形参param2的实际类型将等同于右值X()，即X&&

// 更多右值引用和转发型引用 const auto&& var4 = 10; // 右值引用 template<typename T> void h(const T&& param1); // 右值引用 template <typename T/\*, class Allocator = allocator\*/> class vector { public:

void push\_back( T&& t ); // 右值引用

template <typename Args...>

void emplace\_back( Args&&... args ); // 转发型引用 };

Type &&a =10。与左值区别在于：右值是临时变量切不变，如函数返回值。右值引用可以理解为右值的引用，右值初始化后临时变量消失。

Int xx=100;

Int &&a=xx; //不能将右值引用绑定到左值上

Int &&a=std::move(xx) //通过移动语义来实现

----------------------------

Void foo4(int && a){

Int x=a; cout<<x+100<<endl;}

Foo4(100)//只能传入右值哦，因为参数时右值引用类型。

**\*移动赋值：**

背后的思想就是，赋值不一定要通过拷贝来做，可以通过把源对象简单的偷换给目标对象，就是交换所有权。

基础数据类型和POD类型的移动和拷贝没什么区别。

使用move实现完美swap：

template<typename T>

void nSwap(T& a, T& b)

{

T tmp = move(a);

a = move(b);

b = move(tmp);

}

移动语义：

无名引用右值和具名引用右值的引入为了解决移动语义的问题，接管一个对象所有权。

移动语义通常步骤如下：

\*如果this对象自身也拥有资源，释放该资源

\*将this对象的指针或句柄指向that对象所拥有的资源

\*将that对象原本指向该资源的指针或句柄设为空

总结：

1.释放this（this不为空的时候）2.移动that

移动构造函数和移动赋值函数两者区别在于。移动构造对象时this对象为空，不需要释放this。

拷贝函数通常步骤：

\*如果this对象自身拥有资源，释放该资源

\*拷贝（赋值）that对象所拥有的资源

\*将this对象的指针或句柄指向新生资源

\*如果that对象为临时对象（右值），那么拷贝完之后that对象所拥有的资源也会释放。

总结：

1.释放this（this非空时），2.拷贝that，3.释放that（that为右值时）。

拷贝构造和拷贝赋值的区别在于。移动构造对象时this为空，不需要释放this。

Ex：

class MemoryBlock { public:

// 构造器（初始化资源）

explicit MemoryBlock(size\_t length)

: \_length(length)

, \_data(new int[length]) {

}

// 析构器（释放资源）

~MemoryBlock() {

if (\_data != nullptr)

delete[] \_data;

}

// 拷贝构造器（实现拷贝语义：拷贝that）

MemoryBlock(const MemoryBlock& that)

// 拷贝that对象所拥有的资源

: \_length(that.\_length)

, \_data(new int[that.\_length]) {

std::copy(that.\_data, that.\_data + \_length, \_data);

}

// 拷贝赋值运算符（实现拷贝语义：释放this ＋ 拷贝that）

MemoryBlock& operator=(const MemoryBlock& that) {

if (this != &that)

{

// 释放自身的资源

delete[] \_data;

// 拷贝that对象所拥有的资源

\_length = that.\_length;

\_data = new int[\_length];

std::copy(that.\_data, that.\_data + \_length, \_data);

}

return \*this;

}

// 移动构造器（实现移动语义：移动that）

MemoryBlock(MemoryBlock&& that)

// 将自身的资源指针指向that对象所拥有的资源

: \_length(that.\_length)

, \_data(that.\_data) {

// 将that对象原本指向该资源的指针设为空值

that.\_data = nullptr;

that.\_length = 0;

}

// 移动赋值运算符（实现移动语义：释放this ＋ 移动that）

MemoryBlock& operator=(MemoryBlock&& that) {

if (this != &that)

{

// 释放自身的资源

delete[] \_data;

// 将自身的资源指针指向that对象所拥有的资源

\_data = that.\_data;

\_length = that.\_length;

// 将that对象原本指向该资源的指针设为空值

that.\_data = nullptr;

that.\_length = 0;

}

return \*this;

} private:

size\_t \_length; // 资源的长度

int\* \_data; // 指向资源的指针，代表资源本身 };

MemoryBlock f() { return MemoryBlock(50); }

int main() {

MemoryBlock a = f(); // 调用移动构造器，移动语义 ，将新对象赋值，会调用移动赋值函数。

MemoryBlock b = a; // 调用拷贝构造器，拷贝语义

MemoryBlock c = std::move(a); // 调用移动构造器，移动语义

a = f(); // 调用移动赋值运算符，移动语义，将新对象赋值，会调用移动赋值函数。

b = a; // 调用拷贝赋值运算符，拷贝语义

c = std::move(a); // 调用移动赋值运算符，移动语义 }

**Explicit用法：**

首先，该关键字只能用于修饰一个参数的类构造函数（准确点是一个非默认参数的构造函数），后面有默认参数，也只看钱前面非默认参数的个数。表明该构造函数是显示的，而非隐式的，和implicit对应。构造函数默认是implicit的。

先看不加explicit的例子：

class Coo1{

public:

Coo1(int n){

\_size = n;

pStr = (char\*)malloc(sizeof(char)\*(n+1));

memset(pStr, 0, n + 1);

}

Coo1(const char \*p){

int len = strlen(p);

pStr = (char\*)malloc(sizeof(char)\*(len + 1));

strcpy(pStr, p);

\_size = len;

}

private:

int \_size;

char\* pStr;

};

Coo1 co1 = 10;//ok，因为隐式转换了，调用相当于于co1=Coo1(10)

Coo1 co2 = "123";//ok因为隐士转化了，相当于调用co2=Coo1（“123”）

Coo1 co3(10);//没毛病

Coo1 co4("123");//没毛病

Coo1 co5;//不行，没有默认构造函数

co1 = 20;//ok，相当于第一条

co2 = "456";//ok，相当于第二条

再看加如果加explicit关键字后：

就声明为显式，防止编译器隐式转换。

Coo1 co1 = 10;//错误，没有合适的构造函数

Coo1 co2 = "123";//错误

Coo1 co3(10);//没毛病

Coo1 co4("123");//没毛病

Coo1 co5;//不行，没有默认构造函数

co1 = 20;//错误，相当于第一条

co2 = "456";//错误，相当于第二条

颜色相加与相乘的意义：

加法一般是表示光的强度相机，用于不同光源计算结果来相加

乘法一般是表示颜色非等比缩放，例如光通过玻璃，玻璃有不同的颜色吸收率，漫反射可以理解为材质吸收了某百分比的光然后进行反射。

如果考虑HDR光的颜色可以大于1，但吸收率还是0-1

C++随机数类：

Default\_random\_engine e();//参数可以设置随机数种子

Uniform\_int\_instribution<int> u(0,1)//生成范围为【0-1】范围内的随机数，均匀随机数模板

Normal\_distribution<float> u(-1,1)//生成正太分布随机数。

Cout<<u(e);//有好多种随机数类型

Ifstream、ofstream、fstream的基本用法。

文件流的本质是利用了一个buffer中间层。

Ifstream----从已有文件读

Ofstream---向文件写入内容

Fstream---打开文件供读写

实际上，文件类型分为两种：文本文件和二进制文件。

文本文件保存的是可读字符，二进制文件保存的都是二进制数据。利用二进制模式可以操作图像等文件。用文本模式只能读写文本文件。否则报错。

**写入：**

Ofstream file;

File.open(“test.txt”);

File<<”1111”<<75;

File.close();

**读取：**

Ifstream file;

Char output[100];

File>>output;

File.close();

**读写：**

Fstream file

File.open(“x.txt”,iso::in|iso::out) //ios实在std下的

Xxxxxx

File.close();

Ios::in 读

Ios::out 写

Ios::app 从文件末尾开始

Ios::binary 二进制模式

Ios::nocreate 打开一个文件时，如果文件不存在，不创建文件

Ios::noreplace 打开文件不存在则创建

Ios::trunc 打开一个文件，清空内容

Ios::ate 打开文件，将位置移动到文件尾

默认是文本模式

如果文件不存在，默认创建新的

可多种操作混合，用|

文件的bute索引从0开始

istringstream和ostringstream

Ostringstream：可以用来格式化字符串

For(---)

Ostringstream<<a[i]<<””

Return ostringstream.str();

Istringstream:可以用来对流中的字符串进行操作

String str=”i love you”

Istringstream iss(str)

String s;

While(iss>>s)

Cout<<s<<endl;

输出结果为“i换行love换行you换行”

位域‘：’：

可以显示一个数据类型的所占的位数

限定：

\*位域采用压缩存储，能用1bit就不用2bit

\*会增加处理时间，访问位域成员要把位域从他所在的字节中分解出来或把值压缩存储到位域所在的字节中。

\*一般用于unsigned数据类型

\*未指定占用位默认占用所有位数

\*位域以数据类型来区分区域，有几种数据类型就表示有几个位域段

\*同类位域使用的bit超过该位域区域后继续占用该类型位域数据类型大小。

\*不要在两个位域字段之间穿插非位域字段

\*未用完的bit还是要占用内存空间，比如使用int类型的4位，后面28位还占用空间

\*遵循字节对齐原则

\*位域字段不能是静态成员

\*位域字段在内存中是按低位到高位顺序放的

\*不能用于&取地址符号。

语法：

Class Test{

Unsigned char a:4; //正常的char是一个字节，一个字节八位，现在限制为了4位 int b:8 //正常的int是4个字节，一个字节八位（共32位）限制限制为了八位只能表示小于2的8-1次方以下的数。

上面占用八字节，虽然a和b的位数都用完，但由于是两个不同的数据类型，所以位域也是两个不同的。

Class NewTest{

Unsigned char a:4

Uint32\_t b:10;

Uint32\_t c:10;}

上面占用还是八字节，因为，b和c位同一位域，两个加起来采用了20位，还有12位剩下呢。所以更具字节对齐只占用了8字节。

Decltype和auto的区别：

\*auto：根据明确的类型推导出一个元素的类型，auto变量必须有初始值。

---同一语句的类型要一致：

int i = 3;

auto a = i, &b = i, \*c = &i;//正确: a初始化为i的副本,b初始化为i的引用,c为i的指针.

auto sz = 0, pi = 3.14;//错误,两个变量的类型不一样

---auto会忽略引用类型与const类型：

int i = 100, &ii = i;

auto rs = ii;

会发现rs为int类型，可见会忽略引用类型。

int iii = 100;

int\* const a = &iii;

const int b = 100;

auto rs = a;

auto rs1 = b;

会发现rs，rs1都为int类型，可见会自动忽略引用类型。

如果使用的话，要明确加上&或者const。

\*decltype：可以推导出表达式或变量的类型

--不能使用lamdba进行推到。

--不会执行表达式或函数来推导类型：

Decltype(f()) sum =x; //实际不会调用f，而是分析f的返回值。

--decltype就不会忽略const和引用：

const int ci = 42, &cj = ci;

decltype(ci) x = 0; // x 类型为const int

auto z = ci; // z 类型为int

decltype(cj) y = x; // y 类型为const int&

auto h = cj; // h 类型为int

int ii = 100;

int \* const iii = &ii;

decltype(iii) zx =&ii;

--decltype和auto还有一处重要的区别是，decltype的结果类型与表达形式密切相关。有一种情况需要特别注意：对于decltype 所用表达式来说，如果变量名加上一对括号，则得到的类型与不加上括号的时候可能不同。如果decltype使用的是一个不加括号的变量，那么得到的结果就是这个变量的类型。但是如果给这个变量加上一个或多层括号，那么编译器会把这个变量当作一个表达式看待，变量是一个可以作为左值的特殊表达式，所以这样的decltype就会返回引用类型：

int i = 42;

//decltype(i) int 类型

//decltype((i)) int& 类型

--decltype与auto结合的后置类型：可以更明确的展示函数返回值

auto he(int x)->decltype(x)

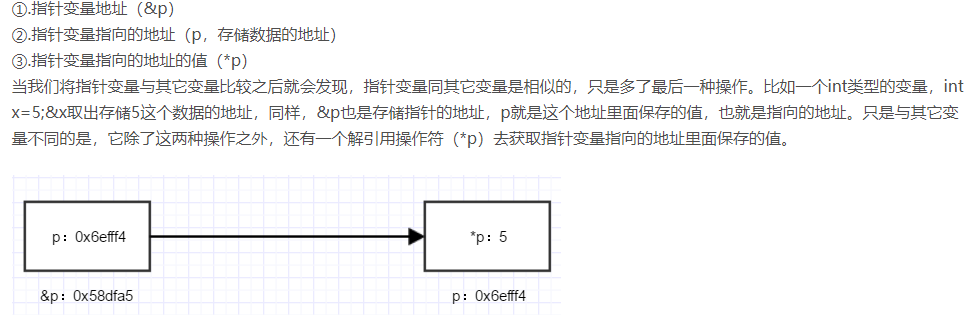
{

return x+100;

}

使用指针做参数的区别（一级指针和二级指针）：

一级指针：



void Set(int \*p)

{

int a = 100;

p = &a;

}

Int aa=200;

Int\* xx= &aa;

Set(xx)//

结果发现修改失败，因为不加引用的话都是用值传递，所谓值传递就是传递指针的值，传递进入的是副本，终究修改的还是副本。所以要修改的时候，要传递二级指针。

起初xx的指向a这个变量，传参其实传入了一个指针副本，这个副本目前同样指向aa，然后修改这个副本指针p的指向，指向临时变量a（ok没毛病）。但是没有把这个指针返回，所以最终实际的xx还是指向aa并没有被修改。两个指针分道扬镳。

函数的参数进行值拷贝,即使传的是指针,也的对指针(即指针里存的地址)的拷贝，可不是拷贝指针里地址所指的值

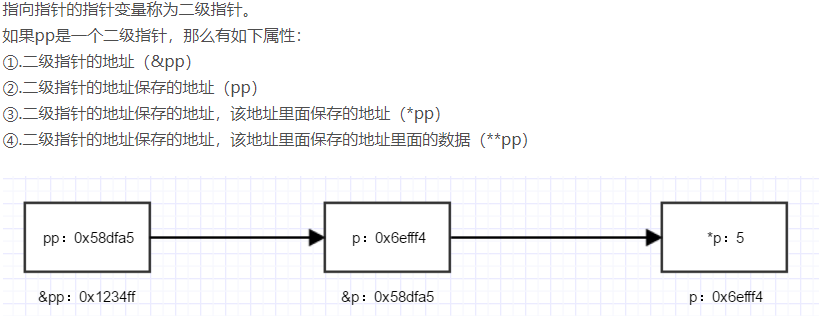
总结：

函数的形参都是原数据的“副本”，因此在函数内无法改变原数据

函数中参数都是传值，传指针本质上也是传值

如果想要改变入参内容，则需要传该入参的地址（指针和引用都是类似的作用），通过解引用修改其指向的内容

二级指针：



void NSet(int \*\*p)

{

int a = 100;

\*p=&a;

}

//二级指针，在传参时pp被复制出来一份，这是修改p指向的内容就可以达到修改的目的。但如果是一级指针的话，指针被复制出来，指针修改指针指向的内容，实际指针无法被修改。

引用类型返回值和非引用类型返回值：

\*一定不可以返回局部的引用类型！！！

\*

Std::iota:

顺序为区间内元素赋值: std::iota(a, a + 4, 100);cong100开始依次递增

For\_each():

遍历区间内元素，进行操作：for\_each(a, a + 4, [](int n) {cout << n <<".."<< endl;});

std::ref和std::reference\_wrapper的区别：

**Ref的作用：**让按值传参的模板可以接受一个引用作为参数。Ref是按照模板参数推导来实现的，如果创建了按值传递的非模板函数想传递引用，那么ref是不可以的

template<typename T>

void rrr(T x){

x += 200;

}

Main-----------:

Int x=100;

Rrr(x）//这样x的值并不会修改

Rrr(std::ref(x))//这样x的值会修改

**reference\_wrapper的作用：**

\*stl容器中提供的value语义而不是reference语义，所以容易不支持元素为引用，使用reference\_wrapper可以实现。

std::list<int> l(10);

std::iota(l.begin(), l.end(), -4);

std::vector<std::reference\_wrapper<int>> v(l.begin(), l.end());

// can't use shuffle on a list (requires random access), but can use it on a vector

std::shuffle(v.begin(), v.end(), std::mt19937{ std::random\_device{}() });

std::cout << "Contents of the list: ";

for (int n : l) std::cout << n << ' '; std::cout << '\n';

std::cout << "Contents of the list, as seen through a shuffled vector: ";

for (int i : v) std::cout << i << ' '; std::cout << '\n';

std::cout << "Doubling the values in the initial list...\n";

for (int& i : l) {

i \*= 2;

}

std::cout << "Contents of the list, as seen through a shuffled vector: ";

for (int i : v) std::cout << i << ' '; std::cout << '\n';

结果发现：list和vector中的都被修改了。

Struct和union的区别：

**\*在**存储多个成员信息时，编译器会给每个struct成员都分配内存，而union中的元素只会位所有元素分配一块内存（即元素中最大的字节那个元素为分配的内存空间）。

**\*struct**中的元素时刻都在，但是union中最后只有第一个使用的元素。

例如：

Struct Test\_1{ int a;char b;}

Union Test{int a；char b；}

Test tt; tt.a=65;tt.b=’b’; 这样最后a就成了97（输出位b的ascii码，在不超过char数据范围的情况下，如果a设置位1000，那么就会出现奇怪的数），b还是输出‘b'’。

上面中，Test\_1占用8字节（因为内存对齐），Test占用4字节，因为只使用int所占的字节数，因为元素中int最大。

**\*union**中的元素后面的会对前面的进行覆盖，struct不会。

#pragma pack(show)用来查看当前内存对齐的值，在警告界面输出。

内存对齐！！！：

\*内存对齐的对齐数默认为结构体中所占最大的数据类型！！！。

到底哪个是对的？： \_\_declspec(align(x)) #pragma pack（），前者规定对齐的最小值，后者规定最大值。？？？？？

例如：

Struct Test{

Int a;double b;char c;} 总共要24字节，因为默认以double（8字节对齐）。

如果加上#pragma pack（4），设置为使用4字节对齐，就需要16个字节。

\*使用\_declspec(align(x))的特点是，它仅仅规定了数据对齐的**位置**啊，而没有规定数据实际占用的内存长度，当指定的数据被放置到确定的位置后，之后的数据填充任然是按照#pragma pack的方式填充，这时候类/结构的实际大小和内存格局的规则是这样的：

在\_declspec(align(x))之前，数据按照#pragma pack规定的方式填充，

遇到\_declspec(align(x))时，首先寻找距离当前偏移量最近的对齐点（满足对齐长度为max（数据长度自身，指定值x）），

然后把被指定的数据类型从这个点开始填充，其后的数据类型从它后面开始，任然按照#pragma pack填充，直到遇到下一个\_declspec(align(x))。

当所有数据填充完毕，把结构的整体数值和\_declspec(align(x))规定的值做比较，取其中较大的作为整个结构的对齐长度。

特别的：当\_declspec(align(x))指定的数值比对应类型长度小的时候，这个指定不起作用。

设置在结构体外面的\_declspec(align(x))只有结构体中没有重新设置过\_declspec(align(x))才起作用，并且只会影响整个结构体的大小。|||设置在结构体里面的\_declspec(align(x))会影响下面成员的地址起始位置，整个结构体的大小由最后一次设置决定。

例如：

\_declspec(align(4))

Struct A{

Char a;

Double b;

Int c;

Char d;

}

这个对齐的结果步骤为：

\*第一个char在最开头，不用考虑对齐，直接占据内存的第0字节

\*第二个double为8字节，于是min（4，8）=4，这时a后面的第一个地址是0+1=1，不是4的倍数，所以要把接下来的1，2，3字节空出来用于对齐，刚好是4的倍数，所以double从第4字节开始存放，从第四字节占据到了第11字节。

\*int为4字节，min（4，4）=4，第11字节后面是12字节，刚好是4的倍数，不需要多余字节用来对齐，直接在double后面存放，占据到第15字节。

\*char为1字节，min（4，1）=1，这时候15下一个字节为16，刚好是1的倍数，无需多余的对齐，直接放在double后面，占据到第16字节。

\*\*最后是整个结构体的对齐，对齐量为max（成员个数，其中成员的最大字节数），上面例子就是max（4，8（double的字节数））=8，结构体从0-16共占据了17个字节，不是8的倍数，所以要在后面补7个空白字节变成24，达到8的倍数。

POD类型：

plain old data：plain代表是一个不同类型，old代表可以于c兼容，可以用memcpy 等c中的原始函数进行操作。C++11中吧POD分为两个概念的集合：平凡的和标准部据的。

用途：

--可以安全的使用字节赋值，比如memset，memcpy等操作。

--对c内存布局兼容。

--保证了静态初始化的安全有效。

\*平凡的定义（trival）：

。拥有平凡的构造函数和析构函数，就是编译器默认为我么生成的，不允许自己定义，一旦自己定义，就是不平凡的。但有了default关键字，也可以修饰。

。拥有平凡的拷贝构造函数或移动移动构造函数。也可用default修饰。

。拥有平凡的拷贝赋值运算符和移动复制运算符号

。不能包含虚函数和虚基类

~~使用is\_trivial<T>::value可以来判断一个数据类型是否平凡。

\*标准部据的定义：

。所有静态成员都拥有相同的访问级别。

。类中第一个非静态类型与基类不是同一个类型

。没有虚类和虚基类

~~使用is\_standrad\_layout<T>::value来判断是不是标准布局类型

\*POD就是满足平凡和标准布局这两个方面，可以用is\_pod<T>::value来判断一个类型是 不是POD类型

Noexcept用法：

可以设置条件来决定是否抛出异常。

用法1：

Template<typename T>

Void Fun(T a,T b) noexcept(is\_pod<T>::value)

{a=b}

//main----

Cout<<noexcept(Fun(1,2))<<” “<<noexcept(fun(“aa”,”bb”))<<endl;

上面这种情况

内存泄漏：

\*类中有指针类型，在拷贝构造函数中没有注意处理。

Ex：

class HasPtrMem {

public:

HasPtrMem() : m\_data(new int(0)) {}

//HasPtrMem(HasPtrMem& h) : m\_data(new int(\*h.m\_data)) {} //拷贝构造函数,从堆中分配内存，用h.m\_data初始化

~HasPtrMem(){

if (m\_data != nullptr)

{

cout << m\_data << "..." << endl;

delete m\_data;

m\_data = nullptr;

}

}

int \*m\_data;};

//--main：

HasPtrMem a;

HasPtrMem b(a);

上面这种情况在最后连个对象析构时候就会出现问题，程序崩溃，因为使用系统默认的拷贝构造函数，只是简单的进行数值拷贝，所以a和b其实公用一个地址，两个指针变量指向同一个快内存空间，最后结束当b使用完先进行析构，然后a进行析构，可是指针变量m\_data已经被析构了，所以就会报错。所以在拷贝构造函数中要进行特殊处理，例如

c++对象模型学习

——类对象所占用的空间

\*空类占用一个字节空间

\*成员函数不占用类对象空间

\*成员变量占用对象内存空间

\*静态成员不占对象内存空间

\*虚函数会使对象增加4个字节来容纳虚表指针

——类中构造函数的调用顺序

例如有如下类：

class AA {

public:

AA() { cout << "AA" << endl; }

};

class BB {

public:

BB() { cout << "BB" << endl; }

};\*

class XX1 {

public:

AA a;

XX1() { cout << "XX1" << endl; }

};

class XX2:public XX1 {

public:

AA a;

BB b;

XX2() { cout << "XX2" << endl; }

};

class T :public XX2 {

public:

T() { cout << "T" << endl; }

};

结果:

AA

XX1

AA

BB

XX2

T

则他的调用顺序是先跳转到基类中，根据声明顺序执行对应的对象的构造函数与基类构造函数，然后跳到子类重复，直到完毕。

——虚函数表

AA a;

获取虚函数表首地址（也就是对象的地址，因为虚函数表是存放在一个对象开头的）：(int\*)(&a)

获取虚函数表中第一个函数的首地址：(int\*)\*(int\*)(&a)

调用执行虚函数表中的第一个函数：typedef void(\*Func)(); ((Func)\*((int\*)\*(int\*)(&a)))()

调用执行第二个：((Func)\*((int\*)\*(int\*)(&a)+1))()

——进程内存空间布局

有如下代码：以及输出结果

int \*ptest = new int(120); //堆

int g1; //bss 段

int g2;//bss 段

int g3 = 12; //数据段

int g4 = 32; //数据段

int g5;//bss 段

int g6 = 0;//bss 段

static int g7;//bss 段

static int g8 = 0; //bss 段

static int g9 = 10;//数据段

void mygfunc(){

return;

}

//定义一个类

class MYACLS{

public:

int m\_i;

static int m\_si; //声明不是定义 静态成员不占用具体对象的内存空间 存储在程序的数据段中

int m\_j;

static int m\_sj;

int m\_k;

static int m\_sk;

//static void myclsfunc() {}

};

int MYACLS::m\_sj = 0; //这才是定义； //数据段

int main()

{

int i = 7;;

printf("i地址=%p\n", &i);

//下面打印的是全局变量 这些全局变量原来是存放在可执行程序中的，多次加载到内存观察其内存位置都不变

printf("ptest地址=%p\n", &ptest);

printf("g1地址=%p\n", &g1);

printf("g2地址=%p\n", &g2);

printf("g3地址=%p\n", &g3);

printf("g4地址=%p\n", &g4);

printf("g5地址=%p\n", &g5);

printf("g6地址=%p\n", &g6);

printf("g7地址=%p\n", &g7);

printf("g8地址=%p\n", &g8);

printf("g9地址=%p\n", &g9);

printf("MYACLS::m\_sj地址=%p\n", &(MYACLS::m\_sj));

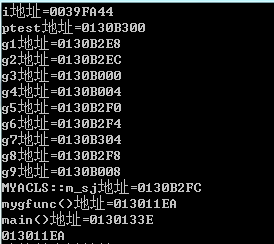
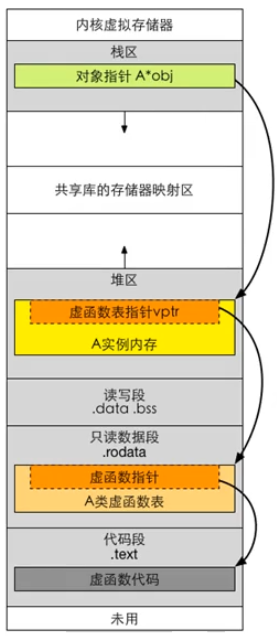
printf("mygfunc()地址=%p\n", mygfunc);

printf("main()地址=%p\n", main);

cout << (void\*)mygfunc << endl;

return 1;

}



#include <iostream>

using namespace std;

class NameStr

{

private:

char \*m\_pName;

char \*m\_pData;

public:

NameStr()

{

static const char s\_szDefaultName[]="Default name";

static const char s\_szDefaultStr[]="Default string";

strcpy(m\_pName,s\_szDefaultName);

strcpy(m\_pData,s\_szDefaultStr);

}

~NamedStr(){}

NameStr(const char\* pName,const char\* pData)

{

m\_pData=new char[strlen(pData)];

m\_pName=new char[strlen(pData)];

}

void Print()

{

cout<<"Name:"<<m\_pName<<endl;

cout<<"String:"<<m\_pData<<endl;

}

};

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

{

NameStr\* pDefNss=NULL;

pDefNss=new NameStr[10];

NameStr ns("hello","world");

delete pDefNss;

return 0;

}

找错误，一共大于等于5处:::

析构没释放指针

分配字符指针空间时要[sizeof(xx)+1]，给\0留一个

Delete要加[]，因为是对象数组

因为有指针，要写拷贝构造，拷贝赋值

规范代码：

#include <iostream>

using namespace std;

class NameStr

{

private:

char \*m\_pName;

char \*m\_pData;

public:

NameStr()

{

static const char s\_szDefaultName[]="Default name";

static const char s\_szDefaultStr[]="Default string";

strcpy(m\_pName,s\_szDefaultName);

strcpy(m\_pData,s\_szDefaultStr);

}

~NamedStr(){}

NameStr(const char\* pName,const char\* pData)

{

m\_pData=new char[strlen(pData)];

m\_pName=new char[strlen(pData)];

}

void Print()

{

cout<<"Name:"<<m\_pName<<endl;

cout<<"String:"<<m\_pData<<endl;

}

};

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

{

NameStr\* pDefNss=NULL;

pDefNss=new NameStr[10];

NameStr ns("hello","world");

delete pDefNss;

return 0;

}

Q(B.x-a.x)+W(c.x-a.x)=p.x-a.x 1

Q(b.y-a.y)+W(c.y-a.y)=p.y-a.y 2

Q=((p.y-a.x)-W(c.y-a.y)/(b.y-a.y)

1 (b.x-a.x)(((p.y-a.x)-W(c.y-a.y)/(b.y-a.y))+W(c.x-a.x)=p.x-a.x