# 类：

父类与子类有同名的数据成员

#include<iostream>

**using** **namespace** std;

**class** Base

{

**public**:

**int** a;

    Base(){a=0; }

**void** Print(){cout<<a<<endl; }

};

**class** Simple: **public** Base

{

**public**:

**int** a;

Simple(){a=1; }

};

**void** main()

{

    Simple b;

    b.Print();   //0

    cout<<b.a<<endl;     //1

}

new()子类对象时先调用父类构造函数，再调用子类构造函数。要指出调用哪个父类构造函数（不指出，则调用父类无参构造函数），必须：

Dog::Dog(const Dog& rrr):Mammal(4){……}

而不能:

Dog::Dog(const Dog& rrr)

{

Mammal:: Mammal(4);

……

}

后者执行实际结果为，仍旧先调用父类无参构造函数，再执行Mammal:: Mammal(4)调用指定的int型参数的构造函数

Cat为自定义的类

void main()

{

cout<<"Cat friSky..."<<endl;

Cat fraSky; //局部变量，在栈中创建之，调用构造函数

Cat \*p=new Cat; //栈中定义指针p；堆中创建对象，调用构造函数

delete p; //调用析构函数，释放堆内存

}//函数结束后，调用析构函数释放栈中对象fraSky

SimpleCat为一个自定义的类；

类成员方法void SimpleCat::name(){cout<<"i don't kown myname"<<endl;}

方法SimpleCat functionOne(SimpleCat theCat){return theCat},按值传递对象，按值返回对象

按值传递对象，在栈中创建对象的一个副本，调用复制构造函数。函数返回时调用析构函数。

按值返回对象，创建对象的一个副本。在该副本不再有用时，销毁之

执行functionOne(obj).name();运行结果：

SimpleCat复制构造函数...//按值传递对象，创建对象的副本

SimpleCat复制构造函数...//按值返回，创建theCat对象的另一个副本2

SimpleCat析构函数... //函数返回副本1被销毁

i don't kown my name //副本2执行它的name()方法

SimpleCat析构函数... //副本2再无用处被销毁

C++中：protected对子类可见，对其他类私有，不可以在main()中对象名.调用

（1）

Mammal为父类，有析构函数~Mammal() {cout<<"~Mammal()"<<endl;}

Dog为子类，public继承，有析构函数~Dog() {cout<<"~Dog()..."<<endl;}

Mammal \*p=new Dog();

delete p; p=0；

运行结果是：~Mammal()

基类的析构函数不是虚函数,在main函数中用基类的指针去操作继承类的成员,此行为未定义，一般情况是是:只是释放了基类的资源,而没有调用继承类的析构函数。一般情况下,这样的删除只能够删除基类对象,而不能删除子类对象,形成了删除一半现象,造成内存泄漏。

（2）

如果将Mammal的析构函数改为虚函数，即virtual ~Mammal() {cout<<"~Mammal()"<<endl;}

则运行结果为：~Dog()... ~Mammal()

基类的析构函数被定义为虚函数,在main函数中用基类的指针去操作继承类的成员,释放指针P的过程是:先释放了继承类的资源,再调用基类的析构函数。

C++中构造函数也可以是private类型，如实现单例类的时候

static const int成员变量在类体外初始化，像一般静态成员变量那样。

template<classT1,classT2> 其中，class也可置换为typename

在多继承中，公有派生和私有派生对于基类成员的派生类中的可访问性与单继承的规则完全相同

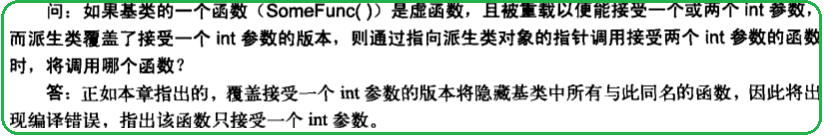
一个虚函数无论配公有继承了多少次，它仍然是虚函数;

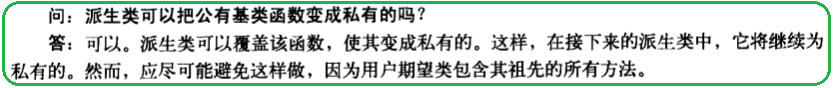
虚函数（virtual）不能是static

类对象在内存中的大小由成员变量的大小总和决定，类方法只占用少量的内存，这些内存存储指向方法位置的指针

class A {}; std::cout <<sizeof(A) << std::endl;//1

默认参数与重载函数尽量不要同时采用

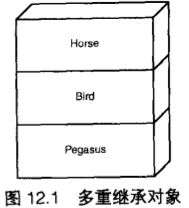




p为基类指针Dog\* pdog=dynamic\_cast<Dog\*>(p);

向下转型，如果p指向子类Dog对象，则正确返回，转型为子类指针。如果p指向父类Mammal对象，则pdog=NULL

Pegasus类继承了Horse类、Bird类，则一个Pegasus对象包括三部分：

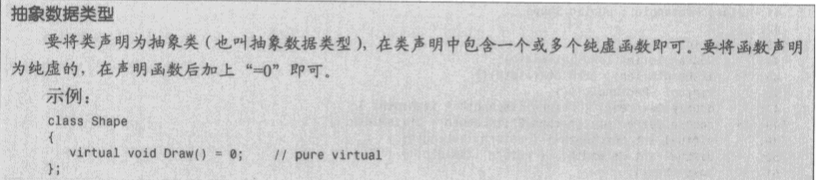


（1）两个基类都有color成员变量，都有virtual getColor()，ppeg为子类对象指针，公有继承两个基类：

ppeg->getColor()，会报错，必须指明哪个基类的getColor

ppeg->Bird::getColor()

纯虚函数：virtual void draw()=0;任何包含一或多个纯虚函数的类都是抽象类，抽象类不能实例化。抽象类的子类必须覆盖所有纯虚函数，不然其子类也是抽象类。



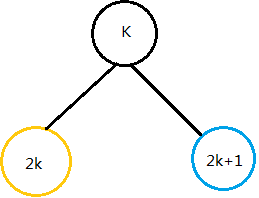
抽象类的纯虚函数可以有实现，可以不实现，但都不能实例化

使用默认参数来声明模板：template<typename T1=int,typename T2=double>

满二叉树：所有叶子结点都在最后一层，且没有度为1的结点

完全二叉树：若删掉最后一层则是满二叉树，且最后一层的元素均向左靠齐

满二叉树按层从上到下、从左到右编号，1、2、3、4……若某结点k序号为k，则其左孩子结点为2k，右孩子结点序号为2k+1



类的成员变量有指针时，需要深拷贝，不能使用默认复制构造函数

C++中的NULL必须大写，java中的null必须小写

void fun(char\*\* p)

{

p++;

cout<<\*p<<endl;

}

void main()

{

static char\* a[]={"MORNING","AFTERNOON","EVENING"};

char\*\* p=a;

fun(p); //AFTERNOON

}

p是char\*\*指针，指向的是char\*类型，char\*类型4字节，因此p++，即p指向的地址加4字节

a[]中存放的是指针，p=a，p加四字节，即指向数组的第二个元素，因此打印AFTERNOON

register：这个关键字请求编译器尽可能的将变量存在CPU内部寄存器中，而不是通过内存寻址访问，以提高效率

内联函数在编译时是将该函数的目标代码插入每个调用该函数的地方

::x = 6; //通过作用域解析运算符::引用全局变量x,并且改变其值

不能忽略警告消息

C++规定short int的长度<=int，而int的长度<=long int的长度

# extern用途：

extern：用来声明外部变量和外部函数，extern只是声明而不是定义。

extern在这里起的作用是告诉编译器，你这个变量已经在其它某个.cpp中已经定义了

// 1.cpp

#include <iostream>

**void** print()

{

  std::cout << "hello!" << std::endl;

}

//2.cpp

#include <iostream>

**using** **namespace** std;

**extern** **void** print();

**void** print()

{

  cout << "world!" << endl;

}

**int** main ()

{

  print();   //输出结果为world!

}

例子：

头文件：state.h    源文件：state.cpp

 其它源文件：t1.cpp   t2.cpp  t3.cpp,  这些源文件都包含头文件state.h。

需要定义一个全局变量供这些源文件中使用：方法如下

1、在 state.h声明全局变量： extern int a;

2、在state.cpp中定义该全局变量：int a = 10;

这样其它源文件就可以使用该变量啦

# typedef：

在C++中，则可以直接写：结构名 对象名，即：tagPOINT1 p1;

以前，C中要求声明struct新对象时，必须要带上struct，即形式为： struct 结构名 对象名，如：

struct tagPOINT1

{

int x;  
 int y;  
};

struct tagPOINT1 p1;

于是就发明了：  
typedef struct tagPOINT  
{  
 int x;  
 int y;  
}POINT;

POINT p1; // 这样就比原来的方式少写了一个struct，比较省事，尤其在大量使用的时候

**陷阱：**

typedef是定义了一种类型的新别名，不同于宏，它不是简单的字符串替换。比如：  
先定义：

typedef char\* PSTR;  
int mystrcmp(const PSTR, const PSTR);

const PSTR实际上相当于const char\*吗？不是的，它实际上相当于char\* const。

原因在于const给予了整个指针本身以常量性，也就是形成了常量指针char\* const。(指针指向位置不能变)

**using和typedef：**

using可以定义模板别名（alias templates）：

template<class T>

using MyUSet = std::unordered\_set<T, MyHash<T>>;

如果使用typedef则编译报错，c++标准不支持

template<class T>

typedef std::unordered\_set<T, MyHash<T>> MyUSet; //错误

在C++11前，typedef实现模板别名的技巧为：

template<class T>

struct MyUset{

typedef std::unordered\_set<T, MyHash<T>> type;

};

int main()

{

MyUset<A>::type s1;

s1.emplace(1);

s1.emplace(2);

}

其中，

struct A{

A(int x) :m\_x(x) {}

bool operator == (const A& rhs) const { return m\_x == rhs.m\_x; }

int m\_x;

};

template<class T>

struct MyHash {

size\_t operator()(const T& a)const {return (size\_t)(a.m\_x);}

};

如果在一个模板类内使用别名，该别名又依赖模板类的参数T，则struct包裹typedef形式的别名前必须加typename，而using声明的别名前不需要

template<class T>

struct MyUSetOld{

typedef std::unordered\_set<T, MyHash<T>> type;

};

template<class T>

using MyUSetLatest = std::unordered\_set<T, MyHash<T>>;

template<typename T>

class Test {

typename MyUSetOld<T>::type us1; //需要使用typename强制指定这样的成员为类型，而不是说这样的::type是一个静态成员亦或者其它情况可以满足这样的语法

MyUSetLatest<T> us2; //编译器知道MyAllocList是一个别名模板，所以知道MyAllocList<T>一定是一个类型名

};

在C++11中，我们可以使用type traits的方法来进行移除const, 添加引用等操作，如：

std::remove\_const<T>::type // const T -> T

std::remove\_reference<T>::type // T& / T&& -> T

这些即struct包裹typedef的例子。从C++14开始，标准库提供了使用using模板别名的方法，如上面的例子可以变为:

std::remove\_const\_t<T>;

std::remove\_reference\_t<T>;

每一个方法都是在之前的基础上添加了\_t。而其实现正是using模板别名：

template <typename T>

using std::remove\_const\_t = typename std::remove\_const<T>::type;

# typdef struct

typedef struct Stu

{

std::string name;

int id;

} Stu,\*pStu;

pStu等价于typdef Stu\* pStu;

string y;

y += 49;

cout << y << endl; //'1'的ASCII码为49

STL算法double r=std::round(3.49); //3四舍五入

# C++编译链接与内联函数

C+ +语言支持“分别编译”（separate compilation）。也就是说，一个程序所有的内容，可以分成不同的部分分别放在不同的.cp5p文件里。.cpp文件里的东西都是相对独立的，在编译（compile）时不需要与其他文件互通，只需要在编译成目标文件后再与其他的目标文件做一次链接（link）就行了。比如，在a.cpp文件中定义了一个全局函数“void a() {}，而在文件b.cpp中需要调用这个函数。即使这样，文件a.cpp和文件b.cpp并不需要相互知道对方的存在，而是可以分别地对它们进行编译， 编译成目标文件之后再链接，整个程序就可以运行了。

这是怎么实现的呢？从写程序的角度来讲，很简单。在文件b.cpp中，在调用 “void a()”函数之前，先声明一下这个函数“void a();”，就可以了。这是因为编译器在编译b.cpp的时候会生成一个符号表（symbol table），像“void a()”这样的看不到定义的符号，就会被存放在这个表中。再进行链接的时候，编译器就会在别的目标文件中去寻找这个符号的定义。一旦找到了，程序也就可以 顺利地生成了

头文件中只能写声明，形如：extern int a;和void f();的句子

但有4个例外：

* 头文件中可以写static变量或函数的定义
* 头文件中可以写const对象的定义。即使被包含到其他多个.cpp文件中，这个对象也都只在包含它的那个cpp文件中有效，对其他文件来说是不可见的，所以便不会导致多重定义
* 头文件中可以写内联函数（inline）的定义
* 头文件中可以写类（class）的定义。把类的定义放在头文件中，而把函数成员的实现代码放在一个.cpp文件中。或直接把函数成员的实现代码也写进类定义里面。如果把函数成员的定义写在类定义的头文件中，而没有写进类定义中，这是不合法的，因为这个函数成员此时就不是内联的了。一旦头文件被两个或两个以上的.cpp文件包含，这个函数成员就被重定义了

//a.h，可行

#pragma once

**class** A  {

**public**:  **inline** **int** max();

};

**int** A::max()  {

**return** 5 > 3 ? 5 : 3;

}

//a.h  ，不可行，一旦被两个cpp文件include，则重定义错误

#pragma once

**class** A  {

**public**:  **int** max();

};

**int** A::max()  {

**return** 5 > 3 ? 5 : 3;

}

static变量或函数、const变量、inline函数、类定义虽然“可以定义在多个源文件中”，但是“在一个源文件中只能出现一次”

const变量默认为文件作用域

声明有constexpr 的函数默认是inline 函数

C++编译器是否对某个函数进行内联优化，取决于优化级别，与inline关键字无关。

inline关键字的作用是，当编译器没有内联inline修饰的函数时，即使多个cpp文件都有该函数定义，也不报重定义错误，而是只留下一个进行链接。

“inline的作用是能定义不止一次(可以定义在多个源文件中，但是在一个源文件中只能出现一次)，能定义不止一次的好处是方便你放到头文件里，放到头文件里的好处是每个include这个头文件的.cpp文件都能看到函数体，看到函数体的好处是编译器可以内联，内联的好处是避免了函数调用开销。另外，所有函数体定义必须一模一样，不然出了问题概不负责。”

# .h文件和.cpp文件

预处理#include：一般main函数写在.cpp文件中，不妨设为main.cpp。根据main.cpp 的#include指令，插入其它头文件，插入过程中可能其它头文件也有#include指令，插入之，递归，直到最后得到一个完整的main.cpp文件。

编译器**独立地**对每个文件（无论对.h还是.cpp）进行词法分析、语法分析、语义分析，只要所有文件单独看没问题，则编译器不会报错

链接器由main函数开始多个文件

两个.cpp文件里有同一个函数的定义，链接报错“符号重定义”。如果是函数模板，则正确

//a.cpp

#include<iostream>

void func() {std::cout << "func()" << std::endl;}

//b.cpp

#include<iostream>

void func() { std::cout << "func()" << std::endl;}

//main.cpp

int main{}

一个.cpp一个.h 里有同一个函数的定义，正确。如果是函数模板，也正确

两个.h文件里有同一个函数的定义，正确。如果是函数模板，也正确

.h文件里有函数定义void func(){}，main.cpp包含了该文件，main函数里调用func()，正确。如果是函数模板，也正确

//a.h

#include<iostream>

void show() { cout << "show()" << endl;}

//main.cpp

#include"a.h"

int main() {}

.cpp文件里有函数定义void func(){}，main.cpp包含了该文件，main函数里调用func()，链接报错“符号重定义”。如果是函数模板，不报错

//a.cpp

#include<iostream>

void going(){std::cout << "going()" << std::endl;}

//b.h

#include<iostream>

void going();

#include"ten.cpp"

//main.cpp

#include<iostream>

#include"a.h" //报错，“符号重定义”，如果注释掉该句，则不会报错。如果going是函数模板则不报错

int main()

{ cout <<”main” << endl; }

# C++语法解析：

(1)参数为函数指针：

int g(int(\*pf)()); //函数声明，一个函数指针pf作为参数

int g(int pf()); //同上；pf其实是一个指针

int g(int());//参数名可以省略，所以这是g的第三种声明，去掉了pf这个名字；double与()之间可以无空格

(2)int func(double); //函数声明，参数为double变量

(3)int g(int());函数声明，函数参数为int(\*p)()函数指针

int g=int();g为变量，初始化为0，int()为0

(4)Udt a; //类Udt的对象，调用无参构造函数

Udt a(); //函数声明，按值返回Udt类型，无参

(5) background\_task是仿函数

std::thread my\_thread(background\_task()); //函数声明，按值返回std::thread类型，参数为background\_task(\*)()函数指针。

要创建一个线程，则修改为：

std::thread my\_thread{background\_task()};//创建线程，入口函数为background\_task()

(6) std::ifstream file("C:/Users/myth/Desktop/ints.dat");

list<int> data(std::istream\_iterator<int>(file), std::istream\_iterator<int>()); // <iterator>

函数声明，第一个参数为std::istreambuf\_iterator<int>类型的file，第二个参数为函数指针istreambuf\_iterator<int>(\*)()

要定义一个list对象，则修改为

list<int> data{std::istream\_iterator<int>(file), std::istream\_iterator<int>()};

类似A a(B(b))、A a(B())都会被解析为函数声明

类模板中可以内嵌一个类模板

**template**<**class** T>

**class** MyAllocator

{

**public**:

**template**<**class** U>

**struct** rebind

    {

**typedef** MyAllocator<U> other;

    };

**void** show() { cout << **typeid**(T()).name() << endl; }

};

**struct** Node {};

**int** main()

{

    MyAllocator<**int**>::rebind<Node>::other x;  //等价于MyAllocator<Node>

    x.show(); //struct Node

}

std::iterator\_traits<std::vector<int>::iterator>::value\_type x = 10;

# 十进制字面值不会是负数

对-42，负号并不在字面值之内，

它的作用仅仅是对字面值42取负值而已

C++的下标运算符[]有一个怪异之处：对于内置指针或数组类型的变量p，p[1] 和 1[p] 等价。p[-1] 和 -1[p] 不等价，-1[p]即-(1[p])，即-p[1]

# 构造析构的是否为虚函数与异常抛出

构造函数不能是虚函数、析构函数可以是虚函数

构造函数和析构函数内不应该调用虚函数，如果调用了，则此时虚函数相当于非虚函数

构造函数可以吐出异常到函数外

析构函数不能吐出异常到函数外，如某个函数内抛出异常，此时析构之前创建的局部对象，该对象的析构函数如果又抛出异常，则出现同时抛出两个异常，程序崩溃

**class** Object

{

**public**:

    ~Object() {

        cout << "Object deconstruct" << endl;

**throw** 3;

    }

};

**int** main()

{

**try** {

        Object obj;

**throw** "Only for testing";

    }

**catch**(...){   cout << "exception" << endl;  }

}

使用RAII时，有时会在析构函数里调用resource.close()，而close()操作可能抛出异常，怎么处理？

**class** DBConn {

**public**:

**void** close()

    {

        db.close();

        closed = **true**;

    }

    ~DBConn()

    {

**if** (closed) **return**;

**try** { db.close(); }

**catch** (…) {

             在日志中记下对close 的调用失败；

                然后视情况调用std::abort() 还是吞下异常什么也不作

        }

    }

**private**:

    DBConnection db;

**bool** closed;

};

如果用户没有手动调用DBConn::close()，而且在析构函数中db.close() 抛出了异常，析构函数吞下该异常或结束程序，客户没有立场抱怨，毕竟他们曾有机会自己调用DBConn::close()来第一手处理问题，而他们选择了放弃

宏assert(判断式)用于运行时断言，不能提前到编译期发现错误

关键字static\_assert(常量表达式，字符串字面量)用于编译器断言，如果指定的常量表达式为false，则编译器显示指定的消息，并且编译失败，错误为 C2338；否则，声明不起作用。

const bool a = false;

assert(a); //编译正常，运行时报错

static\_assert(a, "a not is true"); //编译出错

# +运算符

对内置bool、char、short替提升为int

cout << typeid('c').name() << endl; //char

cout << typeid(+'c').name() << endl; //int

cout << typeid('c'<<2).name() << endl; //int，为396

cout << typeid(true).name() << endl; //bool

cout << typeid(+true).name() << endl; //int

cout << typeid(+(short)3).name() << endl; //int，不提升则为short

cout << typeid(+(unsigned short)3).name() << endl; //int，不提升则为unsigned short

使用+显式地将无捕获列表的lambda表达式转为函数指针，虽然隐式转换存在，但在大量使用模板的情况下隐式转换通常要炸。写法为func(+[](){})

void\* ptr = nullptr;;

int\* a = (int\*)ptr; //void\*指针不能直接赋值给int\*指针，必须显示转换

void\* b = a; //int\*指针可以直接赋值给void\*指针

int\* x = (int\*)malloc(sizeof(\*x));//malloc返回void\*

free(x); //free参数为void\*

string x = "123";

const char\* p = x.c\_str();

cout << std::boolalpha << (p == &x[0]) << endl; //true

const char\* x = std::to\_string(100).c\_str(); //bug,to\_string返回临时对象，在语句结束后析构

对象没有被成功创建，空指针依然可以调用成员方法。因为对象指针指向的是私有变量（们）的地址，成员方法根本不在那个段上。所以如果成员方法完全不访问成员变量，这个野指针指向的对象的成员方法居然能被调用成功！但是如果访问到了成员变量，就会在访问的那一行挂掉

如 class Widget{public:void show() { cout << "Widget show()" << endl; }};

Widget\* p = nullptr;

p->show(); //打印出Widget show()

# 类型转换运算符

static\_cast 、dynamic\_cast、 reinterpret\_cast 、const\_cast

1. 编译器执行的任何隐式类型转换都可以由static\_cast来完成，static\_cast等价于C风格的括号强制转换，如double d=10.3; int i = (double)d;
2. dynamic\_cast在运行时进行转换。父类Base，子类Sub，有父类对象base，子类对象sub，如果Base\* sub2base = dynamic\_cast<Base\*>(sub)，则返回指向子类对象的父类指针；Sub \* base2sub = dynamic\_cast<Sub\*>(base)，如果含父类不含虚函数则编译报错，如果父类含虚函数则返回nullptr。
3. const\_cast将常量指针(int const\* p)、常量引用(const int& ref)转换为普通指针、普通引用，或者将普通指针、普通引用转换为常量指针、常量引用。const\_cast中的类型必须是指针或引用。
4. reinterpret\_cast其实什么都没做，只是告诉编译器把某种类型视为另一种类型。如可以将指针转换为整数，如uint32\_t ptr\_addr = reinterpret\_cast<uint32\_t>(ptr)，如果ptr的地址为0x0061E6D8，则ptr\_addr值为 0061e6d8。reinterpret\_cast转换前后，地址数值不变，因此只有当你知道一个指针可以转换成目标类型的指针，才可以使用reinterpret\_cast，而如何才能知道是否可以转换成目标类型，这涉及到对C++对象布局的理解。

C语言中，基本类型间的强制转换，如int到double，内存发生了变化。struct指针之间的类型转换，其实什么都没做，只是告诉编译器把某种类型视为另一种类型，指针的值并未发生变化。

C++中有了继承，static\_cast在父子类指针之间转换时，会对指针进行相应的偏移，以保证经过偏移后的指针正确指向目标类型，指针的值发生了变化

示例：

#include <iostream>

class ICover{int b = 1;};

class IPage{int a = 2;};

class Book : public ICover, public IPage {

public:void f(){

std::cout << "c = " << c << std::endl;

std::cout << "c = " << \*((int\*)(((char\*)this) + 8)) << std::endl;

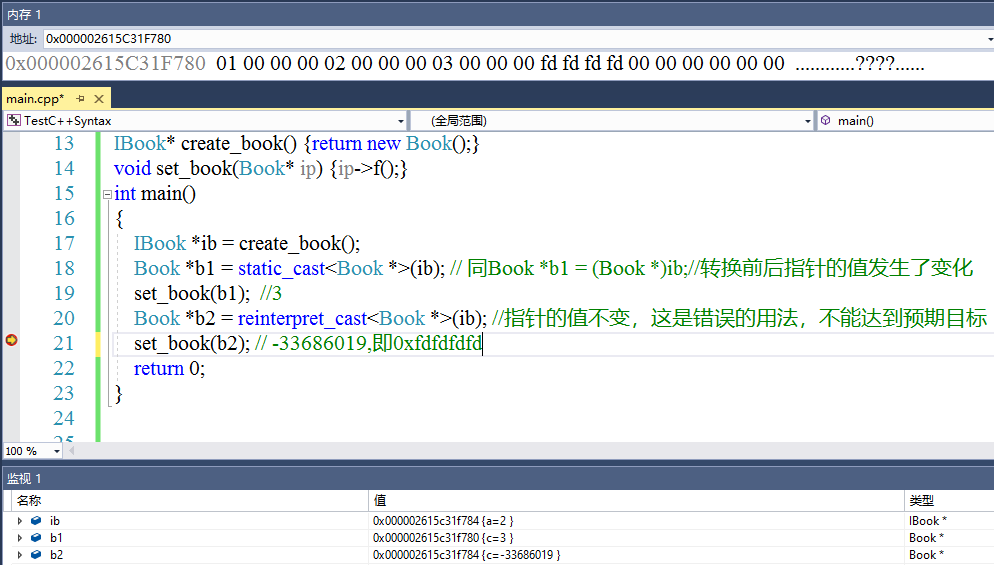
}

int c = 3;

};

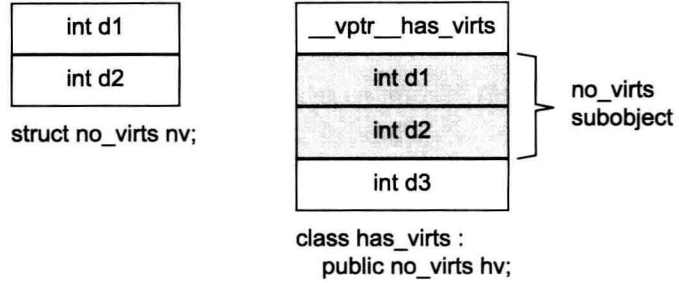
ICover\* create\_book() {return new Book();}

void set\_book(Book\* ip) {ip->f();}



许多程序员相信，转型其实什么都没做，只是告诉编译器把某种类型视为另一种类型。这是错误的观念，如

1. int转double、double转int，两者底层表述并不相同，因此几乎肯定会产生一些代码
2. 父子类指针转换。父类无虚函数，子类有虚函数时，子类指针转为父类指针



class A {int a;};

class B :public A{int b; public:virtual void show() {}};

B b;

A\* p = &b;

cout << &b << endl; //00D9F900

cout << p << endl; //00D9F904

如果class A也有虚函数，或者class B也没有虚函数，则&b和p值相同

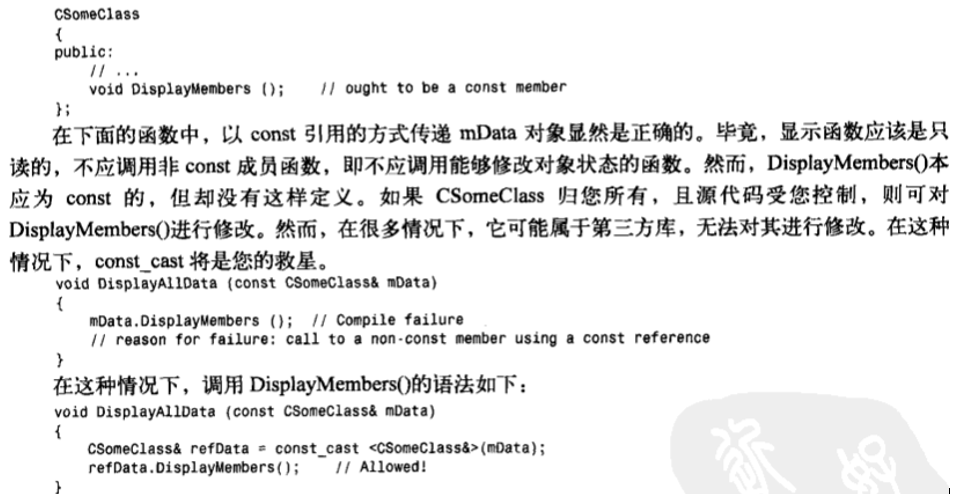
g++ -fdump-class-hierarchy main.cpp生成输出文件，通过文件可以查看内存布局及虚表内容

const\_cast用途：

class的const对象能调用其const成员函数，不能调用非const成员函数

const成员函数内部不能调用该类的非const成员函数

static成员函数内部不能调用该类的非static成员函数；类的静态成员函数不能直接访问非静态成员变量



const int&转为int&：

int b = 10;

const int& rb = b;

(int&)rb = 20;

cout << b << endl; //20

转换为自定义类：

A a(30);

const A& ref = a;

((A)ref).add(); //实际上是生成的一个A类型副本调用了add函数

# 初始化

* 用户不显式初始化时，类对其class成员默认调用无参构造函数初始化，对内置类型不保证初始化
* 不同编译单元里的非局部静态变量的初始化顺序不保证

如a.cpp中int a=10;b.cpp中extern int a; int b=a;则很有可能b的值是0，因为a的初始化可能在b的初始化之后

静态变量（static变量）包括：全局变量、命名空间里的变量、文件域static变量、类的static成员变量、函数内的static变量

函数内的static变量也被称为局部静态变量，除此之外所有变量都是非局部静态变量（non-local static）

对静态变量，程序开始运行时就为之分配好了内存，用 0填充，直到程序结束内存才被释放

* 常量必须在定义的同时初始化，包括指针常量，如int\* const p=&a；
* 类的引用类型成员、const成员变量必须在初始化列表初始化
* 左值引用、右值引用必须在定义的同时初始化
* 类的static const成员同普通static成员一样，在类体外初始化，
* 类的静态成员在类内声明，类外定义并初始化

**class** Singleton  {

**public**:

    Singleton() { cout << "Singleton构造.." << endl; }

    ~Singleton() { cout << "Singleton析构.." << endl; }

**static** Garbo gar;  //此处仅声明，定义在类外

};

Garbo Singleton::gar; //此处定义并初始化

# 慎用const int\*强转为int\*：

const int a = 10;

const int \*cpa = &a;

int \*pa = const\_cast<int\*>(cpa); //或nt \*ip = (int\*)(p);

\*pa += 2; //未定义行为

cout << \*pa << endl; //10

cout << a << endl; //12

int b = 10;

const int \*cpb = &b;

int \*pb = const\_cast<int\*>(cpb); //或nt \*ip = (int\*)(p);

\*pb += 2; //合法

If you cast away the constness of an object that has been explicitly declared as const, and attempt to modify it, the results are undefined

However, if you cast away the constness of an object that has not been explicitly declared as const, you can modify it safely

# 类的成员函数里的静态变量为类的所有对象所共享：

class Yer

{

public:

void show()

{

static bool tag = true;

tag = !tag;

qDebug() << tag;

}

};

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

{

Yer y1;

y1.show(); //false

Yer y2;

y2.show(); //true

}

[**C++ 中的 extern "C"**](https://zhuanlan.zhihu.com/p/123269132)

标准头文件常有类似以下的结构：

//incvxworks.h

#ifndef \_\_INCvxWorksh

#define \_\_INCvxWorksh

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*...\*/

#ifdef \_\_cplusplus

}

#endif

#endif /\* \_\_INCvxWorksh \*/

\_\_cplusplus宏用于识别C++编译器，当前代码被C++编译器编译时，则会定义\_\_cplusplus宏

对函数定义void foo( int x, int y )，C编译器编译后在符号库中的名字为 \_foo，被C++编译器编译后在符号库中的名字形如\_foo\_int\_int（C++靠这种方式实现函数重载）。

假设在 C++ 中，模块 A 的头文件如下：

//模块A头文件　moduleA.h

#ifndef MODULE\_A\_H

#define MODULE\_A\_H

int foo( int x, int y );

#endif

在模块 B 中引用该函数：

//模块B实现文件　moduleB.cpp

#include "moduleA.h"

foo(2,3);

在链接阶段，链接器会从模块A生成的.obj文件中寻找\_foo\_int\_int这个符号

如果如果B模块被C编译，而A模块被C++编译，则查找\_foo符号而不得，导致链接错误；同理，如果B模块被C++编译，A模块被C编译，则查找\_foo\_int\_int符号而不得，导致链接错误

模块A的头文件加extern "C"声明后，变为：

//模块A头文件　moduleA.h

#ifndef MODULE\_A\_H

#define MODULE\_A\_H

extern “C” int foo( int x, int y );

#endif

在模块 B 中引用该函数：

//模块B实现文件　moduleB.cpp

#include "moduleA.h"

foo(2,3);

此时，模块A不能被C编译，因为C标准中没有extern “C”关键字，编译器不认识，编译失败；模块A被C++编译器编译时，由于由于extern “C”关键字，会以C语言方式编译，.obj文件中的符号表中存放的是\_foo符号

如果B模块被C编译，则编译报错，因为C标准中没有extern “C”关键字；如果B模块被C++编译，则由于extern “C”关键字，链接时会查找\_foo符号，链接成功

C++代码调用C库示例：

/\* c语言头文件：cExample.h \*/

#ifndef C\_EXAMPLE\_H

#define C\_EXAMPLE\_H

int add(int x,int y);

#endif

/\* c语言实现文件：cExample.c \*/

#include "cExample.h"

int add( int x, int y )

{

    return x + y;

}

// c++实现文件，调用add：cppFile.cpp

extern "C"

{

    #include "cExample.h"

}

int main(int argc, char\* argv[])

{

    add(2,3);

    return 0;

}

C代码调用C++库示例：

//C++头文件 cppExample.h

#ifndef CPP\_EXAMPLE\_H

#define CPP\_EXAMPLE\_H

extern "C" int add( int x, int y );

#endif

//C++实现文件 cppExample.cpp

#include "cppExample.h"

int add( int x, int y )

{

return x + y;

}

/\* C实现文件 cFile.c

/\* 这样会编译出错：#include "cExample.h" \*/

int add( int x, int y );

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

{

add( 2, 3 );

return 0;

}

总结：

1、extern "C" 只是C++的关键字，不是C的。如果在C程序中引入了extern "C"会导致编译错误。

2、extern "C" 这个关键字声明的真实目的，就是实现C++与C的混合编程。一旦被extern "C"修饰之后，便以C的方式工作：编译阶段：以C的方式编译，链接阶段：寻找C方式编译生成的符号

# 常量指针/引用与指针/引用常量

* const int\*p：常量指针，不能通过p来改变所指的内容。

**int** n = 10;

**const** **int**\* p = &n;

\*p = 20;         //非法，不能通过p修改n

n = 20;          //合法，n的值依然可变

* int \*const p：指针常量，指针自身是一个常量，必须定义的同时初始化
* const int\* const a = &b; 指针本身为常量，不能通过指针来改变所指的内容

**C++标准规定，const关键字放在类型或变量名之前等价的**

const int n=5;等价于int const m=10;

const int \*p;等价于int const \*q;

const int& a; 等价于int const& a;

常量引用：

int v=100;

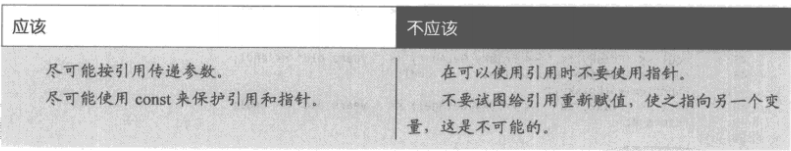
const int& rv=v; //不能通过引用改变v的值

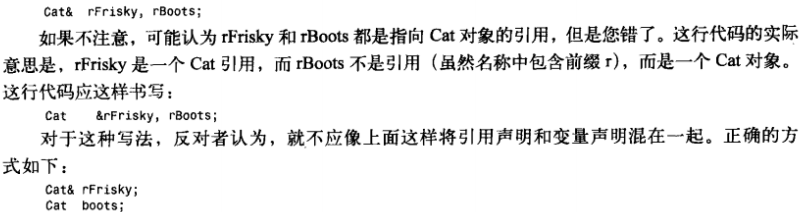
A f; // A为自定义class

const A& e = f; // 这样e只能访问声明为const的函数，而不能访问一般的成员函数

数组名可隐式转换为一个指针常量，指向数组的第一个元素







# 逻辑运算符与位运算符

（1）C++中、java中：

&&、|| 、！执行逻辑运算。其前后各接一逻辑值，返回两逻辑值之“与”运算逻辑值和“或”运算逻辑值、非运算逻辑

&、|、^ 是一组位运算符。其前后各接一数字，返回两数字之“与”运算、“或”运算、“异或”运算值

&、|不仅可以操作布尔型,而且可以操作数值型的，&&、|| 、！只能操作布尔型

对于A&B，不管a是否为假，仍然要判断B。

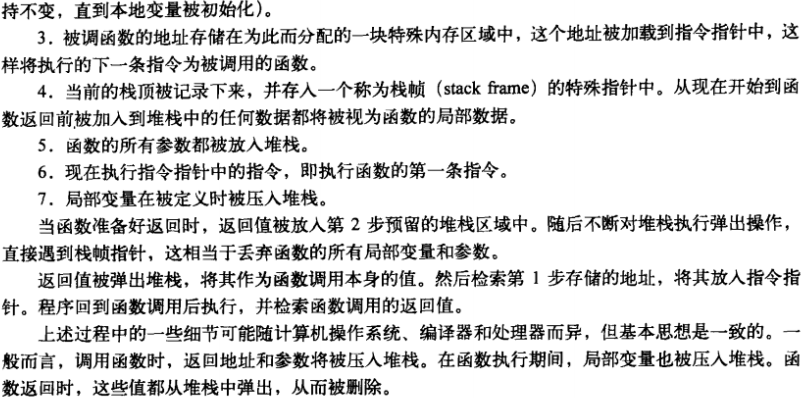
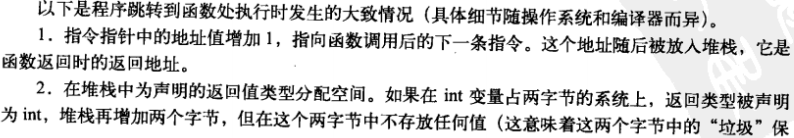
对于A|B，不管a是否为真，仍然要判断B。

而对于A&&B，A||B在以上情况就不会去判断B了。

C++中：逻辑值就是“true”或“false”，为了用数字表示逻辑值，在 C++中规定，非零值即为逻辑真，而零则为逻辑假。!5的返回值是0，因为先由5非零而知是逻辑真，然后取反得逻辑假，故最后返回0。!!345.4的返回值是1，先因345.4非零得逻辑真，取反后得逻辑假，再取反得逻辑真。虽然只要非零就是逻辑真，但作为编译器返回的逻辑真，其一律使用1来代表逻辑真。

Java中：System.out.println(!true)结果为false；System.out.println(!5)则编译出错。Java中boolean型不能与其他类型（如int）转换。boolean x=0则编译出错

# C++函数调用过程深入分析：

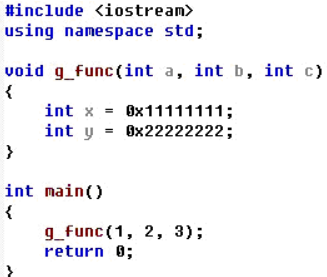
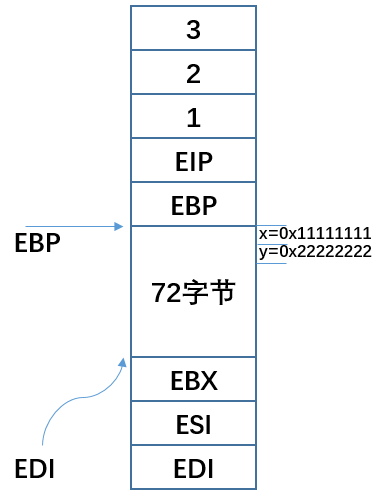


栈向下生长，地址越来越小

ESP：Extended Stack Pointer，扩展栈指针寄存器，指向当前栈顶字节，当小尾端时，ESP也即栈顶元素（如int）的低位地址

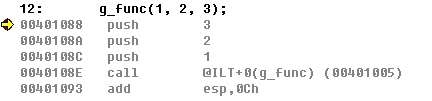
EBP：Extended Base Pointer，扩展基址指针寄存器，存放栈底地址

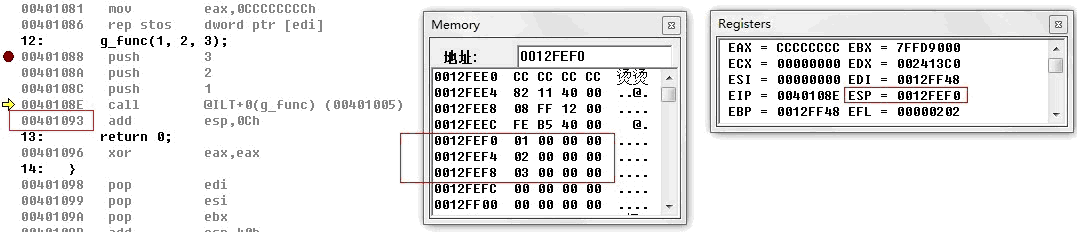
**C++源码示例：**

 ————>

**汇编分析：**

1. 函数参数入栈，参数列表由后往前依次是是3、2、1





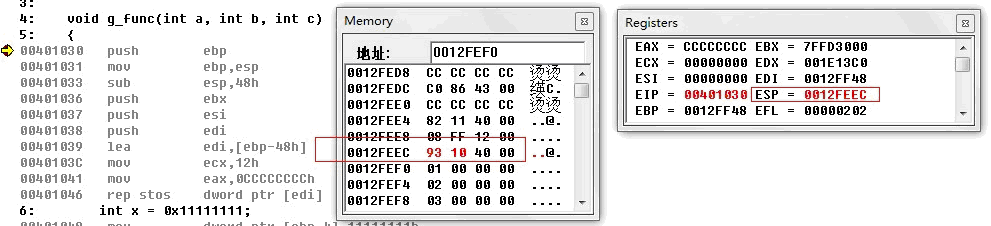
1. call指令

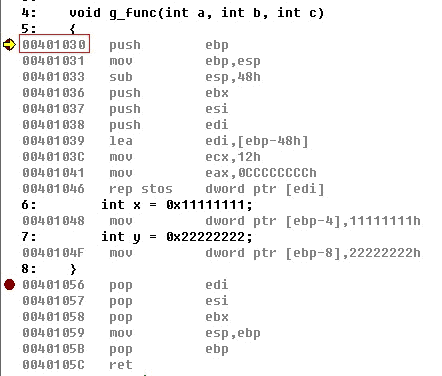
CALL指令跳转到地址0x00401005，那么该地址处是什么呢？我们继续跟踪一下，看到这里又是一条跳转指令，跳转到0x00401030，即真正被调函数的第一条指令。

http://hi.csdn.net/attachment/201108/11/0_13130663458reL.gif

参数压栈后，执行call汇编命令，call命令：

1. 将返回地址即EIP（指令指针寄存器）压栈；
2. 跳转到被调函数处0x00401030。





1. 当前栈底寄存器EBP的值压栈、更新EBP里的值为当前栈顶ESP、ESP减去一段
2. EBX、ESI、EDI压栈

（

*EBX：Extended Base Register，基址寄存器，无重要的专门用途，可存放指针等*

*ESI：Extended Source Index，源变址寄存器，在内存操作指令中作为“源地址指针”使用。*

*EDI：Extended Destination Index，目的变址寄存器，在内存操作指令中作为“目的地址”使用*

）

1. 将减去的一段全赋值为0xCCCCCCCC（对应lea、mov、mov、rep stos四条指令），代表着未被初始化
2. 修改减去的一段中局部变量x、y的值
3. 依次弹栈至EDI、ESI、EBX寄存器
4. 弹栈减去的一段，即mov esp,ebp
5. 弹栈至EBP寄存器
6. 栈顶元素弹至EIP寄存器，即ret指令，恢复断点
7. 栈顶弹出3个int，即add esp, 0Ch让栈顶指针往上移动12Byte的位置。因为入栈的是3个int参数

**ESP减去的一段的大小**：

在VS2015 x86Debug编译选项下，如果g\_func(int,int,int)函数中未定义局部变量，则sub esp,0C0h，如果仅定义一个局部变量，则sub esp,0CCh，如果定义2个局部变量，则sub esp,0D8h，如果定义3个局部变量，则sub esp,0E4h，如果定义4个局部变量，则sub esp,0F0h

即减去的一段包括：（1）间隔区域的大小固定为C0h，即192Byte；（2）预留出的存储局部变量的内存区域

**给减去的一段赋值：**

lea edi,[ebp-48h]

mov ecx,12h

mov     eax,0CCCCCCCCh

rep stos dword ptr [edi]

栈上从ebp-0x48开始的位置向高地址方向的内存赋值0xCCCCCCCC，次数重复0x12(18)次，共赋值了18\*4=72=0x48个字节

AX：累加器，算术运算主存储器，EAX 是AX的32位扩展

CX： Count，计数寄存器，用于循环操作和数据串处理，ECX是CX的32位扩展

lea是“load effective address”的缩写，简单的说，lea指令可以用来将一个内存地址直接赋给目的操作数，例如：lea eax,[ebx+8]就是将ebx+8这个值直接赋给eax，而不是把ebx+8处的内存地址里的数据赋给eax。

而mov指令则恰恰相反，例如：mov eax,[ebx+8]则是把内存地址为ebx+8处的数据赋给eax

rep指令的目的是重复后边的指令，ECX的值是重复的次数。只要ecx的值不为0, 重复就会继续. 每一次字符串指令执行后, ecx的值都会减小

STOS指令的作用是将eax中的值拷贝到ES:EDI指向的地址. 如果设置了direction flag, 那么edi会在该指令执行后减小, 如果没有设置direction flag, 那么edi的值会增加, 这是为了下一次的存储做准备.

**mov eax, dword ptr [12345678]：**

dword 双字 就是四个字节

ptr pointer缩写 即指针

把内存地址12345678中的双字型（32位）数据赋给eax

[]里的数据是一个地址值，这个地址指向一个双字型数据

*在函数调用中返回结果都是存储在eax中的。*

如果是int res=add(2,3)，则res在主调函数栈的局部变量区域中，提前分配了位置，函数调用的结果在EAX寄存器中

在Linux下反汇编32位的程序时，经常会遇到\_\_x86.get\_pc\_thunk.ax函数，该函数的功能是将%eip寄存器内容传入%eax寄存器。相当于mov %eip,%eax。原因是x86的指令集中没有直接读取%eip的指令

# 引用的汇编实现：

#include<iostream>

using namespace std;

void func()

{

const int& a = 10;

int y = a + 1;

}

int main()

{

func();

return 0;

}

为避免出现位置无关指令，形如<\_\_x86.get\_pc\_thunk.ax>，加-no-pie -fno-pic参数

x86编译，加-m32参数

g++ -g -m32 -no-pie -fno-pic -o conref conref.cpp

gdb-peda$ disas func

Dump of assembler code for function func():

endbr32

push ebp

mov ebp,esp

sub esp,0x18

mov eax,gs:0x14

mov DWORD PTR [ebp-0xc],eax

xor eax,eax ;比mov eax,0效率高

mov eax,0xa

mov DWORD PTR [ebp-0x18],eax ;常量10存储于内存[ebp-0x18]

; 以下5句汇编为对引用的使用，即代码里的a+1

lea eax,[ebp-0x18]

mov DWORD PTR [ebp-0x14],eax ; 引用a，实际为指针，存储于内存[ebp-0x14]

mov eax,DWORD PTR [ebp-0x14]

mov eax,DWORD PTR [eax]

add eax,0x1

mov DWORD PTR [ebp-0x10],eax ; 变量y存储于内存[ebp-0x10]

nop

mov eax,DWORD PTR [ebp-0xc]

xor eax,DWORD PTR gs:0x14

je 0x8049236 <func()+64>

call 0x80490b0 <\_\_stack\_chk\_fail@plt>

leave

ret

End of assembler dump.

# C++内存：

使用#define value 15;定义常量，不用指出类型。预处理器进行简单的文本替换，编译器看到的是15，而看不到常量value。

常量必须在定义的同时初始化const unsigned short int value=15，包括指针常量

引用必须在定义的同时初始化

如果是定义在全局下的全局变量，这个数组就在全局变量区。  
如果是定义在函数下的[局部变量](https://www.baidu.com/s?wd=%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujNbuyfkPhDzPyuhPycY0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtdPjD4PjT1rHD)，这个数组就在[函数调用](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujNbuyfkPhDzPyuhPycY0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtdPjD4PjT1rHD)栈里。  
用new申请堆空间存放数组， [函数调用](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujNbuyfkPhDzPyuhPycY0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtdPjD4PjT1rHD)结束后，栈里面为这个函数分配的空间就要失效，从而[局部变量](https://www.baidu.com/s?wd=%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujNbuyfkPhDzPyuhPycY0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtdPjD4PjT1rHD)数组也会失效。但堆空间不会因为[函数调用](https://www.baidu.com/s?wd=%E5%87%BD%E6%95%B0%E8%B0%83%E7%94%A8&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9ujNbuyfkPhDzPyuhPycY0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtdPjD4PjT1rHD)结束失效，函数调用结束后，函数里new出来内存空间仍然有效。

堆空间分配的内存一直有效，直到程序员将其释放

不要用return语句返回指向“栈内存”的指针或引用，因为该内存在函数结束时将自动释放，见下例：

        Char\* GetString(void)

        {

               char p[] = "hello world";     //用字符串常量来初始化数组的内存空间

               return p;                            //编译器将提出警告

        }

        void Test4(void)

       {

              char \*str = NULL;

              str = GetString();

              cout<<str<<endl;               //str的内容是垃圾

       }

       用调试器逐步跟踪Test4(),发现执行str = GetString()语句后str不再是NULL指针，但是str的内容不是"hello world"而是垃圾。

      将上例做如下修改：

      const char \* GetString2(void)

      {

             const char \*p="hello world";

             return p;

      }

      void Test5(void)

     {

            const char \*str = NULL;

            str = GetString2();

            cout<<str<<endl;

     }

     函数Test5()运行时不会出错。字符串常量是被放在静态存储区中的。

     在第一个例子中：char p[]="hello world";中，"hello world"是一个字符串常量，存放在静态数据区；但是把一个字符串常量赋值给了一个局部变量（char p[]型数组），该局部变量是存放在栈中的。这样子就有两块内容一样的内存，意思就是说：char p[]="hello world";这条语句让"hello world"在内存中有两份拷贝，一份在动态分布的栈中，一份在静态存储区，当函数GetString退出时，局部变量p的内存是要被清空的。而第二个例子中变量str是指向静态存储区的指针，此变量指向的内存在程序运行过程中不会被清空

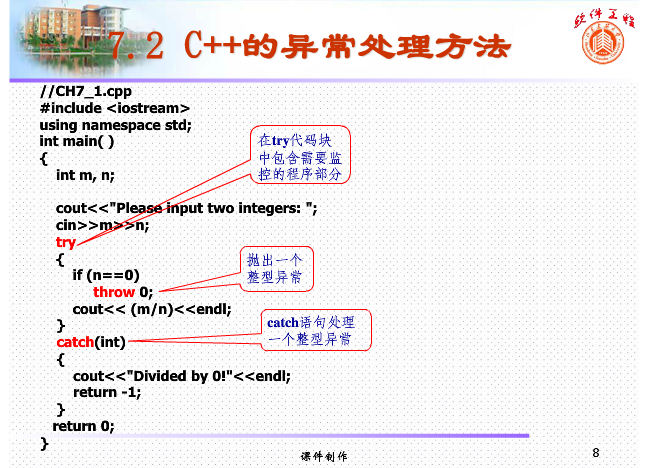
char\* pstr="hello C++";pstr指向常量区，不能执行delete pstr

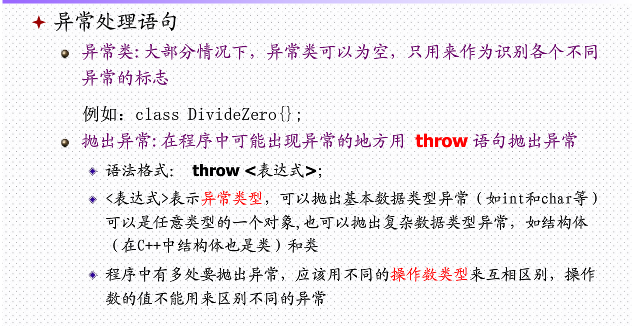
char str[]="hello C++"；常量区创建字符串"hello C++"，栈中亦创建字符串"hello C++"，str指向栈中的字符串

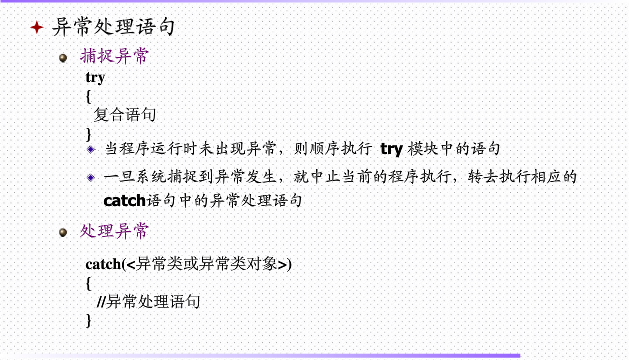
char\* buf="good";

buf[0]='h';//不能修改

# 异常处理;







#include<exception>

class empty\_stack : public std::exception

{

public:

virtual const char\* what() const //多态，覆盖父类的virtual const char\* what() const函数

{

return "empty stack!"; //返回指向Text段内存的地址

}

};

throw empty\_stack()

try{ }

catch (std::exception& e)

{cout << e.what() << endl;}

关键字throw(something)是一种异常规范，只会出现在声明函数时

void fun() throw();      //表示fun函数不允许抛出任何异常，即fun函数是异常安全的。

void fun() throw(...);    //表示fun函数可以抛出任何形式的异常。

void fun() throw(exceptionType);    // 表示fun函数只能抛出exceptionType类型的异常。

举例说明：

void GetTag() throw(int);                     // 表示只抛出int类型异常

void GetTag() throw(int，char);         // 表示抛出in，char类型异常

void GetTag() throw();                         // 表示不会抛出任何类型异常

void GetTag() throw(...);                      // 表示抛出任何类型异常

void GetTag() throw(int); 表示只抛出int类型异常,并不表示一定会抛出异常，但是一旦抛出异常只会抛出int类型，如果抛出非int类型异常，调用unexsetpion()函数，退出程序。

void GetTag() throw();  假如你加一个throw()属性到你的永远不会抛出异常的函数中，编译器会非常聪明的知道代码的意图和决定优化方式

# 产生随机数：

计算机的随机数都是伪随机数，由小M多项式序列生成的，其中产生每个小序列都有一个初始值，即随机种子。小M多项式序列的周期是65535，即每次利用一个随机种子生成的随机数的周期是65535，当你取得65535个随机数后它们又重复出现了。

* 用法：int rand() 所在头文件: cstdlib rand()返回一个随机数值，范围在0至RAND\_MAX 间。用户未设定随机数种子时，系统默认的随机数种子为1，此时rand()产生随机数每次执行时是相同的；若要不同，用函数srand()初始化它
* 用法: void srand(unsigned int seed) 功能:初始化随机数发生器 所在头文件: stdlib.h srand()用来设置rand()产生随机数时的随机数种子。参数seed必须是个整数，如果每次seed都设相同值，rand()所产生的随机数值每次就会一样
* rand()产生的随机数在每次运行的时候都是与上一次相同的。若要不同,用函数srand()初始化它。可以利用srand((unsigned int)(time(NULL))的方法，产生不同的随机数种子，因为每一次运行程序的时间是不同的

#include <cstdlib>

#include <ctime>

using namespace std;

int main()

{

srand((unsigned)time(NULL));

int i =rand();

int j =rand();

}

/\*

函数功能：产生［0,ULONG\_MAX］区间的随机数（ULONG\_MAX=0xffffffff）

<<，二进制左移，其实相当于乘以2

rand()产生［0,RAND\_MAX］区间的随机数　RAND\_MAX=0x7fff＝32767

ulrand()函数要产生［0,ULONG\_MAX］区间的数（ULONG\_MAX=0xffffffff）,则

(1)必须利用移位

(2)不能利用rand()的生成的所有16位数，因为rand的前４位bit的范围在［0,7］区间，即不能是((unsigned long)rand()<<16) & 0xFFFF0000UL)| ((y&0x0000FFFFul)

\*/

unsigned long ulrand ()

{

unsigned long x = rand ();

unsigned long y = rand ();

unsigned long z = rand ();

return (

((x << 24) & 0xFF000000UL)

| ((y << 12) & 0x00FFF000ul)

| (z & 0x00000FFFul)

);

}

无论main中的srand是如何的，线程中的rand结果只与该线程中的srand有关。不受其它线程中srand调用的影响

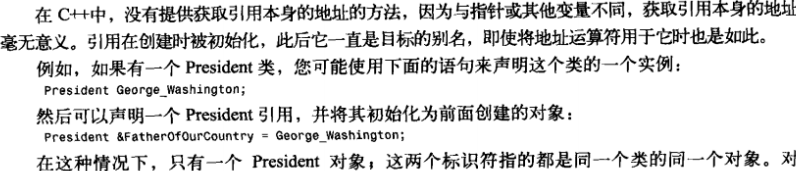
C++的<random>头文件，产生[INT\_MIN, INT\_MAX]范围随机数

std::random\_device rd;

std::uniform\_int\_distribution<int> dist(INT\_MIN, INT\_MAX);

int r = dist(rd);

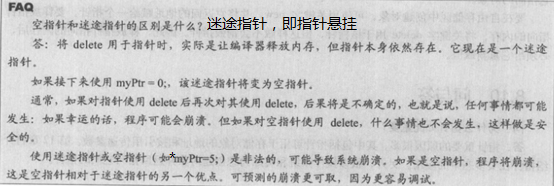
# C++指针：



修饰返回值的const， const A\* fun3( )，不能再通过返回的指针来修改所指向的内容



delete p; //释放指针所指向的堆中内存，但指针依然存在，可以重新给之赋值



内存泄漏：①函数中int \* p=new int()函数返回后指针出栈，指针将不在作用域，但所指堆内存依然存在、不能被释放、再利用；②重新给指针赋值之前，没有释放其原来所指向的内存

delete p;释放堆内存后，若再delete p，则系统崩溃。故删除指针后，最好再赋值为0，delete 空指针不会崩溃

delete指针后，没有设置为空，指针仍指向原来的区域，指针悬挂。若再使用，危险！！

STL 的迭代器、函数对象、内置类型选择pass by value比 pass by reference的效率高。除此外，本来按值传递都应该改为常量引用传递。C++ 编译器的底层，references 往往以指针实现出来，因此pass by reference 通常意味真正传递的是指针。因此如果你有个对象属于内置类型（例如 int) , pass by value往往比 pass by reference的效率高些。

int\* p= new int(10);

cout << p << endl; //000002CDB63F6550

delete p;

cout << p << endl; //0000000000008123

C++指针delete之后，再使用指针就属于未定义行为。VS下指针的值变化，GCC没有。delete一个指针后，成为野指针。

1、指针的类型：

从语法的角度看，只要把指针声明语句里的指针名字去掉，剩下的部分就是这个指针的类型。这是指针本身所具有的类型。

int \*ptr; //指针的类型是int \*

char \*ptr; //指针的类型是char \*

int \*\*ptr; //指针的类型是 int \*\*

int (\*ptr)[3]; //指针的类型是 int(\*)[3]

int \*(\*ptr)[4]; //指针的类型是 int \*(\*)[4]

2、指针所指向的类型

当通过指针来访问指针所指向的内存区时，指针所指向的类型决定了编译器将把那片内存区里的内容当做什么来看待。

从语法上看，只需把指针声明语句中的指针名字和名字左边的指针声明符\*去掉，剩下的就是指针所指向的类型。

int \*ptr; //指针所指向的类型是int

char \*ptr; //指针所指向的的类型是char

int \*\*ptr; //指针所指向的的类型是 int \*

int (\*ptr)[3]; //指针所指向的的类型是 int()[3]

int \*(\*ptr)[4]; //指针所指向的的类型是 int \*()[4]

在指针的算术运算中，指针所指向的类型有很大的作用。

指针的类型(即指针本身的类型)和指针所指向的类型是两个概念。

3、 指针的值

指针的值是指针本身存储的数值，这个值将被编译器当作一个地址，而不是一个一般的数值。在32位程序里，所有类型的指针的值都是一个32位整数，因为32位程序里内存地址全都是32位长。

指针所指向的内存区就是从指针的值所代表的那个内存地址开始，长度为sizeof(指针所指向的类型)的一片内存区。以后，我们说一个指针的值是XX，就相当于说该指针指向了以XX为首地址的一片内存区域；我们说一个指针指向了某块内存区域，就相当于说该指针的值是这块内存区域的首地址。

指针本身占了多大的内存？用函数sizeof(指针的类型)测一下就知道了。在32位平台里，指针本身占据了4个字节的长度。

4、指针的算术运算

指针可以加上或减去一个整数。指针的这种运算的意义和通常的数值的加减运算的意义是不一样的。例如：

char a[20];

int \*ptr=a;

...

...

ptr++;

在上例中，指针ptr的类型是int\*,它指向的类型是int，它被初始化为指向整形变量a。接下来的第3句中，指针ptr被加了1，编译器是这样处理的：它把指针ptr的值加上了sizeof(int)，在32位程序中，是被加上了4。由于地址是用字节做单位的，故ptr所指向的地址由原来的变量a的地址向高地址方向增加了4个字节。

如果上例中，ptr是被减去5，那么处理过程大同小异，只不过ptr的值是被减去5乘sizeof(int)，新的ptr指向的地址将比原来的ptr所指向的地址向低地址方向移动了20个字节。

　对于数组名本身如何实现，标准是没有规定的，数组对象在建立时，编译器管理其地址，这个地址就是数组到指针隐式转换的结果，至于编译器如何管理这个地址，把它放某个地方也好，实现为一个汇编中的立即数也好，这是编译器自己的事情，但是，当对其取地址的时候，编译器必须实现其抽象语义，就是结果为数组对象的地址，而非这个地址的存放地方的地址（如果编译器把它存放在某个地方的话）。

# C++二维数组理解：

#include<iostream>

using namespace std;

void main()

{

int a[3][4]={{11,22,33,44},{111,222,333,444},{1111,2222,3333,4444}};

cout<<sizeof(a)<<endl;//a代表数组，故12\*4=48

cout<<sizeof(&a)<<endl;//&a是一个int(\*)[3][4]指针,指向int[3][3]的指针，指针大小为4，故4

cout<<sizeof(a[1])<<endl;//a[1]是第二个int[4]一维数组的数组名，sizeof(数组名)结果为数组大小，故4\*4=16

cout<<sizeof(&a[1])<<endl;//&(数组名)，为int (\*)[4]指针，指针大小为4，故4

cout<<sizeof(a[0][0])<<endl;//a[0][0表示元素1，int型，故4

cout<<sizeof(&a[0][0])<<endl;//int\*指针，故4

}

//sizeof(指针名称)测出的究竟是指针自身类型的大小呢还是指针所指向的类型的大小？答案是前者

C++标准中数组的的定义：

If, in the declaration “T Dl.” Dl has the form

D [ constant expression opt ]

从形式上看，怎么只有一维数组的定义？这个形式如何定义多维数组？C/C++的数组是数组的嵌套，因此多维数组的定义也反映了这个本质。多维数组的定义是通过嵌套的一维数组定义构造的。

T D[M]当元素为一维数组T[N]时，元素的类型也为数组类型，用T[N]代替T，则为：T[N] D[M].这个语法结构不符合C/C++数组定义的语法形式，将[N]移动到[M]后，就是正式的二维数组的定义了：T D[M][N]其中D[0]---D[M-1]都是一维数组，具有数组类型T[N]。各种维度的多维数组可以用同样的嵌套方法构造出来。

一个一维数组T[M]，经过数组到指针的转换后，类型转换为T\*，二维数组T[M][N]转换为指针后，类型转换为T ( \* )[N]，其实原理还是跟嵌套有关，二维数组为一维数组的嵌套，元素为数组类型。

* 无论几维数组，cout<<数组名<<endl;数组名隐式转换为指向数组首元素（指的是从嵌套观点来看时数组的首元素）的指针；
* 表达式&(数组名)中，不进行数组名到指针的转换，视数组名为数组的代表，故结果为数组的首地址，即指向整个数组的指针，如int\* b[3][4]中，&b为int(\*)[3][4]型指针。
* sizeof(数组名)，不进行数组名到指针的转换，视数组名为数组的代表，结果为整个数组所占空间大小

int a[ 10 ];

int ( \*q )[ 10 ] = &a;

&a是数组的首地址，其地址值与数组名a隐式转换为int\*指针的值相同，但两者的类型是不一样的，&a作为数组首地址，类型是指向数组的指针：int( \* )[10]，而a的转换结果是指向首元素的指针，因此类型是int\*。

# C++指针与数组：

在声明和初始化一个二维数组时，只有第一维（行数）可以省略。比如：

int array[][3]={1,2,3,4,5,6};

相当于：

int array[2][3]={1,2,3,4,5,6};

（1）数组名作为函数实参参数

1. #include <iostream>
2. **using** **namespace** std;
3. **void** arrayTest(**int** a[],**int** len);
4. **void** main()
5. {
6. **int** a[3]={11,12,13};
7. cout<<a<<endl;      //0018FF20
8. //a++;//编译报错
9. cout<<"in main(),sizeof(a) is:"<<**sizeof**(a)<<endl;//12
10. arrayTest(a,3);
11. }
12. **void** arrayTest(**int** a[],**int** len)
13. {
14. cout<<"in arrayTest(int\* a[],int len),sizeof(a) is:"<<**sizeof**(a)<<endl;//4
15. cout<<a<<endl;
16. a++;
17. cout<<a<<endl;//0018FF24
18. }

/\*

　　结论1指出，数据名内涵为数组这种数据结构，在arrayTest函数体内，str是数组名，那为什么sizeof的结果却是指针的长度？这是因为：

　　(1)数组名作为函数形参时，在函数体内，其失去了本身的内涵，仅仅只是一个指针；

　　(2)很遗憾，在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

　　所以，数据名作为函数形参时，其全面沦落为一个普通指针！它的贵族身份被剥夺，成了一个地地道道的只拥有4个字节的平民。

\*/

数组类型到指针类型隐式转换的结果，是一个指向数组首元素（从数组嵌套观点看，数组的第一个自对象）的指针，并且从一个左值转换成一个右值。

当数组名作为sizeof、&的操作数时，它才代表数组对象本身，并不会隐式转换为指针。

int array2[5]={10,11,12,13,14};

cout<<array2<<" "<<&array2[0]<<" "<<&array2<<endl;//0018FE04 0018FE04

//将数组看作是一个整体，&array2是其地址，是int(\*)[5]型指针。array2隐式转换int\*型指针，值为数组首元素的地址。

int (\*parray2)[5];//指针parray类型为int (\*)[5],所指类型为int ()[5],即int [],即指向一个5个元素的数组

parray2=&array2;

cout<<"\*parray2:"<<\*parray2<<endl;//:0018FE04

cout<<parray2<<array2<<endl;//parray2:0018FE04 ,数组名array2:0018FE04

对于数组名本身如何实现，标准是没有规定的，数组对象在建立时，编译器管理其地址，这个地址就是数组到指针隐式转换的结果，至于编译器如何管理这个地址，把它放某个地方也好，实现为一个汇编中的立即数也好，这是编译器自己的事情，但是，当对其取地址的时候，编译器必须实现其抽象语义，就是结果为数组对象的地址，而非这个地址的存放地方的地址（如果编译器把它存放在某个地方的话）。

**数组的引用：**

int main()

{

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

int(&r)[3] = a; //数组引用

r[1] = 5;

cout << a[1] << endl; //5

myFunc(a);

}

const char name[] = "J. P. Briggs"; //name的类型是const char[13]

template<typename T>

void f(T param); //传值形参的模板

f(name); //T被推导为const char\*

template<typename T>

void f(T& param); //传引用形参的模板

f(name); //T被推导为const char[13]，void f(const char (&param)[13])

**智能指针：**

C++11之后智能指针分为了三种：shared\_ptr, unique\_ptr, weak\_ptr，在头文件<memory>

C++11之后的智能指针的构造函数都有explict关键词修饰，表明它不能被隐式转换

1. shared\_ptr<**int**> p1 = **new** **int**(1024); // 错误，必须使用直接初始化形式
2. shared\_ptr<**int**> p2(**new** **int**(1024));

同时只能有一个unique\_ptr智能指针对象指向某块内存。因此不能够将unique\_ptr拷贝到另一个unique\_ptr、不能够按值传递unique\_ptr。

unique\_ptr有移动构造、移动赋值，不能拷贝构造、拷贝赋值

不要用一个原始指针初始化多个shared\_ptr或unique\_ptr，原因在于，会造成二次销毁

{

**int**\* p = **new** **int**(10);

    std::unique\_ptr<**int**> p1(p);

    {

        std::unique\_ptr<**int**> p2(p);

        std::cout << "用原始指针初始化了多个unique\_ptr指针" << std::endl;

    }

    std::cout << "p2已经销毁" << std::endl;

}//于此处崩溃，unique\_ptr对象析构时一定会释放管理的指针指向的空间，因此如果该空间已经通过其它方式被释放，则二次销毁导致程序崩溃

由于vector增长时会复制对象，像std::unique\_ptr这样不可复制的对象是无法放入容器的。但随着移动语义的引入，std::unique\_ptr也可放入std::vector，不复制对象，而是“移动”对象。

vector<unique\_ptr<Moveable>> vec;

**for** (**int** i = 0; i < 5; i++)

{

    unique\_ptr<Moveable> up(**new** Moveable());

vec.push\_back(move(up));

//上述两句可以合并为vec.push\_back(unique\_ptr<Moveable>(new Moveable()));

}

指向子类对象的unique\_ptr指针可以转换为指向父类的unique\_ptr指针，多态性依然存在

指向子类对象的shared\_ptr指针可以转换为指向父类的shared\_ptr指针，多态性依然存在

一般，unique\_ptr的size是一个指针的大小，unique\_ptr的size是两个指针的大小

**unique\_ptr：**

**unique\_ptr的api：**

* unique\_ptr<T> up

空的unique\_ptr，可以指向类型为T的对象，默认使用delete来释放内存

* unique\_ptr<T,D> up(d)

空的unique\_ptr同上，接受一个D类型的删除器d，使用删除器d来释放内存

* up = nullptr

释放up指向的对象，将up置为空

* up.release()

up放弃对它所指对象的控制权，并返回保存的指针，将up置为空，不会释放内存

* up.reset(…)

参数可以为空、内置指针，先将up所指对象释放，然后重置up的值

**unique\_ptr和shared\_ptr传值和引用：**

函数形参：

* 采用unique\_ptr<widget>值类型作为参数，表明将widget的资源所有权转移到该函数内
* 采用unique\_ptr<widget>& 作为参数，表明将在函数中重新设置指针的指向
* 采用const unique\_ptr<widget>&作为参数，不建议，应使用(Widget\*) 或引用 (Widget&)
* 采用shared\_ptr<widget>值类型作为参数，共享所有权。在函数的生命周期中，我是Widget的共享所有者。在函数的开头，我增加了引用计数，在函数结束时，我减少引用计数: 因此，只要我使用Widget，它就会一直存在。
* 采用shared\_ptr<widget>&作为参数，表明可能重新设置指针的指向。我不是Widget的共享所有者，因为我不更改引用计数器。我不能保证Widget在函数执行期间直存在，但我可以重新设置资源。

void ChangeWidget(std::shared\_ptr<widget>& w){

// This will change the callers widget

w = std::make\_shared<widget>(widget{});

}

* 采用const shared\_ptr&作为参数，表明想要保持对象的引用计数不变，不建议，应该使用指针(Widget\*) 或引用(Widget&)

场景示例：如果一个对象的构造函数需要获取某个资源的所有权，那根据需要的所有权是独家的还是共享的，选择unique\_ptr或者shared\_ptr。两种智能指针写形参的时候都直接写成值传递即可，调用的时候根据是转移所有权还是分享所有权决定是否要std::move一下实参。

**返回unique\_ptr和shared\_ptr：**

在函数内部new的资源，最好使用unique\_ptr返回，如果用shared\_ptr返回，则会强逼用户全程使用shared\_ptr管理这个对象。而使用unique\_ptr返回，可以调用std::unique\_ptr::release()释放对资源的管理权，交由裸指针管理，或者轻松高效的转换为std::shared\_ptr：

std::unique\_ptr<Investment> makeInvestment(const std::string& type);

std::shared\_ptr<Investment> sp = makeInvestment(arguments); //将std::unique\_ptr转为std::shared\_ptr

**std::unique\_ptr实现链表：**

使用std::unique\_ptr智能指针的链表实现

**template**<**class** T>

**class** queue

{

**private**:

**struct** node  {

        T data;

        std::unique\_ptr<node> next;

        node(T data\_):data(std::move(data\_)){} //默认调用next的无参构造函数，因此next.get()为nullptr

~node() { cout << "~node" << data << endl; }

    };

    std::unique\_ptr<node> head;

    node\* tail=nullptr;

**public**:

queue(){}

~queue() { cout << "~queue() " << endl; }

    queue(**const** queue& rhs) = **delete**;

    queue& operator=(**const** queue& other) = **delete**;

    std::shared\_ptr<T> try\_pop()   {

**if** (!head) **return** std::shared\_ptr<T>();

**const** std::shared\_ptr<T> res=  std::make\_shared<T>(std::move(head->data));

        head = std::move(head->next);

**return** res;

    }

**void** push(T new\_value)   {

        std::unique\_ptr<node> p= std::make\_unique<node>(std::move(new\_value));

        node\* new\_tail = p.get();

**if** (tail)   {

            tail->next = std::move(p);

        }

**else**  {

            head = std::move(p);

        }

        tail = new\_tail;

    }

};

内存能被正确释放，queue对象析构，执行析构函数体内语句，析构成员unique\_ptr<node> head，delete管理的指针，析构之……

//构造时，先构造成员再执行构造函数体内语句；析构时，先执行析构函数体内语句再析构成员

**std::shared\_ptr**

**构造函数：**

（1）shared\_ptr有一个构造函数模板，explicit，接收一个指针：

template<class \_Ux>

explicit shared\_ptr(\_Ux \*\_Px){/\*……\*/}

（2）constexpr shared\_ptr(nullptr\_t) noexcept，非explicit

nullptr 是一个所谓“指针空值类型” 的常量。指针空值类型被命名为nullptr\_t,

typedef decltype(nullptr) nullptr\_t;

因此

std::shared\_ptr<int> p(new int)可以

std::shared\_ptr<int> p = new int错误 //参数为int\*的构造函数是explicit，无法隐式类型转换

std::shared\_ptr<int> p = nullptr可以 //发生nullptr\_t到shared\_ptr的隐式类型转换

shared\_ptr<int> func() { return nullptr; }可以，不需要写成return shared\_ptr<int>();

**空shared\_ptr和null shared\_ptr:**

non-empty null shared\_ptr：

* empty指shared\_ptr没有控制块，即shared\_ptr.use\_count()==0
* null指shared\_ptr.get()==nullptr

shared\_ptr无参构造、用nullptr构造，则没有控制块；

用(int\*)nullptr构造则有控制块，是non-empty null的shared\_ptr

{

std::shared\_ptr<int> ptr1;或std::shared\_ptr<int> ptr1(nullptr);

cout << ptr1.use\_count() << endl; //0

std::shared\_ptr<int> ptr2 = ptr1;

cout << ptr1.use\_count() << endl; //0

std::cout << std::boolalpha << (ptr1.get()==nullptr) <<endl; //true

}

{

std::shared\_ptr<int> ptr1(static\_cast<int\*>(nullptr));

cout << ptr1.use\_count() << std::endl; //1

std::shared\_ptr<int> ptr2 = ptr1;

cout << ptr1.use\_count() << std::endl; //2

cout << std::boolalpha << (ptr1.get() == nullptr) << endl; //true

}

**示例：**

std::shared\_ptr::reset()包含两个操作。当share\_ptr指针中有值的时候，调用reset()会使引用计数减1。当调用reset（new xxx())重新赋值时，share\_ptr首先是生成新对象，然后将旧对象的引用计数减1（当然，如果发现引用计数为0时，则析构旧对象），然后将新对象的指针交给share\_ptr保管

share\_ptr的引用计数变为0，析构管理的对象释放内存

通过原始指针析构对象释放内存，并不会改变引用计数

**void** main()

{

    {

        std::shared\_ptr<Person> p1(**new** Person(10));

        std::shared\_ptr<Person> p2 = std::make\_shared<Person>(2);

        p1.reset(**new** Person(3));//生成新对象，然后引用计数减1，引用计数为0，故析构Person(1)

        std::shared\_ptr<Person> p3 = p1; //Person(3)的引用计数为2

        p1.reset();  //Person(3)的引用计数为1

        p3.reset();   //Person(3)的引用计数为0,析构Person(3)

    }

        {  //shared\_ptr不能直接支持动态数组，可通过指定删除器实现

            std::shared\_ptr<Person> p2(**new** Person[2], [](Person\* p) {**delete**[] p; });

            std::cout << p2.get()->value << std::endl;  //注意访问方式，不能是p2[0].value

            std::cout << (p2.get() + 1)->value << std::endl;  //

        }

**shared\_ptr的删除器：**

std::shared\_ptr使用delete作为资源的默认销毁机制，也支持自定义删除器。这种支持有别于std::unique\_ptr。对于std::unique\_ptr来说，删除器类型是智能指针类型的一部分。对于std::shared\_ptr则不是

auto loggingDel = [](Widget \*pw){ delete pw;};

std::unique\_ptr<Widget, decltype(loggingDel) > upw(new Widget, loggingDel);

std::shared\_ptr<Widget> spw(new Widget, loggingDel);

std::shared\_ptr的设计更为灵活。考虑有两个std::shared\_ptr<Widget>，每个自带不同的删除器（比如通过lambda表达式自定义删除器）：

auto customDeleter1 = [](Widget \*pw) { … }; //自定义删除器，

auto customDeleter2 = [](Widget \*pw) { … }; //每种类型不同

std::shared\_ptr<Widget> pw1(new Widget, customDeleter1);

std::shared\_ptr<Widget> pw2(new Widget, customDeleter2);

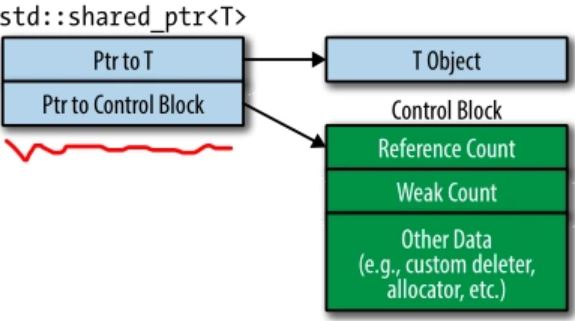
因为pw1和pw2有相同的类型，所以它们都可以放到存放那个类型的对象的容器中：

std::vector<std::shared\_ptr<Widget>> vpw{ pw1, pw2 };

它们也能相互赋值，也可以传入一个形参为std::shared\_ptr<Widget>的函数。但是自定义删除器类型不同的std::unique\_ptr就不行，因为std::unique\_ptr把删除器视作类型的一部分。

**shared\_ptr的控制块与析构：**

shared\_ptr 是引用计数型（reference counting）智能指针，几乎所有的实现都采用在堆（heap）上放个计数值（count）的办法。具体来说，shared\_ptr<Foo> 包含两个成员，一个是指向 Foo 的指针ptr，另一个是指向控制块的指针



“shared\_ptr::get()和operater->操作所使用的指针”，和指向“shared\_ptr管理的对象”的指针并不一定具有相同的类型和值。在从shared\_ptr<T>到shared\_ptr<U>的类型转换中，前一个指针从T\*转换成U\*，而后一个指针存储在控制块（control block）内，一旦创建就不会变动，与T和U都无关。控制块中记录了对象的析构方式（调用哪个析构函数，具体记录方式类似函数指针或std::function），因此std::shared\_ptr拷贝赋值、拷贝构造或shared\_ptr( const shared\_ptr<Y>& r, T \*ptr )构造时，由于控制块被共享，不影响所管理对象的析构函数被正确调用。

MSVC的std::shared\_ptr实现：

std::shared\_ptr<T>继承自\_Ptr\_base。

\_Ptr\_base的数据成员有两个，一个是引用计数\_Ref\_count\_base \_Rep，还有一个就是shared\_ptr::get()和operater->操作所使用的指针\_Ptr

类\_Ref\_count\_base不仅提供了引用计数等功能，还提供了一个非常重要的纯虚函数Destroy()。

当创建一个智能指针的时候，智能指针会创建一个\_Ref\_count\_base的派生类实例，即\_Ref\_count\_obj类对象。\_Ref\_count\_obj覆盖了父类的Destroy()函数，内部调用~Derived()

在智能指针引用计数为0时，由于\_Ref\_count\_base的Destroy()是虚函数，实际上\_Ref\_count\_obj的Destroy()被调用，执行~Derived()

int main()

{

//注意析构函数非虚

struct Base { int \* b; Base() { b = new int(9); } ~Base() { delete b; cout << "~Base\n"; } };

struct Derived : Base { int \* d; Derived() { d = new int(10); } ~Derived() { delete d; cout << "~Derived\n"; } };

std::shared\_ptr<Base> base(std::make\_shared<Derived>()); //不会造成内存泄漏

//运行结果：打印出~Derived、~Base

}

int main()

{

struct Base {~Base() { cout << "~Base\n"; } };

struct Derived : Base {~Derived() {cout << "~Derived\n"; } };

std::shared\_ptr<Base> b = std::make\_shared<Derived>();

std::shared\_ptr<Derived> d = std::dynamic\_pointer\_cast<Derived>(b); //编译报错，析构加上virtual后正常

}

std::shared\_ptr的构造函数之一：

template< class Y >

shared\_ptr( const shared\_ptr<Y>& r, T \*ptr ); //T为shared\_ptr<T>中的T

构造的shared\_ptr与r共享控制块，但指向与ptr相同，即get()返回等于ptr

class B {};

class C:public B {};

std::shared\_ptr<C> uc = std::make\_shared<C>();

std::shared\_ptr<B> ub = uc; //调用了shared\_ptr(const shared\_ptr<C>&, B\*)构造函数

cout << uc.use\_count() <<" "<<ub.use\_count() << endl;//2 2

//上述代码等价于：

std::shared\_ptr<C> sc1 = std::make\_shared<C>();

cout << sc1.use\_count() << endl; //1

std::shared\_ptr<B> sc2(sc1,(B\*)sc1.get());

cout << sc1.use\_count() << endl; //2

cout << sc2.use\_count() << endl; //2

体会下面这段代码，理解shared\_ptr( const shared\_ptr<Y>& r, T \*ptr )构造函数，C++对象的内存模型，编译器工作原理：

int main()

{

struct A { int\* p; A() :p(new int(3)) {} void show() { printf("A=%d\n", \*p); } };

struct B { int\* p; B() :p(new int(5)) {} void show() { printf("B=%d\n", \*p); }};

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

//std::shared\_ptr<B> b = a; //编译报错

std::shared\_ptr<B> b(a, (B\*)a.get());

b->show(); //3

}

指向子类对象的shared\_ptr指针可以转换为指向父类的shared\_ptr指针，且引用计数依然正确

class A { public:virtual void show() { cout << "A" << endl; } };

class B :public A { public:virtual void show() { cout << "B" << endl; } };

std::shared\_ptr<A> pa = std::shared\_ptr<B>(new B);//借助指向子类的智能指针拷贝构造

pa->show(); //B 如果不是show()不是virtual函数，则结果为A

**shared\_ptr<void>：**

shared\_ptr<void>是一个可以自动调析构函数的void\*指针

std::shared\_ptr<void> a = std::make\_shared<std::string>("hello world");

std::shared\_ptr<void> b = std::make\_shared<std::vector<int>>(10, 3);

//访问std::shared\_ptr<void>内部数据

std::shared\_ptr<std::vector<int>> sb = std::static\_pointer\_cast<std::vector<int>>(b);

std::cout << sb->size() << std::endl;

std::cout << b.use\_count() << std::endl; //2

std::cout << sb.use\_count() << std::endl; //2

b = a;

std::cout << a.use\_count() << std::endl; //2

std::cout << b.use\_count() << std::endl; //2

std::cout << sb.use\_count() << std::endl; //1

其中，

std::shared\_ptr<void> a = std::make\_shared<std::string>("hello world");等价于

std::shared\_ptr<void> a = std::static\_pointer\_cast<void>(std::make\_shared<std::string>("hello world"));

由于std::string\*可以直接隐式转换为void\*，所以std::static\_pointer\_cast<void>可以省略

但是std::shared\_ptr<std::vector<int>> sb = std::static\_pointer\_cast<std::vector<int>>(b);

不能写为std::shared\_ptr<std::vector<int>> sb = b;

因为c++中std::vector<int>\*不能隐式转换为void\*

void\* p = nullptr;

std::vector<int>\* ptr = p;

在c++中是不合法的

必须显式转换，std::vector<int>\* ptr = static\_cast<std::vector<int>\*>(p);

**enable\_shared\_from\_this：**

enable\_shared\_from\_this是一个模板类，定义于头文件<memory>，其原型为：

template< class T > class enable\_shared\_from\_this;

若一个类T继承std::enable\_shared\_from\_this<T>，则拥有了成员函数：shared\_from\_this和weak\_from\_this。

shared\_from\_this会创建指向当前对象的std::shared\_ptr却不创建多余控制块，无论在哪当你想在成员函数中使用std::shared\_ptr指向this所指对象时都请使用它

weak\_from\_this会返回std::weak\_ptr<T>对象

从内部来说，shared\_from\_this查找当前对象控制块，然后创建一个新的std::shared\_ptr关联这个控制块。设计的依据是当前对象已经存在一个关联的控制块。要想符合设计依据的情况，必须已经存在一个指向当前对象的std::shared\_ptr（比如调用shared\_from\_this的成员函数外面已经存在一个std::shared\_ptr）。

如果没有std::shared\_ptr指向当前对象（即当前对象没有关联控制块），行为是未定义的，shared\_from\_this通常抛出一个异常。

struct A : std::enable\_shared\_from\_this<A> // 注意：继承

{

std::shared\_ptr<A> getptr() { return shared\_from\_this(); }

~A() { std::cout << "A::~A() called" << std::endl; }

};

int main()

{

std::shared\_ptr<A> sp1(new A());

std::shared\_ptr<A> sp2 = sp1->getptr();

std::cout << sp1.use\_count() << std::endl; //2

std::cout << sp2.use\_count() << std::endl; //2

//如果上文getptr()函数实现是return std::shared\_ptr<A>(this)，则sp1和sp2虽然管理的是同一个A对象，但各自都有自己的控制块，引用计数都为1，都认为自己独占，释放资源时将导致错误

}

std::enable\_shared\_from\_this的使用场景：

在异步调用中，异步函数执行的时间点是无法确定的，然而异步函数可能会使用到异步调用之前就存在的变量。为了保证该变量在异步函数执期间一直有效，我们可以传递一个share\_ptr给异步函数，这样在异步函数执行期间share\_ptr所管理的对象就不会析构。如果异步函数是在类的成员函数里设置的一个回调函数，这种情况下，则没法直接获取一个share\_ptr对象，只能用enable\_shared\_from\_this。示例如下：

std::thread gThr;

struct A : std::enable\_shared\_from\_this<A> // 注意：继承

{

public:

void asynDo() {

auto func = [](std::shared\_ptr<A> sp){

std::this\_thread::sleep\_for(std::chrono::milliseconds(2000));

sp->realDo();

};

std::shared\_ptr<A> tmp= shared\_from\_this();

gThr =std::thread(func, tmp);

//如果第二个参数是std::shared\_ptr<A>(this)，则后文main函数执行崩溃

}

void realDo() { cout << "realdo,m=" << m << endl;; }

~A() { cout << "destroy" << endl; }

private: int m = 10;

};

int main()

{

{

std::shared\_ptr<A> gp1(new A());

gp1->asynDo();

}

cout << "ayncDo called" << endl;

gThr.join();

}

执行结果：

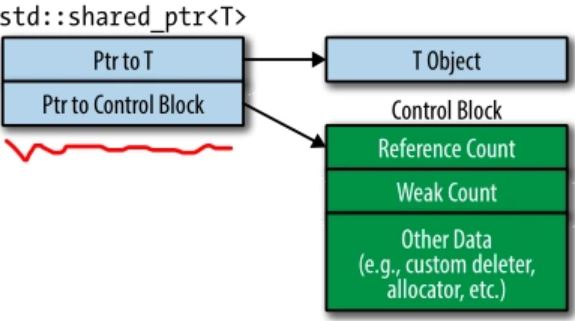
ayncDo called

realdo,m=10

destroy

**std::shared\_ptr的线程安全性**

shared\_ptr不是线程安全的



shared\_ptr的引用计数自身是原子化的，无锁。但因为 shared\_ptr 有两个数据成员，使得多线程读写（或同时写）同一个shared\_ptr对象时需要加锁。

std::shared\_ptr的线程安全性表述：

1. shared\_ptr引用计数的自增自减是原子操作
2. 同一个shared\_ptr对象可以被多线程同时读取
3. 同一个shared\_ptr的对象被多线程同时写或同时读写，非线程安全

比如std::thread的回调函数，是一个lambda表达式，其中引用捕获了一个shared\_ptr对象，即std::thread td([&sp1] () {....});

假设起初shared\_ptr<T> sp1指向的共享资源引用计数为2，在主线程中sp1修改指向为另一个shared\_ptr<T> sp2，即sp1=sp2，在lambda表达式线程中sp1修改指向为另一个shared\_ptr<T> sp3，即sp1=sp3。

期望执行完后，sp1起初所指的控制块引用计数减1，最后sp1指向sp2或sp3。

但shared\_ptr修改指向包含为4步：

(1)sp原先指向的引用计数的值要减去1，

(2)other\_sp指向的引用计数值要加1，

(3)数据指针修改为指向other\_sp的数据，

(4)控制块指针修改为指向other\_sp的控制块，

这几步操作加起来并不是一个原子操作。

因此可能出现，sp1=sp2时，sp1所指控制块引用计数减1，还未修改sp1控制块指针，又执行了sp1=sp3，sp1所指控制块引用计数又减1，引用计数降为0，资源释放，出现问题。

1. 多线程代码操作的不是同一个shared\_ptr的对象，但管理的对象是同一份，线程安全。这里指的是管理的数据是同一份，而shared\_ptr不是同一个对象。比如多线程回调的lambda的是按值捕获的对象，std::thread td([sp1] () {....});
2. 所管理数据的线程安全性。<T>类型的接口如read、write等非线程安全，那在不同线程中通过shared\_ptr（这些shared\_ptr管理的是同一份共享资源）同时调这些接口也就非线程安全。

// thread A

shared\_ptr<**int**> p2(p); // reads p

// thread B

shared\_ptr<**int**> p3(p); // OK, multiple reads are safe

//一个shared\_ptr实体不允许被多个线程同时读写

// thread A

p = p3; // reads p3

// thread B

p3.reset(); // writes p3; undefined, simultaneous read/write

//一个shared\_ptr实体不允许被多个线程同时写

std::shared\_ptr<int> x = std::make\_shared<int>(8);

std::shared\_ptr<int> y = std::make\_shared<int>(9);

std::shared\_ptr<int> z;

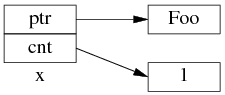
// thread A

z=x;

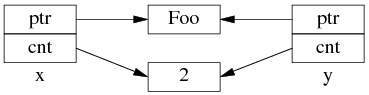
// thread B

z=y; //由于线程交织，可能最终z的ptr指向y管理的数据，z的cnt指向x的引用计数值

shared\_ptr<Foo> x(new Foo); 对应的内存数据结构。

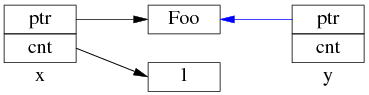


如果再执行 shared\_ptr<Foo> y = x; 那么对应的数据结构如下。

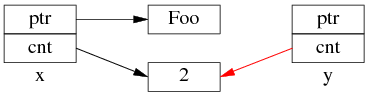


但是 y=x 涉及两个成员的复制，这两步拷贝不会同时（原子）发生。

中间步骤 1，复制 ptr 指针：



中间步骤 2，复制 ref\_count 指针，导致引用计数加 1：



**std::wake\_ptr**

将一个weak\_ptr绑定到一个shared\_ptr对象，不会改变shared\_ptr的引用计数。一旦最后一个所指向对象的shared\_ptr被销毁，所指向的对象就会被释放，即使此时有weak\_ptr指向该对象，所指向的对象依然被释放。

 #include <iostream>

 #include <memory>

*void* observe(std::weak\_ptr<*int*> *ptr*)

 {

     if(std::shared\_ptr<*int*> obs=*ptr*.lock())

     {

         printf("able to lock, val=%d, use cnt=%ld\n",\*obs,obs.use\_count());

         return;

     }

     printf("unable to lock\n");

 }

*int* main()

 {

     std::weak\_ptr<*int*> weak;

     observe(weak); //unable to lock

     {

*auto* shared=std::make\_shared<*int*>(23);

         weak=shared;

         observe(weak); //able to lock, val=23, use cnt=2

         printf("use cnt=%ld\n",shared.use\_count()); //use cnt=1

     }

     observe(weak); //unable to lock

 }

# 函数指针：

**void** myFunc(**char**\* name, **int** x); //函数

**void** (\*funcPtr)(**char**\*, **int**); //函数指针

cout << typeid(myFunc).name() << endl; //void \_\_cdecl(char \*, int)

cout << typeid(&myFunc).name() << endl; //void (\_\_cdecl\*)(char \*,int)

//不能有表达式&&myFunc

cout << typeid(\*myFunc).name() << endl; //void \_\_cdecl(char \*,int)

cout << typeid(\*\*\*myFunc).name() << endl; //void \_\_cdecl(char \*,int)

对全局函数：

* &函数名 为函数指针
* 函数名可隐式转换为函数指针，除非作为sizeof 或者单目 & 操作符的操作数。因为作为sizeof操作数时函数名是函数，因此sizeof(函数名)不合法
* \*函数名 结果仍为函数名
* 函数调用为myFunc("ghi", 3)。由于解引用函数指针为函数名，因此也可以(\*funcPtr)("ghi", 3)。但也可以funcPtr("ghi", 3)
* 指向全局函数的函数指针

int happen() {……}

double happen(double x) {……}

int(\*p)() = &happen; //也可以没有&，此时隐式转换为函数指针

cout<<happen()<<endl;

double(\*p2)(double) = &happen; //也可以没有&

cout << happen(0.099) << endl;

也可以借助typedef定义函数指针

typedef double(\*p2Func)(double);

p2Func ph = &happen;

cout << ph(0.0666) << endl;

对类的成员函数，函数名不能隐式转换为函数指针

* 类的成员变量是一个函数指针，要指向类外部的一个函数：

**class** A

{

**private**:

**double**(\*fun)(**double**);

**public**:

    A(**double**(\*p)(**double**)) :fun(p) {}

    auto getFun()**const** ->**double**(\*)(**double**) { **return** fun; }  //函数返回类型为函数指针的正确写法

}

A a(&happen);

a.getfun()(0.99);

* 类外部的一个函数指针，指向类的一个成员函数：

void(A::\*pfun)() = &A::show; //符号&不能省略

(dog.\*pfun)();

或者：

A\* ptr = &dog;

(ptr->\*pfun)();

* 函数指针数组

int (\*s[10])(int) 函数指针数组，每个指针指向一个int func(int param)的函数。

# std::function()：

C++中有如下几种可调用对象：函数、函数指针、lambda表达式、bind对象、重载了()运算符的类的对象

函数对象：像一个函数的class对象。这个类定义了应用操作符operator()

函数对象可存储先前调用结果，保存在某个数据成员中。它可以在使用前保存，使用后你想取出都没问题。而普通函数则需要将先前调用的结果存储在全程或者本地静态变量中，但是全程或者本地静态变量有某些我们不愿意看到的缺陷

std::function类模板是一种通用的函数包装器，它可以容纳所有可调用对象（[Callable](http://en.cppreference.com/w/cpp/concept/Callable)），包括**函数**、**函数指针**、**Lambda表达式**、**bind表达式**、成员函数、函数对象。通过std::function可以储存、拷贝或调用Callable对象

#include <iostream>

#include <functional>

**double** f(**int** x, **char** y, **double** z) {

**return** x + y + z;

}

**void** print\_num(**int** num) {

    std::cout << num << std::endl;

}

**struct** Dog {

**int** id;

**explicit** Dog(**int** id) : id(id) {}

**void** print\_add(**int** i) **const** {

        std::cout << id + i << std::endl;

    }

};

**class** PrintString {

**public**:

**void** operator()(std::string&& s) **const** {

        std::cout<< s.data() << std::endl;

    }

};

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

    // common function

    std::function<**void**(**int**)> func\_display\_num = print\_num;  **// 亦可&print\_num**

    func\_display\_num(9);

    // common function

    std::function<**double**(**int**, **char**, **double**)> func\_display = f;  **// 亦可&f**

    std::cout << func\_display(3, 'a', 1.7) << "\n";

    // lambda expression

    std::function<**void**(**const** **char**\*)> lbd\_dsp\_str = [](**const** **char** \*s) {std::cout << s << std::endl; };

    lbd\_dsp\_str("Scala");

// bind expression

//auto func\_num\_bind也可换作std::function<double(int)> func\_num\_bind

    auto func\_num\_bind = std::bind(&f, std::placeholders::\_1, 'a', 2.4); //亦可f

    std::cout << func\_num\_bind(24) << "\n";

    // function object函数对象

    std::function<**void**(std::string&&)> func\_obj\_print = PrintString();

    func\_obj\_print("C++ 17 Nice!");

    // member function

    const Dog dog(2424);

    std::function<void(const Dog&, int)> func\_mem\_display\_num = &Dog::print\_add;

func\_mem\_display\_num(dog, 24);

void(Dog::\*func)(int)const = &Dog::print\_add;

(dog.\*func)(24);

}

bind表达式：

int myFunc(char\* name, int value) { std::cout << name<<"'s age is " <<value<< std::endl; return value; }

std::function<int(int)> func5 = std::bind(&myFunc, const\_cast<char\*>("xyLi"), std::placeholders::\_1);

func5(23); //xyLi's age is 23

占位符std::placeholders，其定义如下：其中\_1, \_2, \_3是未指定的数字对象，用于function的bind中。 \_1用于代替bind函数中的第一个参数， \_2用于代替bind函数中的第二个参数，以此类推

举个例子：

        void function(arg1,arg2,arg3,arg4,arg5)

        {

            //do something

        }

        auto g = bind(function,a,b,\_2,c,\_1);

        新的函数对象：g

        被调用函数：function

        当调用函数对象g时候，函数对象g会调用function函数，并把其参数传给function函数，g的第一个参数会传给function的持有占位符\_1的位置，即arg5。第二个参数会传给function的持有占位符\_2的位置，即arg3。

# 运算符重载;

CData& operator ++ ()//前缀递增运算符重载

CData operator ++(int) //后缀递增运算符，故按值返回对象由于编译器必须能够识别出前缀自增与后缀自增，故人为规定了用一个 int 区分

operator int() //转换运算符重载 使得int x=obj;可执行

operator bool()

operator double ()

函数对象类中的重载的函数调用运算符

有参的：int operator()(int x, int y){return x + y;}

无参的：int operator()(){return 100;}

双目运算符+ :CDate operator + (int n)

# lambda表达式：

lambda表达式可以转化为指针，但指针不能转化为lambda表达式，因此decltype(cal) count = pfunc;报错

auto cal = [](int x, int y) {return x + y; };

cout << std::is\_same<decltype(cal), int(int, int)>::value << endl; //false

cout << typeid(cal).name() << endl; //class <lambda\_d3eebd8caad516e7e08d61c0ba72adc1>

auto cal = [](int x, int y) {return x + y; };

cout << typeid(cal).name() << endl;//class <lambda\_d3eebd8caad516e7e08d61c0ba72adc1>

int(\*pfunc)(int x, int y) = cal;

cout << pfunc(55, 10) << endl; //65

decltype(cal) compute = cal; //compute是个lambda表达式

cout << typeid(compute).name() << endl;//class <lambda\_d3eebd8caad516e7e08d61c0ba72adc1>

cout << compute(54, 6) << endl; //60

对于按值方式传递的捕捉列表，其传递的值在lambda 函数定义的时候就已经决定了。 而按引用传递的捕捉列表变表，其传递的值则等于lambda 函数调用时的值。原因：lambda用函数对象实现，按值捕获列表等价于构造函数的按值传递参数；按引用捕获等价于类的成员为引用类型，构造函数为按引用传递参数。

int age = 100;

class AY

{

public:

AY(int& age):age\_(age) {}

void operator()() const //类的成员变量是引用类型，则const成员函数可以修改之；类的成员变量不是引用类型，则const成员函数不可以修改之

{

age\_++; //虽然为const成员函数，但不报错，因为age\_为引用类型。

}

private:

int& age\_;

};

AY ay(age);

ay();

cout << age<< endl; //101

按值捕获参数，没有mutable关键字，则不能在lambda体内修改捕获参数的值；加上mutable关键字，则可以修改按值捕获的变量。原因在于现有 C++11 标准中的 lambda函数实际是一个class的 ()运算符重载函数，且是常量成员函数。

返回类型 类名::operate()(参数列表)const{}

int x = 10;

int y = 100;

auto cal = [x,y]()mutable {x++; return x + y; }; //没有mutable则报错，因为

cout << cal() << endl; //111

[&]。函数体内可以使用Lambda所在作用范围内所有可见的局部变量（包括Lambda所在类的this）

lambda 表达式只能捕获其父作用域（即父作用域、爷作用域……）automatic(具有自动存储期)变量

lambda不能捕获具有静态存储持续时间的变量，如全局变量、局部static变量，但可以直接使用

**class** C

{

**public**:

**static** **const** **int** ele = 30;

**int** cone=1;

};

C c\_out;

**int** height = 170;

**int** main()

{

**static** **int** score = 99;

**int** age = 20;

    C c\_in;

    cout << c\_in.ele << endl;

**do**

    {

**int** weight = 45;

        auto func = [c\_in]()//不能捕获全局变量[height]、[c\_out]、局部static变量[score]；不可以捕获[用户自定义类型的对象名.数据成员]，如不可以[c\_in.cone]或[c\_in.ele]

        {

            //cout << weight << endl; error

            //cout << age << endl; //error

            cout << height << " " << score << endl;

            cout << C::ele << endl;

            cout << c\_in.ele << endl;

        };

        func();

**class** Myfunc

        {

**public**:

**void** operator()()**const**

            {

                //cout << weight << endl; error

                //cout << age << endl; error

                cout << height << " " << score << endl;

                cout << C::ele << endl;

            }

        };

        Myfunc()();

    }**while** (**false**);

}

# C++左值右值：

（1）

C++关于左值性的定义是这样的: 一个表达式不是左值就是右值

函数调用表达式，返回左值引用的都是左值，其余都是右值(包括按值返回自定义类的对象、返回右值引用)。就是说，非左值引用返回的函数调用都是右值

数组名属于左值，不是右值，而且是一个不可修改的左值

右值包括：

* 一些运算表达式产生的临时变量值，比如 1+3 ；
* 字面值常量，比如：true、4.36、’a’。
* lambda 表达式是右值

auto&& fn = [](int x) { return x \* x; }; //lambda表达式是右值,可以用来初始化右值引用

cout << std::is\_rvalue\_reference<decltype(fn)>::value << endl; //1

* std::move()的返回值是右值。
* static\_cast<>的返回值可能是左值，可能是右值。static\_cast<int&>返回左值，其余的转换包括static\_cast<int&&>返回的都是右值
* 返回A&&类型的函数

int a=10;

++a=12;

cout<<a<<endl<<endl;//++a是左值表达式，a++是右值表达式

int b=120;

int& p1=b;

//int& p2=3; 编译报错，非常量左值引用不能绑定右值，可修改为const int& p2=3

const int& p3=b;

（2）&后的表达式可能是右值

函数按值返回自定义对象，返回的对象是右值，返回对象的数据成员也是右值，但都可以取地址

#include<iostream>

**using** **namespace** std;

**class** Test { **public**:  **int** a[10]; };

Test fun(Test\* p1) { **return** \*p1; }

**void** main()

{

    Test obj;

**int** (\*q)[10]=&(fun(&obj).a); // 正确， fun( &T ).a是右值

    cout << q << endl;  //000000F53CCFF918

    cout << &fun(&obj) << endl;  //000000F53CCFF998

}

(3) 右值可以出现在了等号=左边

**class** Rational

{

**public**:

    Rational(**double** x) :x\_(x) {}

**double** x\_;

};

Rational operator\*(**const** Rational& lhs, **const** Rational& rhs)

{ **return** Rational(lhs.x\_\*rhs.x\_);}//运算符重载函数，但不是类的成员函数

**int** main()

{

    cout << (Rational(3.0)\*Rational(5.0) = Rational(18.0)).x\_ << endl;  //18

    //按值返回自定义类型，返回的右值可以调用成员函数，包括赋值运算符重载函数

    //右值可以出现在了等号=左边

}

虽然c++标准允许 f()(因为返回一个 class type的对象) 这个表达式 来调用operator=这个成员函数来更改 f() 表达式返回临时对象的状态，但c++标准还是明确规定了f() 表达式是一个右值

1) 对于内置的类型，右值是不可被修改的(non-modifiable)，也不可被 const, volatile 所修饰(cv-qualitification ignored)

2) 对于自定义的类型(user-defined types)，右值却允许通过它的成员函数进行修改。

自定义类型允许有成员函数，而通过右值调用成员函数是被允许的，但成员函数有可能不是 const 类型，因此通过调用右值的成员函数，也就可能会修改了该右值，done!

# C++拷贝复制函数：

对于一个类A，复制构造函数的形式：

* A(A& obj)
* A(const A& obj)
* A(volatile A& obj)
* A(const volatile A& obj)

如果类中没有自定义复制构造函数，且需要调用拷贝构造函数，则编译器自动产生一个默认的复制构造函数，形式为A(A& )或A(const A&)，由编译器根据上下文决定选择哪一个

默认复制构造函数是浅拷贝

复制构造函数调用时机：

* 函数按值返回对象
* 对象按值传递的方式传入函数参数
* 对象需要通过另一个对象进行初始化，A a; A obj=a；

**class** A

{

**public**:

    A() = **default**;

    A(**int** a, **int** b, **int** c) :a\_(a), b\_(b), ptr\_(**new** **int**(c)) { cout << "构造" << endl; }

    ~A() { **if** (ptr\_ != nullptr) **delete** ptr\_; cout << "析构" << endl; }

    A(**const** A& rhs) :a\_(rhs.a\_), b\_(rhs.b\_) { **int**\* temp = **new** **int**(\*(rhs.ptr\_)); ptr\_ = temp; cout << "cop" << endl; }

    A(A&& rhs) :a\_(rhs.a\_), b\_(rhs.b\_), ptr\_(rhs.ptr\_) { rhs.ptr\_ = nullptr; cout << "mov" << endl; }

    A& operator=(A& rhs) //小心自我赋值，小心new申请内存失败

    {

**if** (**this** == &rhs) **return** \***this**;

**int**\* temp = **new** **int**(\*(rhs.ptr\_));

**delete** ptr\_;

        ptr\_ = temp;

**return** \***this**;

    }

    A& operator=(A&& rhs)  //小心自我赋值

    {

        cout << "mov assign" << endl;

**if** (&rhs == **this**) **return** \***this**;

**delete** ptr\_;

        ptr\_ = rhs.ptr\_;

        rhs.ptr\_ = nullptr;

        a\_ = rhs.a\_; b\_ = rhs.b\_;

**return** \***this**;

    }

**void** change(**int** a) { a\_ = a; }

**void** show() **const** { cout << a\_ << " " << b\_ << endl; }

**int** geta() **const** { **return** a\_; }

**int** getb() **const** { **return** b\_; }

**const** **int**\* getptr() **const** { **return** ptr\_; }

**private**:

**int** a\_;

**int** b\_;

**int**\* ptr\_;

};

# 右值引用&移动语义

* 右值引用、左值引用都是引用，不占内存、必须在定义时就初始化
* 右值引用只能绑定右值，非常量右值引用只能绑定非常量右值，常量右值引用既可以非绑定常量右值，又可以绑定常量右值
* 非常量左值引用只能绑定非常量左值
* 常量左值引用既可以左值（包括常量左值、非常量左值），又可以绑定右值（包括常量右值、非常量右值）
* 左值引用是个左值，右值引用可能是左值，也可能是右值。C++规定：具名的右值引用是左值(函数形参是右值引用，在函数体中该参数是左值)，不具名的右值引用为右值(函数返回右值引用类型，或static\_cast<A&&>()返回值)

(1)右值引用延长右值的生存周期

class A {};

A retval() { …… }

A&& rr = retval();

retval()函数返回的右值在表达式语句结束后，其生命也就终结了（通常我们也称其具有表达式生命期）， 而通过右值引用的声明，该右值又“ 重获新生”，其生命期将与右值引用类型变量 a 的生命期一样。只要 a 还“活着”，该右值临时量将会一直“存活” 下去

(2)

如果函数按值返回，return语句又直接返回了一个栈上的左值对象（输入参数除外）时，标准要求优先调用移动构造函数，如果不符再调用拷贝构造函数。尽管v是左值，仍然会优先采用移动语义，返回vector<string>从此变得云淡风轻。

vector<string> str\_split(const string& s) {

vector<string> v;

// ...

return v; // v是左值，但优先移动，不支持移动时仍可复制。

}

(3)非常量左值引用不能绑定右值，但VS2015有bug可以将非常量左值引用绑定到右值

VS2015下：

**struct** STRUCT

{

  STRUCT(**int** a){}

};

**void** func1(**int**& i){}

**void** func2(STRUCT& s){}

**void** main()

{

  //func1(int(56));  // ERROR: C2664

  func2(STRUCT(65)); // OK: anonymous object is created then assigned to a reference

}

编译器bug，VS不是标准C++编译器

在gcc下，func2(STRUCT(65))编译报错，非常量左值引用不能绑定右值，匿名对象为右值

同理STRUCT& r= STRUCT(65)，在VS下不会报错，在GCC下报错

(4)有移动语义后，swap()函数

**template**<**class** T>

**void** swap(T& a, T&b)

{

    T temp(std::move(a));

    a = std::move(b);

    b = std::move(temp;)

}

对于左值，如果我们明确放弃对其资源的所有权，则可以通过std::move()来将其转为右值引用。std::move()实际上是static\_cast<T&&>()的简单封装。



函数形参是void func(int&& x),则实参必须是右值，更准确地说是非常量右值

tempalate<class T> void func(T&& x), 模板函数，则实参既可以是右值，又可以是左值。因为此时的模板类型T由实参推导，实参如果是左值，则模板参数被推导为T&，模板函数即(T& && x)，即(T& x)；实参如果是右值，则模板参数被推导为T&&，模板函数即(T&& && x)，即(T&& x)。无论实参是左值还是右值，最后模板函数被实例化后，形参恰好可接受实参

总结：

可能有人已经习惯了一看到 T&& 就以为这是右值引用，这确实很容易误解，但事实是，T&& 为右值引用只有当 T 为一个具体的类型时才成立，而如果 T 是推导类型时（如模板参数, auto 等）这就不一定了

未定的引用类型T && v被一个右值初始化后变成了一个右值引用、被一个左值初始化后变成了一个左值引用



右值引用的提出还解决另一个问题：完美转发 (perfect forwarding)，转发问题针对的是模板函数

**class** Stu

{

**public**:

    Stu() { p = **new** **int**(99);  cout << "构造" << endl; }

    //Stu(Stu& a):p(new int(\*a.p)) { cout << "Stu(Stu& a)" << endl; }

    Stu(**const** Stu& a):p(**new** **int**(\*a.p)) { cout << "Stu(const Stu& a)" << endl; }

    Stu(Stu&& ra):p(ra.p) { ra.p = nullptr; cout << "移动构造" << endl; }

    ~Stu() { **delete** p; cout << "析构" << endl; }

**void** showAge() **const**{cout << "age is:" << \*p << endl;}

**private**:

**int**\* p;

};

Stu e = d;

Stu&& rre = d;编译错误，无法将右值引用绑定到左值

int a = 10;

int&& rra = a; 编译错误，无法将右值引用绑定到左值

**void** showStuInfo(Stu&& s)

{

    cout << "info is:" << endl;

    s.showAge();

}

showStuInfo(Stu());正确

Stu e; showStuInfo(e);编译错误，无法将右值引用绑定到左值，可以使用move将左值转化为右值引用，该右值引用是不具名的，因此自身是个右值，再传递给函数，即showStuInfo(move(e));

但如果不是Stu，而使用模板函数则没有问题

template<class T>

void showStuInfo(T&& s)

{

cout << "info is:" << endl;

s.showAge();

}

Stu e;

showStuInfo(e);正确，传递左值，模板函数的参数类型被推导为T&，T& && s，即T& s，因此模板函数的形参为void showStuInfo(T& s)

如果一个类定义了自己的拷贝构造函数或拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。

编译器合成的默认移动构造函数，单纯地调用该类下所有成员的移动构造函数。

举例：

class A{std::string str1,str2;};

这个类编译器生成的默认移动构造函数大致如下：

A::A(A&& a): str1(std::move(a.str1),str2(std::move(a.str2)){}

//这里的std::move是可以省略的，这里只是单纯的为了方便理解加上的

std::string有移动构造函数

**class** Stu

{

**public**:

    Stu() :name\_(""), id\_(0), age\_(nullptr) {}

    Stu(string name, **int** id, **int** age) :name\_(name), id\_(id)

    {

        age\_ = **new** **int**(age);

    }

    Stu(**const** Stu& s) :name\_(s.name\_), id\_(s.id\_)

    {

        cout << "Stu copy constr" << endl;

        age\_ = **new** **int**(\*s.age\_);

    }

    Stu(Stu&& s) :name\_(s.name\_), id\_(s.id\_)

    {

        cout << "Stu move constr" << endl;

        age\_ = s.age\_;

        s.age\_ = nullptr;

    }

    ~Stu() { **delete** age\_; }

**void** show()

    {

        cout << name\_ << "'s id is " << id\_ << ", age is:" << \*age\_ << endl;

    }

    string name\_;

**long** id\_;

**int**\* age\_;

};

**class** A

{

**public**:

**int** \*p;

    string str;

    Stu stu;

    A() :p(**new** **int**), str("Fuck"), stu("xyLi", 2016, 10) {}

    //没有自定义的拷贝构造函数或拷贝赋值运算符或者析构函数，因此合成默认移动构造

};

**int** main()

{

    A a;

    cout << a.p << " " << a.str << " " << a.stu.name\_ << " " << a.stu.id\_ << " " << a.stu.age\_ << endl;;

    A b(std::move(a));

    cout << a.p << " " << a.str << " " << a.stu.name\_ << " " << a.stu.id\_ << " " << a.stu.age\_ << endl;;

    cout << b.p << " " << b.str << " " << b.stu.name\_ << " " << b.stu.id\_ << " " << b.stu.age\_ << endl;

}

/\*

运行结果：

000001950F9102C0 Fuck xyLi 2016 000001950F910400

Stu move constr

000001950F9102C0 xyLi 2016 0000000000000000

000001950F9102C0 Fuck xyLi 2016 000001950F910400

\*/

# 引用折叠与完美转发：

C++中，禁止 “对引用的引用”，如

int x = 10;

int&rx = x;

int& & rrx = rx; //报错

但编译器在特殊情况下，如实例化函数模板时，会使用对引用的引用

有两种类型的引用（左值和右值），所以有4种可能的对引用的引用的组合（左值对左值，左值对右值，右值对左值，右值对右值）

A为内置类型或用户自定义类型class A{};则引用折叠规则为：如果两个引用中有一个是左值引用，那么折叠的结果是一个左值引用；否则（即两个都是右值引用），折叠的结果是一个右值引用。

* 左值引用的左值引用 A& & 折叠为A&
* 左值引用的右值引用 A& && 折叠为A&
* 右值引用的左值引用 A&& & 折叠为A&
* 右值引用的右值引用 A&& && 折叠为A&&
* 非引用的左值引用 A & 为A&
* 非引用的右值引用 A && 为A&&

对模板函数template<typename T> void func(T&& param){}

传入的实参为左值，则模板类型参数T被推导为左值引用A&；传入的实参为右值，则模板类型参数T被推导为非引用类型A

因此，传入实参为左值时，为A& &&，再按照引用折叠规则折叠为A&;

传入实参为右值时，为A &&，再按照引用折叠规则折叠为A&&;

**template**<**class** T>

**void** g(T && value)

{

     cout << std::is\_lvalue\_reference<T>::value <<" "<<

        std::is\_rvalue\_reference<T>::value <<" "<<

        std::is\_object<T>::value << endl;

}

**int** main()

{

    g(42);  //0 0 1

**int** x = 10; g(x);   //1 0 0

**int**&& y = 10; g(y); //1 0 0  具名的右值引用为左值，因此模板参数T推导为左值引用

    g(std::move(x));   //0 0 1  std::move(x)等价于std::static\_cast<int&&>，不具名的右值引用为右值，因此模板参数T推导为非引用int

}

std::is\_lvalue\_reference<T>::value判断T是否为左值引用

std::is\_rvalue\_reference<T>::value判断T是否为右值引用

std::is\_object<T>::value判断T是否为非引用

借助引用折叠在函数模板中实现完美转发：

template<class T>

void PerfectForward(T &&t) //转发函数

{

RunCode(static\_cast<T&&>(t)); //目标函数

}

传入实参为左值，则T被推导为int&，则t类型为int&，T&&为int& &&为int&，这时static\_cast<T&&>(t)返回的是左值

传入实参为右值，则T被推导为int，则t类型为int&&，T&&为int &&为int&&，这时static\_cast<T&&>(t)返回的是右值

完美转发的含义：

* 以引用方式传递，不会拷贝构造、移动构造产生临时对象，即转发时没有额外开销
* 传给转发函数的实参是左值，则传给目标函数的实参是左值；传给转发函数实参的是右值，则传给目标函数的实参是右值

引用折叠用于四种情况：

(1)模板实例化

(2)auto类型推导。

int a;

auto&& x = 1; //1为右值，auto被推导为int，x类型为int &&，即int&& x=1;

auto& y = a; //a为左值，即int& x=a;

auto&& z = a; //a为左值，auto被推导为int&，z类型为int& &&，折叠为int&，即int& z=a;

(3) 在typedef创建或者评估期间，出现了对引用的引用，引用折叠会出面消除它们

template<typename T>

class Widget {

public:

typedef T&& RvalueRefToT;

...

};

然后假如我们用一个左值引用来实例化Widget：

Widget<int&> w;

在Widget中用int&代替T，typedef变成这样：

typedef int& && RvalueRefToT;

引用折叠后：

typedef int& RvalueRefToT;

**模板类型推导：**

考虑像这样一个函数模板：

template<typename T>

void f(ParamType param);

它的调用看起来像这样:

f(expr); //使用表达式调用f

在编译期间，编译器使用expr进行两个类型推导：一个是针对T的，另一个是针对ParamType的

**情景一：ParamType是值类型**

当ParamType既不是指针也不是引用时，我们通过传值（pass-by-value）的方式处理：

template<typename T>

void f(T param);

1. 值类型的形参，无论传递什么，param都会成为它的一份拷贝，是一个完整的新对象
2. 如果expr的类型是一个引用，忽略这个引用部分
3. 如果忽略expr的引用性（reference-ness）之后，expr是一个const，那就再忽略const。
4. 只有值类型的形参才会忽略const，对于reference-to-const和pointer-to-const形参来说，expr的常量性（const ness）在推导时会被保留

int x = 27;

const int cx = x;

const int & rx = cx;

f(x); //T和param的类型都是int

f(cx); //T和param的类型都是int

f(rx); //T和param的类型都是int

**情景二：ParamType是一个指针或引用，但不是万能引用**

模板使用T&类型为形参：

template<typename T>

void f(T& param); //param是一个引用

int x = 27;

const int cx = x;

const int& rx = x;

f(x); //T是int，param的类型是int&

f(cx); //T是const int，param的类型是const int&

f(rx); //T是const int，param的类型是const int&

1. 当传递一个const对象给一个引用类型的形参时，期望对象保持不可改变性，也就是说，形参是reference-to-const的。这也是为什么将一个const对象传递给以T&类型为形参的模板安全的：对象的常量性（const ness）会被保留为T的一部分
2. 在第三个例子中，注意即使rx的类型是一个引用，T也会被推导为一个非引用 ，这是因为rx的引用性（reference-ness）在类型推导中会被忽略。
3. T&为模板形参时，实参不能是右值（非常量左值引用不能绑定右值，常量左值引用可以绑定右值）

模板使用const T&类型为形参：

template<typename T>

void f(const T& param); //param现在是reference-to-const

int x = 27;

const int cx = x;

const int& rx = x;

f(x); //T是int，param的类型是const int&

f(cx); //T是int，param的类型是const int&

f(rx); //T是int，param的类型是const int&

如果param是一个指针（或者指向const的指针）而不是引用，情况本质上也一样：

template<typename T>

void f(T\* param); //param现在是指针

int x = 27;

const int \*px = &x; //px是指向作为const int的x的指针

f(&x); //T是int，param的类型是int\*

f(px); //T是const int，param的类型是const int\*

**情景三：ParamType是一个万能引用**

template<typename T>

void f(T&& param); //param是一个万能引用类型

int x = 27;

const int cx = x;

const int & rx = cx;

f(x); //x是左值，所以T是int&，param类型也是int&

f(cx); //cx是左值，所以T是const int&，param类型也是const int&

f(rx); //rx是左值，所以T是const int&，param类型也是const int&

f(27); //27是右值，所以T是int，param类型就是int&&

当万能引用被使用时，类型推导会区分实参是左值还是和右值，但是对非万能引用时不会区分

传入实参为左值时，ParamType被推导为左值引用;

传入实参为右值时，ParamType被推导为左值引用;

如果expr是左值，T和ParamType都会被推导为左值引用。这是模板类型推导中唯一一种T被推导为引用的情况。

**auto类型推导：**

auto类型推导可转化为模板类型推导

当一个变量使用auto进行声明时，auto扮演了模板中T的角色，变量的类型说明符扮演了ParamType的角色

const int x=10;

auto& y=x;

即

template<class T>

void f(T& param);

const int x = 10;

f(x)

auto类型推导与模板类型推导的唯一区别在于：当用auto声明的变量使用花括号进行初始化，auto类型推导推出的类型则为std::initializer\_list。如auto y = { 10 };//y被推导为std::initializer\_list<int>类型。

当使用auto声明的变量使用花括号的语法进行初始化的时候，会推导出std::initializer\_list<T>的实例化，但是对于模板类型推导这样就行不通：

auto x = { 11, 23, 9 }; //x的类型是std::initializer\_list<int>

template<typename T> //带有与x的声明等价的形参声明的模板

void f(T param);

f({ 11, 23, 9 }); //错误！不能推导出T

auto用于函数返回值，或者lambda函数的形参声明中时，这些情况下auto实际上使用**模板类型推导**的那一套规则在工作，而不是auto类型推导，所以说下面这样的代码不会通过编译：

auto createInitList()

{

return { 1, 2, 3 }; //错误！不能推导{ 1, 2, 3 }的类型

}

或

std::vector<int> v;

auto resetV =[&v](const auto& newValue) { v = newValue; }; //C++14

resetV({ 1, 2, 3 }); //错误！不能推导{ 1, 2, 3 }的类型

设计一个函数，对容器c实现c[i]元素访问：

template<typename Container, typename Index>

auto authAndAccess(Container&& c, Index i)->decltype(c[i])

{

return c[i];

}

上述函数名前的auto仅仅指示会在函数形参列表后面使用一个”->“符号指出函数的返回类型，实际返回类型由decltype(c[i])推导，这样的写法略有繁琐

template<typename Container, typename Index>

auto authAndAccess(Container&& c, Index i)

{

return c[i];

}

上述函数返回类型为auto，编译器使用模板推导规则，auto被推导为值类型，返回的是c[i]的一个副本（即使c[i]为引用类型），而期望的是返回传入的容器的[i]位置元素的引用，因此正确写法为：

template<typename Container, typename Index>

decltype(auto) authAndAccess(Container&& c, Index i)

{

return c[i];

}

第一次看见decltype(auto)可能觉得非常的矛盾（到底是decltype还是auto？），实际上我们可以这样解释它的意义：auto说明符表示这个类型将会被推导，decltype说明decltype的规则将会被用到这个推导过程中

调用上述函数时，传入的容器实参无论是左值还是右值，在函数authAndAccess体内，c都是左值，因此可以继续优化，实现完美转发：

template<typename Container, typename Index>

decltype(auto) authAndAccess(Container&& c, Index i)

{

return std::forward<Container>(c)[i];

}

# C++ 返回引用：

不能返回局部对象的左值引用。当函数执行完毕时，将释放分配给局部对象的存储空间。此时，对局部对象的引用就会指向不确定的内存。

也不能返回局部对象的右值引用。函数返回，栈空间一定销毁，右值引用不能指向已销毁的对象

返回左值引用，要求在函数的参数中，包含有以引用方式或指针方式存在的，需要被返回的参数。此时返回是左值。

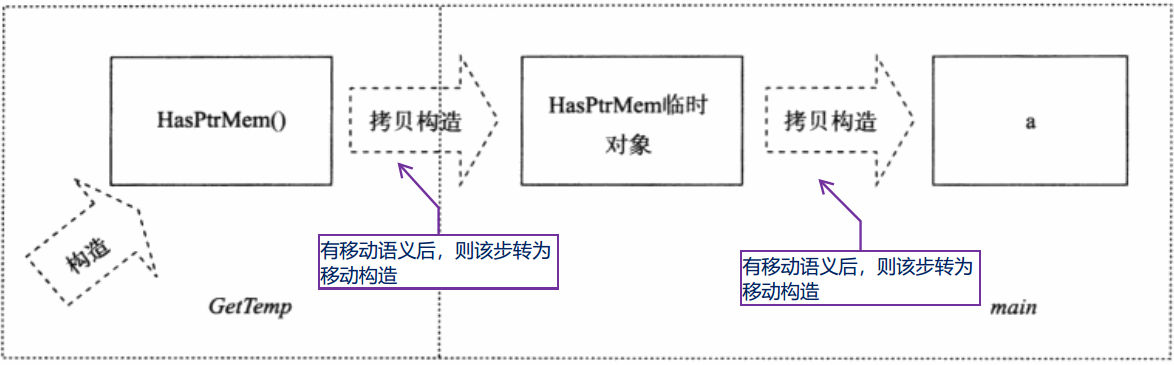
const string &shorterString(const string &s1, const string &s2)

{return s1.size < s2.size ? s1 : s2;}

A为自定义类型，A get(){A a; return a;} get()

A getTemp() { A a; return a; }

A x=getTemp(); //不考虑移动语义，则构造、拷贝构造、析构、拷贝构造、析构、析构



A x=getTemp(); //引入移动构造后，则构造、移动构造、析构、移动构造、析构、析构

实现一个有理数相乘的函数：（自定义类Rational，有数据成员x\_）

只能按值返回对象，但有了移动构造函数后，优先调用移动构造函数

**const** Rational operator\*(**const** Rational& lhs, **const** Rational& rhs)  //运算符重载函数，但不是类的成员函数 返回const的原因是，内置类型3\*5=10不被允许，因此设计Rational类理应如此

{

    Rational a(lhs.x\_\*rhs.x\_);

**return** a;

}//需要调用构造函数、拷贝构造、析构

const Rational&& operator\*(const Rational& lhs, const Rational& rhs)   ///错误，返回的右值引用指向栈中已销毁对象

{

    Rational a(lhs.x\_\*rhs.x\_);

    return std::move(a);

}

const Rational& operator\*(const Rational& lhs, const Rational& rhs)  //错误，返回的左值引用指向栈中已销毁对象

{

    Rational a(lhs.x\_\*rhs.x\_);

    return a;

}

# noexcept：

noexcept形如其名，表示其修饰的函数不会抛出异常。在函数声明后加上noexcept关键字。在C++11中如果noexcept修饰的函数抛出了异常，编译器可以选择直接调用std::terminate()函数来终止程序的运行

对于移动构造函数来说，抛出异常有时是件危险的事情。因为可能移动语义还没完成，一个异常却抛出来了，这就会导致一些指针就成为悬挂指针。因此程序员应该编写不抛出异常的移动构造函数，为所有移动构造函数添加noexcept关键字，可以保证移动构造函数中抛出来的异常会直接调用terminate程序终止运行，而不是造成指针悬挂的状态。

# 重载、隐藏、覆盖

重载(overload)：

(1)同一作用域，函数名相同，参数列表不同（参数个数不同或参数类型不同）

(2)函数重载与返回值相同与否无关

(3)两个类的成员函数，函数名相同，参数列表相同，但一个声明后有const，另一个没有const，这两个函数也是重载

(4)不能通过访问权限public、protect、private进行重载，是重定义。如同一类中public: void show() {}和private: void show() {}是重定义

(5) int selfadd(int x) {} 和 int selfadd(const int x) {}不是重载，是重定义；

int selfadd(int& x) {}和 int selfadd(const int& x) {}是重载；

int selfadd(int\* x) {}和 int selfadd(const int\* x) {}是重载

(6) 参数类型是左值引用与右值引用构成重载，如void push\_back (const T& value); 和 void push\_back (T&& value);

隐藏(hide)：

(1)父类不是虚函数，子类和父类的函数名称相同，但参数列表相同或不同，父类函数被隐藏

(2)父类是虚函数，子类和父类函数名称相同，但参数列表不同，父类函数被隐藏

(3)父类是虚函数，子类和父类函数名称相同，但参数列表相同，常量性不同，父类函数被隐藏

覆盖(override)

(1)父类为virtual函数，子类和父类函数名相同、参数列表相同、返回值类型相同、常量性相同（即函数后的const关键字）

(2)父类为virtual函数，子类和父类函数名相同、参数列表相同、常量性相同、返回值类型不同但父类函数返回类A的指针或引用，子类返回类A的某个子类的指针或引用（称为返回值类型协交）；

注意：

* 覆盖函数的访问修饰符可以不同于被重写的函数，如基类的virtual函数的修饰符为private，派生类改为public或protected也是可以的。
* 静态方法不能被重写，也就是static和virtual不能同时使用。

#include<iostream>

**using** std::cout;

**using** std::endl;

**class** Person

{

**public**:

**virtual** **void** show(**int**) { cout << "void Person::show(int)" << endl; }

**virtual** **void** who() { cout << "void Person::who()" << endl; }

**virtual** **void** add(**int**) { cout << "void Person::add(int)" << endl; }

**virtual** **void** subtract() { cout << "void Person::subtract()" << endl; }

**virtual** Person& multiply() { cout << "Person& Person::multiply" << endl; **return** \***this**; }

**virtual** Person\* divide() { cout << "Person\* Person::divide" << endl; **return** **this**;}

**virtual** ~Person(){}

};

**class** Stu :**public** Person

{

**public**:

**virtual** **void** show(**int**) { cout << "void Stu::show(int)" << endl; }//覆盖

**virtual** **void** show() { cout << "void Stu::show()" << endl; }//不是覆盖，也没有隐藏(因为上一行代码的存在)

**virtual** **void** add() { cout << "void Stu::add()" << endl; }  //隐藏了父类的add(int)

    //virtual int subtract() { cout << "void Stu::subtract()" << endl; return -1; } 报错 ，与重写虚函数Person::subtract既不相同也不协交

    //父类子类都是虚函数，函数名与参数列表相同，则返回类型与父类必须相同或协交，否则报错。如果相同或协交，则是覆盖

**virtual** Stu& multiply() //覆盖，函数名、参数列表与父类一致，返回类型与父类协交

    {

        cout << "Stu& Stu::multiply" << endl; **return** \***this**;

    }

**virtual** Stu\* divide() //覆盖，函数名、参数列表与父类一致，返回类型与父类协交

    {

        cout << "Stu\* Stu::divide" << endl;  **return** **this**;

    }

};

**int** main()

{

    Person per;

    Stu stu;

    Person\* pp = &stu;

    pp->show(1);  //void Stu::show(int)

    //pp->show();  报错

    stu.who(); //正常状态，void Person::who()

    stu.add(); //void Stu::add()

    //stu.add(1); 报错，被隐藏了

    pp->multiply(); //Stu& Stu::multiply

    pp->divide(); //Stu\* Stu::divide

}

**class** Shape

{

**public**:

**virtual** ~Shape() {}

**virtual** **void** draw() **const** {cout << "abstract drawing mechanism" << endl;}

};

**class** Circle :**public** Shape

{

**public**:

**virtual** ~Circle() {}

**virtual** **void** draw()  //因常量性不同，因此未覆盖，而是隐藏

    {

        cout << "draw a circle" << endl;

    }

};

Circle obj;

Shape\* p = &obj;

p->draw();//"abstract drawing mechanism",调用父类函数

obj.draw();//"draw a circle",调用子类函数

**const** Circle c;

//c.draw(); 报错

子类覆盖父类任一个方法后，父类中被覆盖方法的所有重载(overload)方法都将被隐藏，子类对象不能再使用这些方法。如果不希望该方法的重载方法被隐藏则，子类应一一覆盖之。

fido.move();//fido为子类对象；子类覆盖了父类的virtual move()方法

//fido.move(4); //报错，被覆盖方法的重载方法被隐藏了

fido.Mammal::move();

fido.Mammal::move(3);

覆盖基类方法后，子类对象可以 “子类对象名.基类名::被覆盖方法” 的格式调用基类被覆盖方法，及被覆盖方法的重载方法

函数形参为**const**类型

一：

(1)**void** shout(**const** **int** a);**void** shout(**int** a);是重定义，编译报错，如果定义在相同的作用域中

**void** shout(**const** **int** a);实参可以是 **const** **int** a=10; shout(a);可以是**int** a=10; shout(a);

**void** shout(**int** a);实参可以是 **const** **int** a=10; shout(a);可以是**int** a=10; shout(a);

二：

**void** shout(**const** **int**& a);**void** shout(**int**& a);同时出现，是重载函数

**int** a=10;

    shout(a);//调用void shout(int& a)

**const** **int** b=8;

    shout(b);//调用void shout(const int& a);

**void** shout(**const** **int**& a);实参可以是 **const** **int** a=10; shout(a);可以是**int** a=10; shout(a);

**void** shout(**int**& a);实参不可以是 **const** **int** a=10; shout(a);

三：

（7）**int** a=10;**const** **int**& c=a;可以

（8）**const** **int** c=12;  **int**& r=c报错，无法从“**const** **int**”转换为“**int** &”;

区分隐藏与覆盖：

#include <iostream.h>

**class** Base

{

**public**:

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

**void** g(**float** x){ cout << "Base::g(float) " << x << endl; }

**void** h(**float** x){ cout << "Base::h(float) " << x << endl; }

};

**class** Derived : **public** Base

{

**public**:

**virtual** **void** f(**float** x){ cout << "Derived::f(float) " << x << endl; }

**void** g(**int** x){ cout << "Derived::g(int) " << x << endl; }

**void** h(**float** x){ cout << "Derived::h(float) " << x << endl; }

};

（1）函数Derived::f(**float**)覆盖了Base::f(**float**)。

（2）函数Derived::g(**int**)隐藏了Base::g(**float**)，而不是重载。

（3）函数Derived::h(**float**)隐藏了Base::h(**float**)，而不是覆盖。

实际上public继承，子类不应该隐藏父类的public非虚成员函数

有、无const是重载

void eat()

{

cout<<"eatting"<<endl;

}

void eat() const

{

cout<<"eat"<<endl;

}

Mammal my;

my.eat(); // 输出eatting

const Mammal mz;

mz.eat(); // 输出eat

const对象仅能调用const成员函数，不能调用非const成员函数 （在成员函数都是public类型的前提下）

无const修饰的对象，既能调用const成员函数，又能调用非const成员函数 （在成员函数都是public类型的前提下）

**using避免类继承的隐藏机制：**

class Person

{

public:

virtual void eat(const char\* str) { cout << "person eating" << endl; }

virtual void eat(int) { cout << "person eating" << endl; }

virtual void sleep(const char\* str) { cout << "person sleep" << endl; }

virtual void sleep(int) { cout << "person sleep" << endl; }

void run(const char\* str) { cout << "person run" << endl; }

};

class Son : public Person

{

public:

virtual void eat(const char\* str) override { cout << "son eating" << endl; }

virtual void sleep(const char\* str) override { cout << "son sleeping" << endl; }

using Person::sleep;

void run(int) { cout << "son run" << endl; }

using Person::run;

};

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

{

Son son;

son.eat(""); //son eating

son.Person::eat(""); //person eating

son.Person::eat(3); //person eating, 不能son.eat(0)，被隐藏

son.sleep("hi"); //son sleep，子类覆盖

son.sleep(3); //person sleep

son.Person::sleep("");

son.Person::sleep(3);

son.run(3); //son run

son.run("aa"); //person run

}

转换为自定义类：

A a(30);

const A& ref = a;

((A)ref).add(); //实际上是生成的一个A类型副本调用了add函数

子类覆盖父类虚函数，想调用父类函数

class A { public:virtual void add() {/\*\*/ } };

class B :public A {

public:virtual void add() { ((A)(\*this)).add(); 其它操作 } //调用的并不是当前

对象上的函数，而是转型动作所建立的一个\*this对象的基类成分的暂时副本身上的add函数

};

正确做法应该是：

class B :public A { public:virtual void add() { A::add(); 其它操作 } };

class B :public A { public:virtual void add() { \_\_super::add(); 其它操作 } };

# 封装、继承、多态

1. 类的封装性

数据成员最好是private：

* public数据成员如果改变，则所有客户码被破坏、所有继承类被破坏
* protected数据成员如果改变，则所有继承类被破坏

因此，设计类时，数据成员只应该是private。

此外，public成员函数的声明一定要慎重，则以后对类修改时，只要不需改变public函数声明，则其它改动的影响都局限于当前类和继承类中，也就是“封装性”很好。

class Date {……};//内部省略

class Person

{

public:

Person( const Date birthday);

std::string setBirthday();

/\*实现细目中Date类相关的操作为创建Date对象、调用public成

员，所以只要Data类的public成员声明不变则该类Person不受影响。如

修改了Date类的私有对象或修改了public Date add(int)函数的

内部实现，都不会影响Person类\*/

private:

Date\* theBirthDate;

};

(2)

创建子类对象时，先调用父类构造函数，再调用子类构造函数；销毁子类对象时，先调用子类析构函数，再调用父类析构函数

创建子类对象时，向基类的构造函数传递参数：子类构造函数名 冒号 基类(基类构造函数需要的参数)

Dog(int age,BREED breed):Mammal(age),breed\_(breed)

{

cout<<"Dog(int,BREED).."<<endl;

}

其中，Mammal为基类，Dog为子类，BREED为子类新增的成员变量

(3)

virtual方法，借助指针和引用实现多态

1. **class** Mammal
2. {
3. **public**:
4. Mammal() :age(1) { cout << "Mammmal construc" << endl; }
5. **virtual** ~Mammal() { cout << "Mammmal destruc" << endl; }
6. **virtual** **void** speak() **const** { cout << "Mammal speak" << endl; }
7. **protected**:
8. **int** age;
9. };
10. **class** Dog :**public** Mammal
11. {
12. **public**:
13. Dog() { cout << "Dog construct\n"; }
14. ~Dog() { cout << "Dog destruct\n"; }
15. **void** wagTail() { cout << "Dog wag tail..\n"; }
16. **virtual** **void** speak()**const** { cout << "wolf\n"; }//覆盖，前边关键字virtual可有可无
17. };
18. Mammal\* dog = **new** Dog; //Mammmal construc  Dog construct
19. dog->speak();  //wolf
20. ((Dog\*)dog)->wagTail();//Dog wag tail..
21. **delete** dog; //Dog destruct Mammmal destruc

public继承，描述的是is-a关系，因此子类不应该隐藏父类的非虚public函数（即不能有与父类非虚public函数同名的成员函数）

public继承，父类指针可以指向子类对象，父类引用可以绑定子类对象；private继承则不行

public继承，子类对象可以赋值给父类对象或作为父类拷贝构造函数的实参，发生截断；private继承则不行

class Person {};

class Student : private Person {};

void eat(const Person& p);

void study(const Student& s);

Person p;

Student s;

eat(p); //没问题

//eat(s); //错误！私有继承表示的不是is-a关系

public继承，指向子类对象的基类指针p，p->func()：

* 该方法是父类的虚函数，且子类覆盖之，则调用子类的方法
* 该方法是父类的一般函数，且子类隐藏之，则调用父类的方法（实际上pubic继承不应该隐藏父类非虚函数）
* 该方法父类没有，仅子类有，则编译出错，必须将指针p强制转换为子类指针，再((Dog\*)p)->func(),才能访问子类的方法

(4)

* 通过指针和引用作为函数行参时，且是public继承，则能发挥虚方法的威力
* 子类对象强制转换为父类对象，且是public继承，则切除子类对象的子类部分；
* public继承，指向子类对象的指针可以直接赋给基类指针
* public继承，基类指针可以强制转换为子类指针：

p是基类指针，func()为子类新增的方法(Dog\*)p->func();

* public继承，基类指针动态转换为子类指针：

Dog\* p=dynamic\_cast<Dog\*>(p);如果p指向子类Dog对象，则正确返回，转型为子类指针，如果p指向基类对象，则返回NULL

**class** Mammal

{

**public**:

    Mammal() {}

**virtual** ~Mammal() {}

**virtual** **void** speak()**const** { cout << "Mammal speak.." << endl; }

};

**class** Dog :**public** Mammal

{

**public**:

**void** speak()**const** { cout << "wolf" << endl; }  //必须有cosnt，不然不能形成覆盖，而是新增函数

};

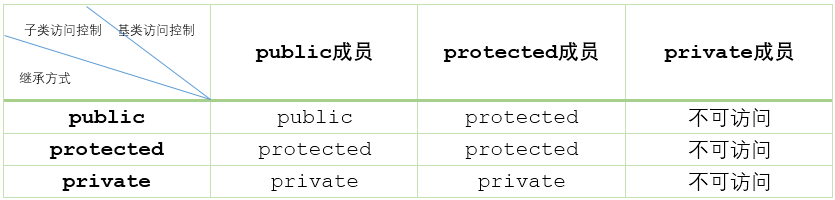
Dog dog;

Mammal mal = dog;       //切除了dog的子类部分

mal.speak();  //Mammal speak..

（5）如果类中任意一个函数是虚函数（纯虚和非纯虚），则析构函数一定要设置为虚函数

（6）继承方式是public、protected、private三者之一。继承方式影响基类成员在派生类中的访问控制属性



由表可知：

* public继承，父类的protected成员在子类中的访问权限是protected。protected成员不能被本类对象访问；protected成员可以被子类成员函数访问；protected成员不能被子类对象访问。即
* 无论哪一种继承方式，子类成员函数都可以访问基类public、protected成员，但不能访问private成员
* 只有public继承的子类对象可以访问基类的public成员，protected、private继承的子类对象不能访问基类的public成员
* 子类对象想访问父类的成员，除非是public继承且访问的是父类public成员
* 类的成员函数可以访问其所属类的所有private、protected、public成员，也可以访问父类的public、protected成员。在类外部的一个函数中声明一个对象，则可以访问所有public成员

public继承建立的是is-a关系,如学生Stu是人Person。

复合，即一个类中的数据成员是另一类的对象，复合建立的是has-a关系（如人有住址Address）“由某物实现出”关系或（如双端队列可由vector实现）。

private继承建立的是“由某物实现出”的关系，与复合的区别在于子类不但可以使用父类的public成员函数而且可以使用父类的protected成员函数而且可以重新定义继承而来的 virtual 函数，而复合只能使用public成员函数

class Fan:private ElectricMotor这种继承能让Fan使用ElectricMotor的【public或protected限定的函数，private限定的函数仍不能使用】，甚至覆盖其虚函数，然而Fan的用户不能访问基类ElectricMotor及其函数，对Fan的用户来说，该基类的所有成员都是私有的，而不管其使用的访问权限限定符是什么。

**class** Motor

{

**public**:

    Motor(){};

**virtual** ~Motor(){};

**void** startMotor()    {  accelerate(); cruise();  }

**void** stopMotor()  { cout<<"Motor stop"<<endl;   }

**protected**:

**void** accelerate()  { cout<<"motor start---accelerate"<<endl;  }

**private**:

**void** cruise()  { cout<<"motor running at a constant speed"<<endl; }

};

**class** Fan:**private** Motor

{

**public**:

    Fan(){}

    ~Fan(){}

**void** startFan() {accelerate();}

**void** stopFan(){stopMotor();}

};

**void** main()

{

    Fan fan;

    fan.startFan();

    fan.stopFan();//不能fan.stopMotor();

}

(7)

class Mammal

{

public:

Mammal():age(11){cout<<"constr,age is:"<<age<<endl;}

~Mammal(){}

void tellAge()

{

Mammal mal;

cout<<mal.age<<endl; //在类的成员函数里，创建所属类的对象，该对象可以直接访问类的private、protected成员。

}

private:

int age;

};

(8)

* 多重继承，构造函数和析构函数调用顺序

class FlyHorse:public Horse,public Bird

FlyHorse\* pfh=new FlyHorse;

delete pfh;

/\*

Horse() construct

Bird() construct

FlyHorse() construct

FlyHorse() destruct

Bird destruct

Horse destruct

\*/

* 多重继承，创建子类对象时，给两个基类构造函数传递参数

FlyHorse::FlyHorse(COLOR color,HANDS height,bool migrate,int age,long number):Horse(color,height,age),Bird(color,migrate,age),itsNumberBelievers(number) ｛ ｝

* 两个基类都有color成员变量，都有virtual getColor()，ppeg为子类对象指针，公有继承两个基类：ppeg->getColor()，会报错，必须指明哪个基类的getColor

ppeg->Bird::getColor()

(9)

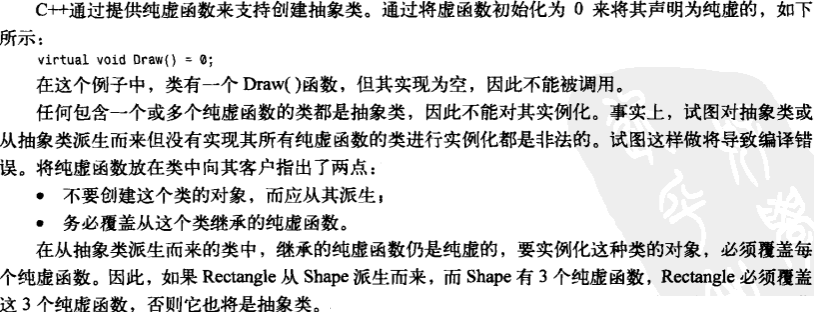
抽象类，纯虚函数

对public继承：

纯虚函数也可以有定义，有定义的纯虚函数用于：告知子类必须继承接口，且必须显式指明自己实现还是使用父类的实现

非纯虚函数：子类继承接口，且默认使用父类实现

非虚函数：子类强制采用这种接口和实现（不应该隐藏之）



**class** Shape

{

**public**:

    Shape(){};

**virtual** ~Shape(){}

**virtual** **void** draw() **const** =0

    {

        cout<<"abstract drawing mechanism"<<endl;

    }

};

**class** Circle:**public** Shape

{

**public**:

    Circle(){}

    ~Circle(){}

**void** draw() **const**

    {

        cout<<"draw a circle"<<endl;

        Shape::draw();

    }

};

**void** main()

{

    Circle obj;

    obj.draw();

}

通常，不给抽象类的纯虚函数以实现，因为抽象类用作从其派生来的类的接口定义。但也可以给纯虚函数提供实现，在派生类的覆盖方法中调用该实现，旨在给所有函数提供通用功能

# 虚函数实现机制：

虚函数用于实现多态。编译器借助虚函数表实现虚函数机制。

一个类有虚函数，则会维护一个虚函数表，虚函数表中存放虚函数指针，依次是指向继承的父类虚函数的函数指针、指向子类虚函数的函数指针。如果子类虚函数覆盖了其父类版本，则子类的虚函数表中父类虚函数指针被替换为覆盖的子类虚函数的指针。

public继承，无虚函数覆盖时：

1）虚函数指针按照所指虚函数的声明顺序放于表中。

2）父类的虚函数指针在子类的虚函数指针前面

public继承，有虚函数覆盖时：

1）覆盖的f()函数的指针被放到了虚表中原来父类虚函数指针的位置。

2）没有被覆盖的函数依旧。

有虚函数的类B，实例化产生一个对象时，分配给该对象的内存空间包括指向类B的虚函数表的指针，继承自父类的成员变量、子类特有的成员变量。

class A

{

public:

virtual void f(){};

virtual void g(){};

private:

int a

};

class B : public A

{

public:

virtual void g(){};

private:

int b;

};

某处有语句 B b;的时候，编译器分配空间时，除了A的int a，B的成员int b；以外，还分配了一个虚指针，指向B的虚表vtableB。

因为A有virtual void f()，和g()，所以编译器为A类准备了一个虚表vtableA，内容如下：

|  |
| --- |
| A::f 的地址 |
| A::g 的地址 |

B因为继承了A，则B一定也有虚函数，所以编译器也为B准备了一个虚表vtableB，内容如下：

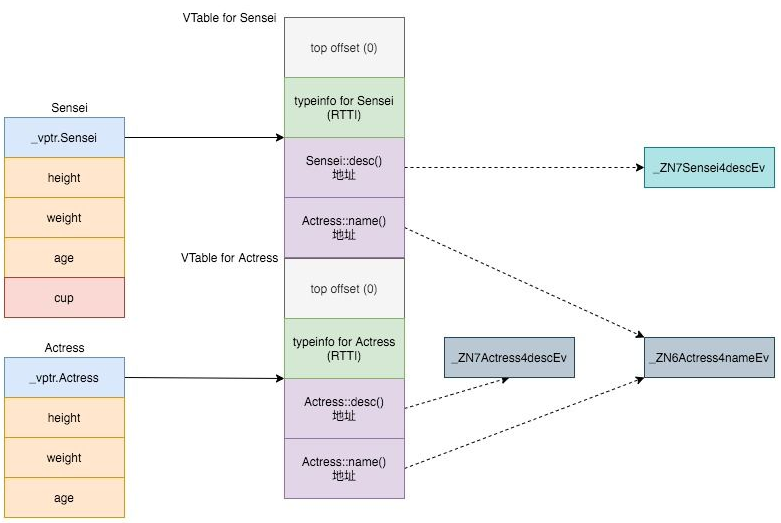
|  |
| --- |
| A::f 的地址 |
| B::g 的地址 |

当如下语句的时候：

A \*pa = &b;

pa->g();

编译器知道的是，g是一个声明为virtual的成员函数，那么编译器编译这条语句的时候就通过对象b内存中指向虚函数表的指针访问虚函数表，进而访问虚函数。由于b的内存中的虚函数表的g()的位置已经被B::f()函数地址所取代，于是在实际调用发生时，是B::f()被调用了。这就实现了多态。



# A a和A a()：

函数的声明不一定必须是全局的，可以放在函数内

C++函数不能嵌套定义，即一个函数体中不能有另一个函数的定义

但一个函数体中可以有另一个函数的声明，如

#include<iostream>

void main()

{

double func(int, int, int);

std::cout << "4+3/2=" << func(4, 3, 2) << std::endl;

}

double func(int a, int b, int c)

{

return (double)a + (double)b / c;

}

一个类有无参构造函数函数，则在main函数中

类名 a; //实例化类的对象

类名 a()； //声明一个无参函数

class A

{

public:

A() { std::cout << "A()" << std::endl; }

A(int x) {std::cout << "A(int x)" << std::endl;}

~A() { std::cout << "~A()" << std::endl; }

};

void main()

{

//A a1; //是定义，会调用无参构造函数

//A a2(3); //是定义，调用构造函数A(int )

A a3();//是一个无参函数的声明，函数的返回值是A类对象，无参，typeid(a3).name()的返回值是class A \_\_cdecl(void)

}

# 构造、拷贝语义、移动语义

对于一个类X, 如果一个构造函数的第一个参数是下列之一:  
a) X&

b) const X&

c) volatile X&

d) const volatile X&

且没有其他参数或其他参数都有默认值,那么这个函数是拷贝构造函数.

Stu(const Stu& a, const char\* x="cjzhang") :p(new int(\*a.p)) { cout << "Stu(const Stu& a)" <<x<< endl; }是拷贝构造函数

构造、析构、拷贝构造、拷贝赋值、移动构造、移动赋值函数仅满足一定条件时，编译器才可以自动生成。即使可以自动生成，也仅当需要调用时，才会生成。

仅当用户定义的类中没有显式定义的任何构造函数（包括构造、拷贝构造、移动构造），编译器才可以自动为该类型生成默认构造函数，则编译器可以自动生成，但仅当默认构造函数需要被调用时，才会自动生成。如果程序中，一直没有A a;类似的语句，则编译器不会自动生成默认构造函数

用delete关键字或default关键字，也相当于“显示定义”

如显式自定义了A::A(){}，则编译器不再生成构造函数

如移动构造函数仅当没有显式定义析构、拷贝构造、拷贝赋值函数时，编译器才可以自动生成。但程序中没有（形如A a2(std::move(a1))）语句需要调用移动构造函数，则编译器不会生成默认移动构造函数

用户显式定义了拷贝构造函数，编译器仍可以生成默认拷贝赋值函数

用户delete拷贝构造函数，则编译器不会生成移动构造函数、仍可生成拷贝赋值函数、仍可以生成移动赋值函数

用户delete拷贝赋值函数，则编译器不会生成移动赋值函数、仍可生成拷贝构造函数、仍可以生成移动构造函数

如果 A(const A&) = delete;

A& operator=(const A&) = delete;，则类A既无移动语义（构造、赋值），又无拷贝语义（构造、赋值）

编译器生成的默认析构函数非虚，除非其父类的析构函数为virtual

编译器产生的拷贝构造函数，只拷贝非静态成员函数（包括引用类型数据成员）

编译器合成的默认移动构造函数，单纯地调用该类下所有成员的移动构造函数（如果是内置类型数据成员则复制每个bit）

编译器合成的默认拷贝构造函数，单纯地调用该类下所有成员的拷贝构造函数 （如果是内置类型数据成员则复制每个bit）

拷贝赋值运算符：MemoryBlock& operator=(const MemoryBlock& other)

移动赋值运算符MemoryBlock& operator=(MemoryBlock&& other)

编写拷贝赋值运算符和移动赋值运算符函数时：判断if (this != &other) 、考虑new失败抛出异常的情况

如果类有引用数据成员或const数据成员，则编译器不会生成默认拷贝赋值函数，但会生成默认拷贝构造函数（如果其它条件满足）

如果父类的拷贝赋值运算符函数是私有的，则编译器不会为子类生成默认拷贝赋值函数（编译器想调用调用父类的，但无权调用，因此无能为力）

如果父类的拷贝构造函数是私有的，则编译器不会为子类生成默认拷贝构造函数（编译器想调用调用父类的，但无权调用，因此无能为力）

为子类撰写拷贝构造函数、拷贝赋值函数，如果基类存在私有的数据成员且没有提供访问函数，则不能以赋值的方式，只能借助子类的拷贝构造函数、拷贝赋值函数

class A

{

public:

A(int age) :age\_(age) { cout << "A cons" << endl; }

A(const A& other):age\_(other.age\_) { cout << "A cpy" << endl; } //类的成员函数中可以通过一个类对象直接访问其私有成员

A& operator=(const A& other)

{

age\_ = other.age\_;

return \*this;

}

private:

int age\_;

};

class B:public A

{

private:

int height\_;

public:

B(int a, int h) :A(a), height\_(h) { cout << "B cons" << endl; }

B(const B& other):A(other),height\_(other.height\_) { cout << "B cpy" << endl; } //A的拷贝构造需要A类实参，传递A的子类也可以

B& operator=(const B& other)

{

A::operator=(other);

height\_ = other.height\_;

return \*this;

}

};

# 拷贝构造与拷贝赋值函数

拷贝构造函数使用传入对象的值生成一个新的对象的实例，而赋值运算符是将对象的值复制给一个**已经存在的实例。**

A obj = a完全等价于A obj(a)，都是调用拷贝构造函数

赋值运算符函数，返回值类型为A&，函数体中return \*this

**class** Person

{

**public**:

    Person(){}

    Person(**const** Person& p)  {}

    Person& operator=(**const** Person& p)

    {

**return** \***this**;

    }

}

**Person** p;

**Person** p1 = p; /\*调用拷贝构造函数。虽然使用了"="，但是实际上使用对象p来创建一个新的对象p1。也就是产生了新的对象，所以调用的是拷贝构造函数。\*/

**Person** p2;

p2 = p; //调用赋值运算符函数，为一个已经存在的对象p2赋值

# 友元类&友元函数：

类的大小sizeof(类名)或sizeof(对象名)不包括静态成员变量、成员函数所占空间。成员函数在类外单独存放，并且同类的对象共用一份成员函数（节省了空间）

友元函数没有this指针

类的静态成员函数没有this指针

this指针是类的非静态成员函数的第一个隐含形参

total为对象，isbn(）为非静态成员函数 ，如果调用total.isbn()则编译器负责把 total 的地址传递给 isbn 的隐式形参 this，可以等价地认为编译器重写成以下形式：

//伪代码，用于说明调用成员函数的实际执行过程 Sales\_data::isbn(&total)

其中，调用 Sales\_data 的 isbn 成员时传入了 total 的地址。

在成员函数内部，我们可以直接使用成员，而无须通过成员访问运算符来做到这一点，因为 this 所指的正是这个对象。任何对类成员的直接访问都被看作是对 this 的隐式引用，也就是说，当 isbn 使用 bookNo 时，它隐式地使用 this 指向的成员，就像我们书写了 **this->bookNo** 一样。

友元函数不是类的成员函数，在类中声明加friend关键字，类外定义（也可以类中定义）

友元函数中可通过对象名直接访问该对象的私有成员和保护成员

友元类在类中声明，在类外定义

class B

{

public:

B(int b):b\_(b){}

friend class C;

private:

int b\_;

};

class C

{

public:

int getBb\_(B obj) { return obj.b\_;}

};

友元关系没有继承性。假如类B是类A的友元，类C继承于类A，那么B不是C的友元。假如类A是类B的友元，类C继承类A，C不是B的友元

友元关系没有传递性加入类B是类A的友元，类C是类B的友元，那么友元类C是没办法直接访问类A的私有或保护成员，也就是不存在“友元的友元”这种关系

如果类A是类B的友元函数，则类A的所有成员函数都是类B的友元函数，即类A的所有成员函数中都可以通过类B的某个对象名直接访问这一对象的私有成员和保护成员

# int、double数据类型的最大值、最小值:

int和long int（long int即long）都是4字节

sizeof(int):4 INT\_MAX:2147483647

sizeof(long long int):8 LLONG\_MAX:9223372036854775807

sizeof(float) :4

sizeof(double) :8

sizeof(long double) :8

double类型的取值范围：

“正值取值范围为 4.94065645841246544E-324 到 1.79769313486231570E+308

负值取值范围为 -1.79769313486231570E+308 到 -4.94065645841246544E-324”

DBL\_MIN: 2.22507e-308 DBL\_MAX 1.79769e+308

DBL\_TURE\_MIN: 4.94066e-324

DBL\_TRUE\_MIN是对最小非规格化浮点数的通用非标准扩充，而DBL\_MIN才是最小规格化浮点数的值，并且只在DBL\_TRUE\_MIN未定义时使用

头文件<float.h>中，有定义的常量DBL\_MAX，这个常量表示“能表示出来的最大的双精度浮点型数值”。<float.h>中还有常量DBL\_MIN,表示可以用规格化表示的最小的正浮点数，但DBL\_MIN并不是最小的正浮点数，因为可以用非规格化浮点数表示的更小。

void main()

{

cout << "DBL\_MIN:" << DBL\_MIN <<" DBL\_MAX "<< DBL\_MAX << endl;

cout << "DBL\_TURE\_MIN" << DBL\_TRUE\_MIN << endl;

double value;

for (value = 36; value > 0; value /= 2)

printf("%.3le ", value);

}

/\*

DBL\_MIN:2.22507e-308 DBL\_MAX 1.79769e+308

DBL\_TURE\_MIN4.94066e-324

导致循环终止的原因是，循环中最后一个数字4.941e-324除以2之后的结果小于DBL\_TRUE\_MIN的值4.94066e-324

\*/

# 类的const数据成员：

const 类型名 变量名 类中的常数据成员只能通过构造函数的参数初始化表进行初始化

如类Time中定义了常数据成员const int hour;//定义hour为常数据成员；

(1)不能采用在构造函数中对常数据成员赋值的方法，下面的用法是非法的：

Time::Time(x) {hour= h;} // 非法，不能对之赋值。因为常数据成员是不能被赋值的，即VS提示的“表达式必须是可修改的左值”。同理含有常数据成员的类，不会生成默认拷贝赋值和移动赋值函数

(2)如果在类体外定义构造函数，应写成如下形式：

Time::Time(x):hour(h){} //通过参数初始化表对常数据成员hour初始化

分析：构造函数可以分为两个阶段：

1. 初始化阶段（利用初始化列表完成）；
2. {}函数体赋值阶段。

所以：

1. 常数据成员只能通过构造函数的初始化表对其初始化，包括指针常量double\* const range
2. 当数据成员是引用类型时，也只能使用构造函数的初始化表进行初始化。如，int& min

#include<iostream>

**using** **namespace** std;

**class** CParticle

{

**public**:

    CParticle(**int** dim, **double** \*ran,**int**& m) : dimension(dim),range(ran),min(m)  /\*冒号后边的是构造函数的初始化表\*/  {  }

**private**:

**const** **int** dimension;

**double**\* **const** range;

**int**& min;

};

**void** main()

{

**double** ran[3] = {1.1, 2.2, 3.3};

    CParticle obj(3,ran,min);

}

const型的成员函数和非const型的成员函数构成重载

非const对象优先调用非const成员函数，没有非const成员函数则调用const函数。

const对象仅能调用const成员函数，不能调用非const成员函数 （在成员函数都是public类型的前提下）

指向对象的常量指针const APSO\* ptr，也只能调用const成员函数，不能调用非const成员函数

无const修饰的对象，既能调用const成员函数，又能调用非const成员函数 （在成员函数都是public类型的前提下）

class A

{

public:

void func()const {}

};

class B

{

public:

void show() const

{

a.func(); //这里func()必须是类A的const成员函数，否则报错

}

private:

A a;

};

CLR(Common Language Runtime公共语言运行时)

# 类的const函数成员：

C++标准（不像C标准）未能指定哪一个标准头文件必须或者可能被其他标准头文件#include

在一个const成员函数内部，类的所有非静态数据成员都变成常量

class NiftyEmailProgram{

private:

map<string, string> nicknames{ {"zh","120"},{"li","45"} };

int age = 10;

public:

void showEmailAddress(const string& nickname)const{

int& r = age;//报错：int&不能绑定const int类型

map<string, string>::iterator i = nicknames.find(nickname);

//报错。调用的是const\_iterator find()const，而不是iteratorfind()

}

};

const Widget\* p = new Widget(1, 1);

delete p;

//使用strcpy\_s时，没必要将buffer初始化全为0

char str[7];

strcpy\_s(str,7, "hello"); //一定会置str[5]为0

errno\_t strcat\_s(

char \*strDestination,

size\_t numberOfElements,

const char \*strSource

);

strcat\_s 功能追加 strSource 到 strDestination

strSource 的初始字符覆盖 strDestination终止 null 字符。

如果源和目标字符串重叠，则 strcat\_s 的行为未定义。

注意第二个参数是缓冲区的总大小，而不是剩余的大小：

构造函数写成完美转发

class Teacher

{

private:

std::string name;

std::string position;

public:

template <typename S1, typename S2>

Teacher(S1&& n, S2&& p): name(std::forward<S1>(n)), position(std::forward<S2>(p)) {};

};

# 类的static数据成员、函数成员：

* 全局函数中的static局部变量默认初始化为0。全局变量也具备这一属性。static局部变量位于进程虚拟内存空间的全局变量/局部静态变量区域。全局变量，static变量的内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。

1. **void** main()
2. {
3. **static** double x;
4. cout << x << endl;  //结果是0，而不是乱七八糟的数
5. }

* 全局函数中的static局部变量只被初始化一次，所在的全局函数return时存储空间不释放，下一次依据上一次结果值。C++ 保证，函数内的局部静态对象会在“该函数被调用期间”“首次遇上该对象之定义式”时被初始化。
* 全局静态函数，在普通全局函数定义前加static则为静态函数。在一个源文件中定义的静态函数，只能被本文件中的函数调用，而不能被同一程序其它文件中的函数调用。好处是： 不同的人编写不同的函数时，不用担心自己定义的函数，是否会与其它文件中的函数同名，因为同名也没有关系。全局静态变量同理。
* 类的静态成员函数可以在类的内部或外部定义，当在外部定义时，不能重复关键字static

**class** MyCal

{

**public**:

**static** **double** add(**double** x, **double** y) { **return** x + y; }

**static** **double** multi(**double** x, **double** y);

};

**double** MyCal::multi(**double** x, **double** y)

{

**return** x\*y;

}

* 类的静态成员函数不能直接访问非静态成员变量
* 虚函数（virtual）不能是static
* 由于静态成员声明于类中，操作于其外，所以对其取地址操作，就多少有些特殊

变量地址是指向其数据类型的指针 ，函数地址类型是一个“nonmember函数指针”。

**class** T{

**public**:

**static** **int** a;

**static** **void** show() { cout << "myself" << endl; }

**void** run() { cout << "run..." << endl; }

};

**int** T::a = 10;

**int** main(){

**int**\* a = &T::a;

**void**(\*p)() = &T::show;   p();

**void**(T::\*ptr)()= &T::run;  (T().\*ptr)();

}

* 类的静态成员变量仍然遵循public，private，protected访问准则，如下：

#include<iostream>

using namespace std;

class Dog

{

public:

Dog() {}

static char id;

private:

static const int num=10;

static const int age;

static int weight;

};

char Dog::id='w';

const int Dog::age=10; //初始化，不能出现static，但必须有const关键字

int Dog::weight=5; //不能在出现static关键字

void main()

{

cout<<Dog::id<<endl; //id为public成员

//cout<<Dog::num<<endl; 编译出错，无法访问private 成员

//cout<<Dog::age<<endl;

//cout<<Dog::weight<<endl;

}

* 类的静态成员变量（除了静态常量整形static const int可以直接在类中定义并初始化必须在类中声明、类外定义并初始化。

一个类的对象作为另一个类的成员变量时，构造函数的调用：

**class** Stu

{

**public**:

    Stu() { age = 10; cout << "Stu constr()" << endl; }

    Stu(**int** i) :age(i) { cout << "Stu constr(int i)" << endl; }

**private**:

**int** age;

};

**class** Pep

{

**public**:

    Pep() :weight(20) { cout << "Pep copnstr()" << endl; }

    Pep(**int** w) :weight(w), tom(3) { cout << "Pep copnstr(int)" << endl; }

**private**:

    Stu tom;

**int** weight;

};

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

{

    Pep p1;

    Pep p2(10);

}

/\*

运行结果：

stu constr()

pep copnstr()

stu constr(int i)

pep copnstr(int)

\*/

运行结果：

stu constr()

pep copnstr()

stu constr(int i)

pep copnstr(int)

请按任意键继续. . .

对于

class Garbo{

public:

Garbo() { cout << "Garbo构造.." << endl; }

~Garbo() { cout << "Garbo析构.." << endl; }

};

class Singleton{

public:

Singleton() { cout << "Singleton构造.." << endl; }

~Singleton() { cout << "Singleton析构.." << endl; }

Garbo gar;

};

void main(){

Singleton p1;

}

类A的一个数据成员是类B的对象b，则class A{public:B b;};

在A的初始化列表对b构造，A::A():b(){}、A::A():b(3){}，将分别调用B类的无参构造、B(int)构造函数

如果A的初始化列表没有显式指明B()或B(3)，则默认用B的无参构造函数构造b

/\*

运行结果：

Garbo构造..

Singleton构造..

Singleton析构..

Garbo析构.

\*/

如果是

class Singleton{

public:

Singleton() { cout << "Singleton构造.." << endl; }

~Singleton() { cout << "Singleton析构.." << endl; }

static Garbo gar; //此处仅声明

};

/\*

则运行结果：

Singleton构造..

Singleton析构..

\*/

因为类的静态成员在类内声明，类外定义并初始化，代码中仅声明gar，并没有定义及初始化gar

如果是：

**class** Singleton  {

**public**:

    Singleton() { cout << "Singleton构造.." << endl; }

    ~Singleton() { cout << "Singleton析构.." << endl; }

**static** Garbo gar;  //此处仅声明，定义在类外

};

Garbo Singleton::gar; //此处定义并初始化

/\*

则运行结果：

Garbo构造..

Singleton构造..

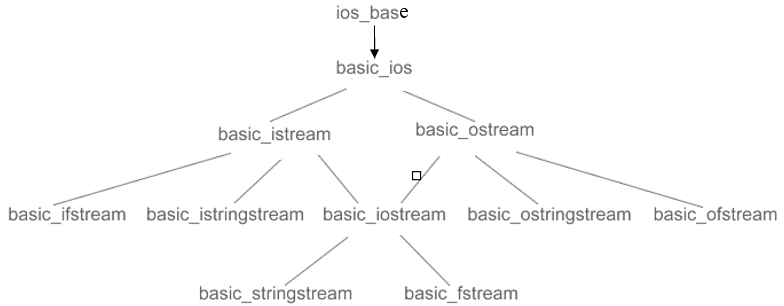
Singleton析构..

Garbo析构..

\*/

# C++中I/O类库继承结构：

C++流是指信息从外部输入设备（如键盘和磁盘）向计算机内部（即内存）输入和从内存向外部输出设备（如显示器和磁盘）输出的过程，这种输入输出过程被形象地比喻为“流”。对stringstream流则字符串相当于外设。



根据一个流类定义的对象也时常被称为流。

C++中的I/O类库，其所有类被包含在<iostream>、<fstream>和<sstream>这三个头文件

<iostream>：对标准输入、输出设备的输入输出简称为标准I/O

<fstream>：对在外存磁盘上文件的输入输出简称为文件I/O

<sstream>：对内存中指定的字符串存储空间的输入输出简称为串I/O。

C++为用户进行标准I/O操作定义了四个类对象 cin、cout、cerr、clog

cin隶属于istream,标准输入通道，对应于C中的 stdin。和键盘连接。

cout隶属于ostream，标准输出通道，对应于C中的stdout。和屏幕连接。

cerr隶属于ostream，是所有报错信息所使用的标准报错输出通道，对应于C中的stderr。和屏幕连接。无缓冲 (not buffered)

clog隶属于ostream, 是标准日志通道，C中没有对应物。有缓冲 (is buffered)

# std::cin

istream输入流类对”>>”运算符重载：istream& operator>>(简单类型标识符&)；从流中读数据赋给变量。返回istream&类型以便继续使用提取操作符为下一个变量输入数据，如cin>>x>>y。

用户输入的是‘d’，而int score; scanf("%d", &score);要从里面吃掉一个整数，因为"d"不是整数，所以吃不掉，scanf返回0（表示没有吃掉任何输入），因为没有吃掉，所以输入字符流没有往前挪，下次再scanf时依然如此，不会要求用户从键盘重新输入

正确做法：

int score=-1;

while (true)

{

int ret=scanf\_s("%d", &score);

if (ret < 0) break;

else if (ret == 0 || score < 0 || score > 100){//要求输入成绩，取值在[0,100]，因此如此限制

char c;

while ((c = getchar()) != '\n' && c != EOF);

printf("score=%d, please input again:\n", score);

}

else break;

}

int x;cin >> x;存在同样的问题，正确做法

**int** x;

**while** (!(cin >> x))

{

    cout << cin .fail() << endl;

//cin.clear();//清除状态位

//char temp;

//while ((temp=cin.get())!=EOF && temp!='\n');  //可以

    //cin.clear();

    //cin.seekg(0, cin.\_Seekend); //无效

    cin.clear();  //清除状态位

    cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');  //删除错误输入

    cout << "input agagin():" << endl;   //提示用户再输入

}

cout << x << endl;

写文件时，向后seek

std::ofstream ofs("a.bin", std::ios::binary);

constexpr int size = 6;

char buf[size] = {0,1,2,3,4,5};

ofs.write(buf,size);

ofs.seekp(2,std::ios::\_Seekcur);

ofs.write(buf, size); //文件内容为00 01 02 03 04 05 00 00 00 01 02 03 04 05

int cin.get()：从流中提取一个字符，包括空格、回车、TAB在内的任意字符

istream& cin.unget()：将最后一个从流中提取的字符放回流中

istream& cin.putback(‘x’)：将字符’x’放回流中

char str[20];cin.get(str, 20);从流中读取最多19个字符，自动后边补‘\0’。 可以接收空格，遇到换行结束。但读取的字符串不包括换行，即‘\n’仍留在流中。如果键盘只输入abcd就回车，则没有问题，不会设置failbit，但换行没被读取。下次再cin.get(str,10)则第一个就碰到换行，直接返回，不会要求用户输入

getline(str, count)如果count太小没读完一行，则会设置failbit，“我本来打算读一行，但你给的count太小了，没读完一行，设置failbit提醒”

而get(str, count)如果count太小没读完一行，则不会设置failbit，“我只想读<=count-1个字节”

缓冲输入的流程是：键盘🡪输入缓冲区🡪程序中的输入队列

非缓冲输入的流程是： 键盘🡪程序中的输入队列

C语言有缓冲输入和非缓冲输入两种情况，默认缓冲输入。缓冲输入下，通过标准输入（键盘）输入的字符会先存在输入缓冲区（即键盘缓冲区）里，同时会回显到屏幕上。这样就可以在输入回车之前，进行输入的修改。按回车键，系统先将字符'\n'添加到键盘缓冲区，再将键盘缓冲区清空，将其中的所有字符发送到一个叫做输入队列的地方，供程序使用。非缓冲输入下，不需要输入回车来将输入发送给程序，而是键盘的任何输入都直接发送到程序的输入队列中。

cin要从程序的输入队列中取数据，

（1）有残留数据可读时则取残留数据，如果残留数据的类型不符合（如cin>>int，但输入队列此时是一个字符）则设置failbit，即cin.fail()返回true；

（2）输入队列中没有数据可读，则阻塞请求并等待键盘输入

cin的>>会跳过空格、tab、换行

cin读取一行整数的方法

（1）当行尾有空格、tab时，需要借助istringstream

char str[128];

cin.getline(str, 128);

std::istringstream iss(str);

while (iss >> temp) {cout << temp << endl;}

（2）当行尾没有空格、tab时：

int temp; //peek读取一个字符，但下次读取时仍会读到该字符

while (cin.peek() != '\n') {cin >> temp; cout << temp << endl;}

或着：while (std::cin.get() != '\n') { std::cin.unget();std::cin >> temp; }

cin的和unget()和putback(char)：

**void** test\_cin\_unget\_putback()

{

**char** ch1, ch2, ch3;

    std::cin >> ch1 >> ch2 >> ch3;  //即使读取的是char，也会跳过空白

    cout << ch1 << ch2 << ch3 << endl;

    std::cin.unget();    //把最后一个字符放回到流中

**char** ch4;

    std::cin >> ch4;

    cout << ch4 << endl;

    std::cin.putback('x');  //将字符‘x’放回到流中

**char** ch5, ch6;

    std::cin >> ch5 >> ch6;

    cout << ch5 << ch6 << endl;

    exit(0);

    /\*测试用例

    abcdef（键盘输入，最后敲回车）

    abc

    c

    xd

    请按任意键继续. . .

    \*/

}

cin.getline( )：

   当键盘缓冲区残留字符数(不包括’\n’)大于等于count-1时，则get函数只读取count-1个字符，而其余的字符仍然保存在缓冲区中，还可再对其进行读取；

    但是函数getline则不然，getline()会设置失效位(failbit)，并且关闭后面的输入，这个时候再用ch=cin.get()或者str=cin.getline()或者cin>>ch等是读取不到留在输入队列中的字符的。 可以用下面的命令来恢复输入：cin.clear()；因为clear()会清除失效位，打开输入。这个时候就可以 读取留在输入队列中的字符。也就是说使用cin.getline()时，最好将提取的字节数设置得很大，否则键盘输入的一行字符有剩余，之后无法继续用cin从键盘读取（除非使用clear重置失效位 \*/

**void** test\_cin\_getline2()

{

**char** str2[5];

    std::cin.getline(str2, 5);//从输入流中提取4个字符，自动在结尾添加‘\0’

    cout << str2 << endl;

    std::cin.clear();//清除failbit，如果没有此句，则接下来cin.getline(str, 5)什么都读取不到

    std::cin.getline(str2, 5);

    cout << str2 << endl;

    /\*测试：

    ABCDEF

    ABCD

    EF

    请按任意键继续. . .

    \*/

}

**scanf\_s：**

与 scanf 和 wscanf 不同，scanf\_s 和 wscanf\_s 要求您为某些参数指定缓冲大小。要为所有 c、C、s、S 或字符串控制集合 [] 参数指定大小。以字符为单位的缓冲大小是作为附加参数传递的。它紧跟在指向[缓冲区](https://www.zhihu.com/search?q=%E7%BC%93%E5%86%B2%E5%8C%BA&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22answer%22%2C%22sourceId%22%3A1511308838%7D)或变量的指针之后。例如，如果您要读取一个字符串，那个字符串的缓冲大小按如下方式传递：

char s[10];

scanf\_s("%9s", s, (unsigned)\_countof(s)); *//缓冲大小为10，宽度指定为9*

缓冲大小包含结尾空字符。您可以使用宽度指定字段确保读入的符号能容纳进缓冲区。当符号过大而无法容纳时，除非有宽度指定，否则没有字符会被写进缓冲区。

%c 格式不但可以用来读取单个字符，还可以用来读取多个字符而不由空字符结尾。因此这个格式也是需要指定缓冲大小的。

读取单个字符：

char c;

scanf\_s("%c", &c, 1);

# fstream：

如果open函数只有文件名一个参数，则是以读/写普通文件打开，即：

fstream file1;

file1.open("c:\\config.sys"); <=> file1.open("c:\\config.sys",ios::in|ios::out,0);

另外，fstream还有和open()一样的构造函数，对于上例，在定义的时侯就可以打开文件了：fstream file1("c:\\config.sys");

fstream有两个子类：ifstream(input file stream)和ofstream(outpu file stream)，ifstream以输入方式打开文件，而ofstream以输出方式打开文件

以追加方式打开文件： std::ofstream of("ofstreamTest.txt",std::ios::app);

ifstream::read读到/超过末尾

std::ifstream ifs("data.bin", std::ios::binary); //data.bin中存放5个字节：abcde

if (!ifs) {

cout << "Failed to open" << endl;

return -1;

}

vector<char> buf(6);

ifs.read(&buf[0],buf.size());

cout << (bool)ifs << " " << ifs.gcount() << endl; //0，5

ifs.read(&buf[0], 3);

cout << (bool)ifs << " " << ifs.gcount() << endl; //0 0

ifstream和ofstream采用RAII，不需要手动调用close()函数。

std::ifstream::seekg()是对输入文件定位，它有两个参数：第一个参数是偏移量，第二个参数是基地址。

对于第一个参数，可以是正负数值，正的表示向后偏移，负的表示向前偏移。而第二个参数可以是：

std::ifstream::\_Seekbeg：表示输入流的开始位置，也可以使用std::ios::beg

std::ifstream::\_Seekcur：表示输入流的当前位置，也可以使用std::ios::cur

std::ifstream::\_Seekend：表示输入流的结束位置，也可以使用std::ios::end

对输出流ofstream使用seekp()

ofstream ofs("nums.bin", ofstream::binary);   //  std::ofstream::binary表示写的是二进制文件

unsigned **char** a[] = {1,2,3,4,5};

ofs.write((**char**\*)a, 5);

ofs.close();

ifstream ifs("nums.bin", ifstream::binary);

unsigned **char** b[5]={ 0 };

ifs.read((**char**\*)b, 1);

ifs.seekg(2, std::ios::cur); //ifstream::\_Seekbeg

unsigned **char** c[5] = { 0 };

ifs.read((**char**\*)c, 2);

ifs.close();

**for** (auto i : b) cout << (**int**)i << " "; cout << endl; //1 0 0 0 0

**for** (auto i : c) cout << (**int**)i << " "; cout << endl; //4 5 0 0 0

功能：**peek函数用于读取并返回下一个字符，但并不提取该字符到输入流中**，也就是说，依然让该字符作为将要提取到输入流的下一个字符

可用于判断有没有到文件末尾while(cin.peek()!=EOF){}

不能对std::cin使用seekg()，可以对std::istringstream或std::ifstream使用

**使用ifs.seekg(2,std::ios::cur)如果本来要超出末尾，则最多跳至末尾，此时返回的ifs未设置failbit，之后再ifs.peek()返回EOF**

**使用带读取字节数的read()时，如果还没读够就碰到末尾，则已读的正确，ifs设置failbit**

ofstream ofs("C:/Users/myth/Desktop/test.bin", std::ios::binary);

if (!ofs) {

cout << "not open" << endl;

return -1;

}

char x = 3;

char y = 4;

char z = 5;

ofs.write(&x,1);

ofs.write(&y, 1);

ofs.seekp(0,std::ios::\_Seekbeg);

ofs.write(&z, 1); //最终结果为0x05 0x04

ifstream ifs("nums.bin", ifstream::binary); // nums.bin中存的是unsigned char型的1 2 3 4 5

unsigned char x;

while (ifs.peek() != EOF)

{

ifs >> x;

cout << "x read : "<<(int)x << endl;

ifs.seekg(2,std::ios::cur);//cin.\_Seekcur

cout << "failed ?:"<<cin.fail() << endl;

}

//输出：x read : 1 failed ?:0 x read : 4 failed ?:0

void test\_read\_to\_end()

{

ifstream ifs("nums.bin", ifstream::binary); //存储的是unsigned char型的1 2 3 4 5

unsigned char buf[16];

cout << ifs.read((char\*)buf, 8).fail() << endl;

for (auto i : buf)

cout << (int)i << " ";

cout << endl;

/\*

运行结果：

1

1 2 3 4 5 204 204 204 204 204 204 204 204 204 204 204

请按任意键继续. . .

\*/

}

ifstream ifs("nums.bin", ifstream::binary); // nums.bin中存的是unsigned char型的1 2 3 4 5

**if** (!ifs) { cout << "not read" << endl; **return**; }

**while**(ifs.peek()!=EOF)  //不能使用while (ifs) 读完最后一个字符后，ifs仍未设置failbit，while(ifs)仍为真，因此会多读一个字节

{

**char** x = ifs.get();

    cout << "x:" << (**int**)x << endl;

    cout << "!ifs" << !ifs << endl;

}

最佳写法为：char temp; while(ifs.get(temp)){}

使用while(ifs.peek()!=EOF)的运行结果：

x:1 !ifs0 x:2 !ifs0 x:3 !ifs0 x:4 !ifs0 x:5 !ifs0

使用while(ifs)的运行结果：

x:1 !ifs0 x:2 !ifs0 x:3 !ifs0 x:4 !ifs0 x:5 !ifs0 x:-1 !ifs1

# 从txt读取一行，再转为string、float、int：

ifstream& getline (ifstream& is, string& str)读取一行，存入string，不包括’\n’

istringstream是一个类，可将string转为int、float类型

#include<fstream> //ifstream

#include<sstream> //istringstream

void main()

{

std::ifstream fin("istringstreamTest.txt");

std::string line;

while (std::getline(fin, line))

{

std::cout << line << std::endl;

std::istringstream iss(line);

std::string name;

iss >> name;

std::vector<float> a;

std::vector<int> b;

float temp1;

float temp2;

while (iss >> temp1 >> temp2) //返回iss>>temp2的结果，因此如果只有1个数，则不会进入循环内，以空格分隔

{

a.push\_back(temp1);

b.push\_back(temp2);

}

for (auto i : a)

std::cout << i << " ";

std::cout << std::endl << "------------------------------\n";

for (auto i : b)

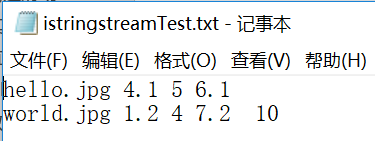
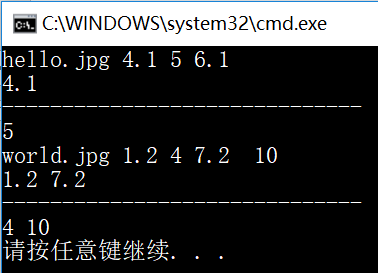
std::cout << i << " ";

std::cout << std::endl;

}

fin.close();

}

//vec转为字符串，无参构造std::ostringstream对象，然后调用<<写内容，再std::ostringstream::str()读出字符串

#include<sstream>

string to\_string(const vector<int>& vec)

{

std::ostringstream ost;

ost << '[';

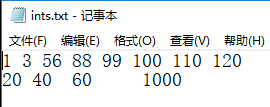
for (auto i : vec) ost << i;

ost << ']';

return ost.str();

}

//读取.dat中的int数据，多行都能读入，到容器vector<int>（或list、deque、set等）

//运行结果：1 3 56 88 99 100 110 120 20 40 60 1000

std::ifstream file("C:/Users/myth/Desktop/ints.txt");

if (!file) throw "open failed"; //ifstream重载了!运算符，打开文件失败则返回false

vector<int> data{ std::istream\_iterator<int>(file), std::istream\_iterator<int>() };

for (auto i : data) cout << i << " "; cout << endl;

或者：（也可读取多行） ifstream的operator>>会忽略空格，以空格为分割，因此如果ifs>>string对象，则含空格的字符串会被认为是多个子string

int temp;

while (ifs >> temp)

{

cout << temp <<" ";

}

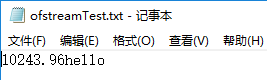
istream\_iterators<T>所依靠的operator>>函数进行的是格式化输入，这意味着每次调用的时候它们都必须做大量工作

istream\_iterator<char>对象使用operator>>来从输入流中读取单个字符。istreambuf\_iterator<char>对象进入流的缓冲区并直接读取下一个字符，不忽略任何字符。读取速度更快

# 将字符串、浮点数、整数写入txt：

类比标准输出流cout<<

1. #include<fstream>
2. #include<string>
3. **void** main()
4. {
5. **int** a = 102;
6. **float** b = 43.96;
7. std::string str = "hello";
8. std::ofstream of("ofstreamTest.txt");
9. of << a<<b<<str;
10. of.close();
11. }



# 字符串常量（字符串字面量）：

C++中，char\* x = "ekl"; //不正确，"ekl"的类型为const char [4]，VS2015可编译通过

const int arr[4]{ 1,2,3,4 };

arr[2] = 8; //error，不能修改

int\* p = arr; //error，const int \*" 类型的值不能用于初始化 "int \*" 类型的实体

C++中字符串字面量(string literal)的类型是const char[N]，然后数组会隐式转化成指向首元素的指针，即const char \*

char\*getmemory(void)

{

    char \*p=”hello world”;

    return p;

}//字符串常量”hello world”在Text段，p指向之，返回p，正确

//如果是char p[]=”hello world”;  return p;则p指向栈中内存，函数返回后栈销毁，指针悬挂，错误

二进制文件中，有.text段、.rodata段、.data段、.bss段……，.text存放代码，.rodata存放只读数据（如字符串常量），.data存放已初始化的全局变量和局部static变量，.bss存放未初始化的全局变量和局部static变量以及所有被初始化为0的全局变量和局部static变量

加载到内存后，.text段和.rodata段被合并为Text段

# 字符串比较：

传统的C字符串比较必须用strcmp函数，不能用==，否则比较的只是两个地址）

string与string比较、string与C字符串比较(const char\*隐式转换为std::string)，可以直接==

string sta = "hello";

string stb = "hello";

string stc = "hi";

cout << (sta == stb) << " " << (stb == stc) <<" "<<(stc=="hi")<< endl; // 1 0 1

int strcmp(const char\* s1,const char\* s2);            比较两个字符串。

int strncmp(const char\* s1,const char\* s2,size\_t n);       比较两个字符串前n位

比较规则：

从左到右逐个字符进行比较（ASCII值），直到出现不同的字符或遇到'\0'为止。

如果全部的字符相同，则认为两字符串相等，返回值为0；

如果出现不同的字符，则对第一次出现不同的字符进行比较，比较方法是以s1的第一个不同的字符减去s2的第一个不同的字符，以所得差值作为返回值(大于0，则返回1，小于0则返回-1)。

删除string中某个字符及其后的所有字符

string url\_str("udp://192.165.56.101:2290@192.165.56.101");

size\_t at\_pos = url\_str.find\_first\_of('@');

if (at\_pos != string::npos)//形如"udp://192.165.56.101:2290@192.165.56.101";

{

url\_str.erase(url\_str.begin() + at\_pos, url\_str.end());

}

cout << url\_str << endl;//udp://192.165.56.101:2290

# std::string和const char\*：

string str="abc";

char\* p=str.c\_str()

string s="abc";

如果string为empty，则string::c\_str()返回const char\*指向null字符

string str;

cout << str.empty() << endl; //1

cout << string("").empty() << endl; //1

cout << str.c\_str() << endl;

std::string str1;

std::string str2("");

std::cout << (void\*)str1.data() << std::endl; //000000DF671AF6A8

std::cout << (void\*)str2.data() << std::endl; //000000DF671AF680

std::cout << (int)\*str1.data() << std::endl; //0

std::cout << (int)\*str2.data() << std::endl; //0

std::cout << (str1==str2) << std::endl; //true

const char\*＝nullptr，cout的结果为？程序卡主但不崩溃

string()和string(“”)的c\_scr()返回的是否为nullptr?不是，返回指向“\0”字符串的指针，即\*c\_scr()＝’\0’

# 字符串和int、float等互转：

## 字符串转long、unsigend long、float、double等

#include<cstdlib>

float strtof(const char \*nptr, char \*\*endptr);

double strtod(const char \*nptr, char \*\*endptr);

long double strtold(const char \*nptr, char \*\*endptr);

long int strtol(const char \*nptr,char \*\*endptr,int base);

long long strtoll(const char \*nptr,char \*\*endptr,int base);

unsigned long strtoul(const char \*nptr,char \*\*endptr,int base);

unsigned long long strtoull(const char \*nptr,char \*\*endptr,int base);

double strtod(const char \*nptr, char \*\*endptr)：扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，到出现非数字或字符串结束时('\0')才结束转换，并将结果返回。若endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr传回

**const** **char**\* str = "-12.3years old";

**double** years = strtod(str, nullptr);

cout << years << endl; //-12.3

//或者

**char**\* mismatch;

years = strtod(str, &mismatch);

cout << years << "  " << \*mismatch << endl; //-12.3 y

long int strtol(const char \*nptr,char \*\*endptr,int base)：

1. 参数base代表采用的进制方式，范围从2至36，或0。如base值为10则采用10进制，若base值为16则采用16进制等。当base值为0时则自动检测，默认10进制，前缀’0’则8进制，前缀’0x’或’0X’则16进制。
2. 一开始strtol()会扫描参数nptr字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，再遇到非数字或字符串结束时('\0')结束转换，并将结果返回。若参数endptr不为NULL，则会将遇到不合条件而终止的nptr中的字符指针由endptr返回。
3. If the str is empty or does not have the expected form, no conversion is performed, and (if str\_end is not a null pointer) the value of str is stored in the object pointed to by str\_end.
4. If the converted value falls out of range of corresponding return type, a range error occurs (setting errno to ERANGE) and LONG\_MAX, LONG\_MIN, LLONG\_MAX or LLONG\_MIN is returned.
5. If no conversion can be performed, 0 is returned

**const** **char**\* s = "17";

**char**\* mismatch;

**long** i = strtol(s, &mismatch, 16);

cout << i << " " << (**int**)(\*mismatch) << endl; // 23  0

i = strtol(s, &mismatch, 10);

cout << i << endl; // 17

int main()

{

const char\* p = "10 200000000000000000000000000000 30 -40";

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "Parsing " << [std::quoted](http://en.cppreference.com/w/cpp/io/manip/quoted)(p) << ":**\n**";

for (;;)

{

// errno can be set to any non-zero value by a library function call

// regardless of whether there was an error, so it needs to be cleared

// in order to check the error set by strtol

errno = 0;

char\* p\_end;

const long i = std::strtol(p, &p\_end, 10);

if (p == p\_end)

break;

const bool range\_error = errno == [ERANGE](http://en.cppreference.com/w/cpp/error/errno_macros);

const [std::string](http://en.cppreference.com/w/cpp/string/basic_string) extracted(p, p\_end - p);

p = p\_end;

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << "Extracted " << [std::quoted](http://en.cppreference.com/w/cpp/io/manip/quoted)(extracted) << ", strtol returned " << i << '.';

if (range\_error) [std::cout](http://en.cppreference.com/w/cpp/io/cout) << " Range error occurred.";

[std::cout](http://en.cppreference.com/w/cpp/io/cout) << '**\n**';

}

}

Parsing "10 200000000000000000000000000000 30 -40":

Extracted "10", strtol returned 10.

Extracted " 200000000000000000000000000000", strtol returned 9223372036854775807. Range error occurred.

Extracted " 30", strtol returned 30.

Extracted " -40", strtol returned -40.

字符串转int：

int atoi(const char \*str ); atoi函数把字符串转换成整型数。其含义是ASCII to integer 的缩写

函数返回 int 值，此值由将输入字符作为数字解析而生成。 如果该输入无法转换为该类型的值，则返回值为 0

1. string str = "130";
2. **int** a = std::atoi(str.c\_str());
3. std::cout << a << std::endl;  //130

字符串转double：

double atof ( const char \* str );

参数：C string beginning with the representation of a floating-point number.

返回值：1. 转换成功返回doublel类型的值 2.不能转换，返回0.0。 3.越界，返回HUGE\_VAL

1. string sd = "3.77";
2. **double** dsd = std::atof(sd.c\_str());
3. std::cout << dsd << std:: endl;  //3.77

## int、float、double转std::string:

std::string to\_string( int value );

std::string to\_string( long value );

std::string to\_string( long long value );

std::string to\_string( unsigned value );

std::string to\_string( unsigned long value );

std::string to\_string( unsigned long long value );

std::string to\_string( float value );

std::string to\_string( double value );

std::string to\_string( long double value );

**int** aa = 108;

string  straa = std::to\_string(aa);

std::cout << straa << std::endl;  //108

**float** bb = 32.47f;

string  strbb= std::to\_string(bb);

std::cout << strbb << std::endl;  //32.470001

**double** cc = 43.96;

string  strcc = std::to\_string(cc);

std::cout << strcc << std::endl;  //43.960000

int转string：使用sprintf\_s()

// int sprintf\_s( char \*buffer, const char \*format, [ argument] … );

//参数列表

// buffer：char型指针，指向将要写入的字符串的缓冲区。

// format：格式化字符串。

// [argument]...：可选参数，可以是任何类型的数据。

// 返回值：字符串长度（strlen）

int aa = 30;

char c[8];

int length = sprintf(c, "%05X", aa);

cout << c << endl; // 0001E

# [std::string截取、替换、查找子串函数](http://www.cnblogs.com/catgatp/p/6407788.html)

* std::string string::substr(pos, n)    截取s中从pos开始的n个字符的子串，并返回。第二个参数默认是std::string::npos，即截取s中从pos开始到末尾的所有字符的子串
* s.replace(pos, n, s1)    用s1替换s中从pos开始（包括0）的n个字符的子串
* 查找子串

s.find(s1) 查找s中第一次出现s1的位置，返回size\_t类型，未找到则返回-1的补码转为size\_t的结果（即size\_t是无符号类型，因此(size\_t)-1即size\_t的最大值）

s.rfind(s1) 查找s中最后次出现s1的位置

s.find\_first\_of(s1) 查找在s1中任意一个字符在s中第一次出现的位置 s.find\_last\_of(s1) 查找在s1中任意一个字符在s中最后一次出现的位置

s.find\_first\_not\_of(s1) 查找s中第一个不属于s1中的字符的位置

s.find\_last\_not\_of(s1) 查找s中最后一个不属于s1中的字符的位置

* 添加

str1.append("\*"); 在str1的末尾添加星号

string str3 = "hello";

str3.insert(0, "world "); 在str3的开头添加，0为添加到的位置

* split

std::vector<std::string> split(std::string str, char ch) //从string中挖去所有ch后，剩余的片段

{

std::vector<std::string> ret;

size\_t start\_pos = str.find\_first\_not\_of(ch);

while (start\_pos != std::string::npos)//未找到则返回std::string::npos，std::string::npos是无符号类型size\_type的最大值

{

size\_t end\_pos = str.find\_first\_of(ch, start\_pos); //从第二个参数的位置开始找

ret.push\_back(str.substr(start\_pos, end\_pos - start\_pos)); //从第一个参数开始，长为第二个参数的子串

start\_pos = str.find\_first\_not\_of(ch, end\_pos);

}

return ret;

}

int main()

{

std::string str = "aa12a34aaa56a78a90aaa";

std::vector<std::string> ret = split(str, 'a');

for (const auto& i : ret)

{

cout << i << endl;//12 34 56 78 90

}

}

int colorX = static\_cast<int>(pp.X + 0.5f); /\*浮点数四舍五入转为int\*/

int colorY = static\_cast<int>(pp.Y + 0.5f);

头文件是一个类，如果想在该头文件中同时实现类的成员函数，则成员函数的实现必须在类里，不能是在类内声明，在类外、该头文件内实现

#if (defined(DEBUG) || defined(\_DEBUG))

…… （一些用于调试的打印语句等）

……

#endif

# <math.h>

#include<math.h>

pow(a,b);------a^b;

sqrt(a);------开方

tan()函数的周期为PI

double atan2(double y, double x) 对于任意不同时等于0的实参数x和y，返回的是原点到点(x, y)的方位角, 返回值的单位为弧度，取值范围为 https://gss0.bdstatic.com/-4o3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D47/sign=f817c2ad06e9390152028c3979ec2ec3/dcc451da81cb39db32331ddbd7160924aa18304f.jpg。点(x, y)位于1、2象限，则返回值为[0, pi]；点(x, y)位于3、4象限，则返回值为(-PI, 0)

atan2 的最大优势是可以正确处理 x=0 而 y≠0 的情况，而不必进行会引发除零异常的 y/x 操作

double atan(double x); 返回值以弧度表示，范围为 -π/2 到 +π/2 (-1.57079 到 1.57079)

acos()取值范围[0,PI]

asin()取值范围[-PI/2.0,PI/2.0]

atan2 的优点在于如果 x2-x1等于0 依然可以计算

cout << acos(-1.0) \* 180 / 3.1415926 << endl; //180°

cout << acos(1.0) \* 180 / 3.1415926 << endl << endl; //0°

cout << asin(-1.0) \* 180 / 3.1415926 << endl; //-90°

cout << asin(1.0) \* 180 / 3.1415926 << endl << endl; //90°

cout << atan(0) \* 180 / 3.1415926 << endl; //0°

cout << atan(DBL\_MAX) \* 180 / 3.1415926 << endl; //90°

# explicit

只对构造函数起作用，用来抑制隐式转换

按照默认规定，只有一个参数的构造函数也定义了一个隐式转换，将该构造函数对应数据类型的数据转换为该类对象，如下面所示：

class String

{

String ( const char\* p ); // 用C风格的字符串p作为初始化值

//…

}

String s1 = “hello”; //OK 隐式转换，等价于String s1 = String（“hello”）;

但是有的时候可能会不需要这种隐式转换，如下：

class String

{

       String ( int n ); //本意是预先分配n个字节给字符串

String ( const char\* p ); // 用C风格的字符串p作为初始化值

//…

}

下面两种写法比较正常：

String s2 ( 10 );   //OK 分配10个字节的空字符串

String s3 = String ( 10 ); //OK 分配10个字节的空字符串

下面两种写法就比较疑惑了：

String s4 = 10; //编译通过，也是分配10个字节的空字符串

String s5 = ‘a’; //编译通过，分配int（‘a’）个字节的空字符串

s4 和s5 分别把一个int型和char型，隐式转换成了分配若干字节的空字符串，容易令人误解。

为了避免这种错误的发生，我们可以声明显示的转换，使用explicit 关键字：

class String

{

       explicit String ( int n ); //本意是预先分配n个字节给字符串

String ( const char\* p ); // 用C风格的字符串p作为初始化值

//…

}

加上explicit，就抑制了String ( int n )的隐式转换，

下面两种写法仍然正确：

String s2 ( 10 );   //OK 分配10个字节的空字符串

String s3 = String ( 10 ); //OK 分配10个字节的空字符串

下面两种写法就不允许了：

String s4 = 10; //编译不通过，不允许隐式的转换

String s5 = ‘a’; //编译不通过，不允许隐式的转换

因此，某些时候，explicit 可以有效得防止构造函数的隐式转换带来的错误或者误解

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

explicit   只对构造函数起作用，用来抑制隐式转换。如：

class   A

{

A(int   a);

  };

  int   Function(A   a);

  当调用Function(2)的时候，2会隐式转换为A类型。这种情况常常不是程序员想要的结果，所以，要避免之，就可以这样写：

class   A

{

explicit   A(int   a);

};

int   Function(A   a);

这样，当调用Function(2)的时候，编译器会给出错误信息，这就避免了在程序员毫不知情的情况下出现错误

#include<stdlib.h> //stdlib.h是C标准函数库的头文件

有一定编程经验的人应该都能明白，在控制台输出信息是很耗时的。我们只是需要结果，不一定非要在控制中全部输出(这样还不便查看)，我们可以将结果保存到文件，这样也比输出到控制台快

出作用域，类的对象即被析构

class A

{

public:

~A() { cout << "destruct" << endl; }

};

void main()

{

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

{

A a;

cout << "i=" << i << endl;

}

}

运行结果：

i=0

destruct

i=1

destruct

i=2

destruct

请按任意键继续. . .

加一个大括号{}，即可改变作用域，让类对象提前析构

class A

{

public:

~A() { cout << "destruct" << endl; }

};

void main()

{

{

A a;

cout << "i=" << 10 << endl;

}

cout << "end" << endl;

}

运行结果：

i=10

destruct

end

请按任意键继续. . .

# vector内存分配：

class Test

{

public:

Test() { cout << "Test cons" << endl; }

~Test() { cout << "Test des" << endl; }

Test(const Test& a) { cout << "复制构造" << endl; }

};

void main()

{

cout << "---------------------------" << endl;

vector<Test> vec;

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

{

cout << "i=" << i <<":"<< endl;

Test p;

vec.push\_back(p);

}

cout << "---------------------------" << endl;

}

运行结果：

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

i=0:

Test cons

复制构造

Test des

i=1:

Test cons

复制构造

Test des

复制构造

Test des

i=2:

Test cons

复制构造

复制构造

Test des

Test des

复制构造

Test des

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

vector是这样分配内存的：

const size\_type len=old\_size !=0?2\*old\_size :1;

也就是说，vector的容量永远是大于或者等于size。而当内存不够的时候，要重新分配空间，于是，allocate(len)；再uninititalize\_copy();再析构原对象，再释放原空间。

# vector::push\_back()与std::move

1、std::move()

* 等价于类型转换static\_cast<T&&>();
* 将一个左值强制转化为一个右值引用（传入的参数是右值，则相当于没转化）
* 返回值是一个右值，将亡值
* 实际上 std::move 并不能移动任何东西

2、push\_back()两个重载版本

**void** **push\_back** (**const** T**&** value); *// (1)*

**void** **push\_back** (T**&&** value); *// (2)*

传入左值调用1，传入右值调用2

vector中存放的都是一个value的一个副本，**push\_back** (**const** T**&** value)内部调用拷贝构造函数产生副本，**push\_back** (T**&&** value) 内部调用移动构造函数产生副本

string st = "I love xing";

vector<string> vc;

vc.push\_back(move(st));

cout << vc[0].data() << endl;

**if** (!st.empty())

    cout << st.data() << endl;

cout << endl;

**std::move()并没有将st的资源移走，而是将左值转化为右值引用，不具名的右值引用是右值**，因此调用push\_back (T&& value)函数，该函数在内部调用了移动构造函数将st的资源移动到容器中。输出结果：I love xing

如果是将vc.push\_back(move(st))替换为vc.push\_back(st)，则输出结果：I love xing I love xing

# vector::erase()函数

vector::erase函数有两种重载形式：

iterator erase(iterator \_Where);

iterator erase(iterator \_First, iterator \_Last);

如果是删除指定位置的元素时：返回一个正向迭代器，指向删除元素下一个元素;如果是删除某区间的元素时：返回值一个正向迭代器，指向最后一个删除元素的下一个元素;

对于std::vector vec;vec在栈上（stack），而其中的元素T保存在堆上（heap）；

对于std::vector\* vec = new std::vector;vec和其中的元素T都保存在堆上

调用vector::erase(iter)后，iter失效，不能再被使用

移除vector中满足某个条件的元素，v1.erase(remove(v1.begin(), v1.end(), 10),v1.end());

错误代码：

**for** (vector<int>::iterator Iter = v1.begin(); Iter != v1.end(); Iter++)

    {

**if** (\*Iter == 10)   { v1.erase(Iter);  } //一旦删除，iter失效，不能再++

}

正确代码：

    for (Iter = v1.begin(); Iter != v1.end(); )

    {

        if (\*Iter == 10)   Iter=v1.erase(Iter);

        else   Iter++;

    }

vector数据成员中有三个指针 \_Myfirst、\_Mylast、 \_Myend;

大小：size=\_Mylast - \_Myfirst；

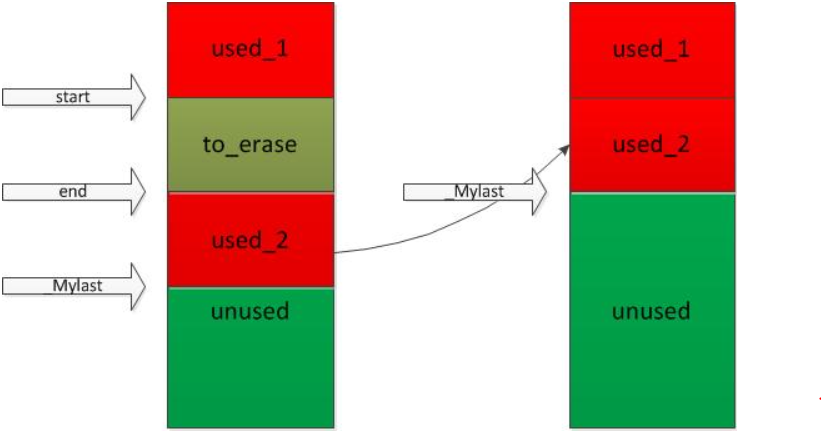
容量：capacity=\_Myend - \_Myfirst

vector::pop\_back()不会改变vector的capacity，调用析构函数

vector::resize(size\_type \_Newsize)向下调整大小：if (\_Newsize < size()) erase(begin() + \_Newsize, end());

vector::reserve(size\_type \_Count) ，如果目标的容量<=当前容量，则什么也不做

erase()操作：将后半部分的有效元素向前拷贝（调用拷贝赋值函数；如果有移动赋值函数，则调用移动赋值函数），并将后面空间的无效元素析构，并更新\_Mylast变量



**class** A

{

**public**:

    A(**int** x) :x\_(x) { cout << "构造x\_=" << x\_ << endl; }

    A(**const** A& a) :x\_(a.x\_) { cout << "拷贝构造x\_=" << x\_ << endl; }

    ~A() { cout << "析构x=" << x\_ << endl; }

    A& operator=(**const** A& other)

    {

        cout << "赋值运算符=重载，other.x=" << other.x\_ << endl;

        x\_ = other.x\_;

**return** \***this**;

    }

**private**:

**int** x\_;

};

vector<A> vecA{ A(0),A(1),A(2),A(3),A(4) };

vecA.erase(vecA.begin()+2);

先赋值运算符覆盖，再析构：

赋值运算符=重载，other.x=3

赋值运算符=重载，other.x=4

析构x=4

1. vecA={ A(5),A(6),A(7),A(8),A(9) };
2. std::remove\_if(vecA.begin(), vecA.end(), [](**const** A& a) {**return** a.x\_ % 2 == 1 ;});
3. **for** (auto i : vecA) cout << i.x\_ << " "; cout << endl;

赋值运算符=重载，other.x=6

赋值运算符=重载，other.x=8

拷贝构造x\_=6 //由于auto i:vecA，i是拷贝构造

6 析构x=6

拷贝构造x\_=8

8 析构x=8

拷贝构造x\_=7

7 析构x=7

拷贝构造x\_=8

8 析构x=8

拷贝构造x\_=9

9 析构x=9

# <algorithm> 中的remove函数：

头文件<algorithm>中有remove函数的两个重载版本：

std::remove(beg,end,val)

std::remove\_if(beg,end,unarryPred)

功能：remove()并不改变容器中元素的个数，通过覆盖从序列中“移去”元素，然后返回一个迭代器，该迭代器指向未移去的最后一个元素的下一个位置。remove移去的元素为等于val值，remove\_if移去的是当unarryPred为真的那些元素

remove实现机制是统一先找出哪些元素需要被覆盖，等全部找到以后在进行一次性覆盖（调用赋值运算符重载函数），而不是每找到一个需要覆盖的元素就将其进行覆盖。因为我们知道，对于顺序存储的容器如vectro deque 来说，移动覆盖代价是很大的，所以这么做完全是从性能上来考虑的

vector<**int**> key{ 10, 20, 10, 15, 12, 7, 9, 10 };

std::remove(key.begin(), key.end(), 10);

**for** (auto i : key) cout << i << " "; cout << endl;  //20 15 12 7 9 7 9 10

# vector::clear()函数

void clear() { erase(begin(), end()); } // 清除全部元素。注意并未释放空间

* vector中存储了对象的指针，调用clear后，并不会调用这些指针所指对象析构函数，因此要在clear之前调用delete；
* 如果vector存储的是对象，调用clear后，自建类型的对象(int之类的)直接删除，若是外部类型，则调用析构函数

# vector初始化、size()、capacity()：

vector<int> a ; //声明一个int型向量a

vector<int> a(10) ; //声明一个初始大小为10的向量，初始化为int()，即0

vector<int> a(10, 1) ; //声明一个初始大小为10且初始值都为1的向量

vector<int> b(a) ; //声明并用向量a初始化向量b

vector<int> b(a.begin(), a.begin()+3) ; //将a向量中0号到第2号(共3个)作为向量b的初始值

vector<int> data{1,3,5,6,7,8,5,10,99,5};

data= {2,4,6,8}; /\*会将之前的的所有元素清空，再赋新值\*/

* capacity：指容器在不分配新的存储空间的前提下它最多可以保存多少元素。
* size：指当前容器已经保存的元素的数目。
* vector 的reserve增加了vector的capacity，但是它的size没有改变！而resize时如果容量足够，则只改变size()，如果容量不够，则需要扩容，这时既改变了vector的capacity同时也增加了它的size！

reserve是容器预留空间，但在空间内不真正创建元素对象，所以在没有添加新的对象之前，不能引用容器内的元素。加入新的元素时，要调用push\_back()/insert()函数。

c.reserve(n)指分配至少能容纳n个元素的内存空间。

两个函数的参数形式也有区别的，reserve函数之后一个参数，即需要预留的容器的空间；resize函数可以有两个参数，第一个参数是容器新的大小， 第二个参数是要加入容器中的新元素，如果这个参数被省略，那么就调用元素对象的默认构造函数

vector扩容重新分配的过程

* allocate：分配新内存块，大小一般是原来的1.5倍或2倍（VS2017是1.5倍扩容）
* construct：把元素从容器的旧内存拷贝（或移动）到新内存
* destroy：销毁旧内存中的对象
* deallocate：回收旧内存

vector<**int**> vec;

cout << vec.size() << endl;  //0

vec.reserve(5);

cout << vec.size() << " " << vec.capacity() << endl;  //0 5

vec.resize(4);

cout << vec.size() << " " << vec.capacity() << endl;  //4 5

vec.push\_back('a');

cout << vec.size() << " " << vec.capacity() << endl;   //5 5

vec.push\_back('b');

cout << vec.size() << " " << vec.capacity() << endl;   //6 7

vec.resize(8);

cout << vec.size() << " " << vec.capacity() << endl;   //8 10

vec.resize(11);

cout << vec.size() << " " << vec.capacity() << endl;   //11 15

vec.reserve(100);

cout << vec.size() << " " << vec.capacity() << endl;   //11 100

vector<>::resize()向下调整大小时，会调用被舍弃对象的析构函数（相当于多次调用pop\_back()）

std::vector<A> va(2); //调用2次无参构造函数

map也可以用另一个map来初始化std::map<std::string, int> ma;std::map<std::string, int> ma2(ma);

两个vector的迭代器可以相减，不能相加

vector::pop\_back()调用vector存储元素的析构函数

vector::clear()：相当于一直pop\_back()直至vector::size()为0，不改变vector::capacity

vector::erase()：erase方法会在将对象删除后，将所有该对象之后的对象向前移动。此时会调用该对象的赋值运算符函数

vector::reserve(num)扩容时：分配新空间，因此先复制构造（或移动构造，当有移动构造时）再析构原内存处的对象

vector::resize(num)：改变容器的size，如果vector::capacity不足，则先调用vector::reserve(cnt)扩容

resize(num)：会调用对象的构造函数（不扩容且num大于当前size时）或析构函数（num小于当前size）

1. #include<iostream>
2. #include<vector>
3. #include<algorithm>
4. **class** A
5. {
6. **public**:
7. A() { std::cout << "A constr" << std::endl; }
8. ~A() { std::cout << "A destru" << std::endl; }
9. A(**const** A& a) { std::cout << "A copy constru" << std::endl; }
10. A(A&& a) noexcept { std::cout << "A move constru" << std::endl; }    //必须加noexcept，下文vector扩容时，才会调用该移动构造函数
11. A& operator = (**const** A& a) { std::cout << "A赋值运算符重载" << std::endl; **return** \***this**; }
12. **void** show() { std::cout << "hello" << std::endl; }
13. };
14. **void** main()
15. {
16. std::vector<A> va(2);    //调用2次无参构造函数
17. std::cout << "-----------1-----------" <<std::endl;
18. std::cout <<"1 va.capacity()"<< va.capacity() << std::endl;  //2
19. va.reserve(4); //调用2次复制构造（或移动构造，当有移动构造时）、2次析构
20. std::cout << "2 va.capacity()" << va.capacity() << std::endl;  //4
21. va.resize(4); //调用2次无参构造函数   //如果之前没有扩容为4，则会先扩容
22. std::cout << "------------2----------" << std::endl;
23. va.resize(3);      //一次析构
24. std::cout <<"--va.size() :"<< va.size() << std::endl;
25. va.pop\_back(); //一次析构
26. std::cout << "\*\*va.size() :" << va.size() << std::endl;
27. va.clear();         //两次析构
28. std::cout << "va.size() :" << va.size() << std::endl;
29. std::cout << "va.capacity() :" << va.capacity() << std::endl;
30. std::cout << "------------3----------" << std::endl;
32. std::vector<A> vb(4);
33. vb.erase(vb.begin() + 1, vb.begin() + 3);  //调用两次析构，一次赋值运算符函数
34. std::cout << "vb.size() :" << vb.size() << std::endl;
35. }

/\*

A constr

A constr

-----------1-----------

1 va.capacity()2

A move constru

A move constru

A destru

A destru

2 va.capacity()4

A constr

A constr

------------2----------

A destru

--va.size() :3

A destru

\*\*va.size() :2

A destru

A destru

va.size() :0

va.capacity() :4

------------3----------

A constr

A constr

A constr

A constr

A赋值运算符重载

A destru

A destru

vb.size() :2

A destru

A destru

请按任意键继续. . .

\*/

整数次方不要用pow，i\*i比pow(i,2)快8倍，i\*i\*i比pow快40倍

在OpenCV中，可以用RNG类来产生均匀分布和正态分布(高斯分布)的随机数。RNG的英文全称是Random number generator

unsigned int与int混合运算时，操作数都转换为unsigned int

#include<iostream>

#include<vector>

#include<algorithm> /\*包含for\_each函数\*/

using namespace std;

void main()

{

cout << UINT\_MAX << endl;

cout << 0u - 1 << endl; /\*4294967295：无符号整型0减1变为无符号整型最大值\*/

cout << 0 - 1 << endl; //-1

vector<int> stu;

/\*vector赋值方法：\*/

for (int i = 0; i < 3; i++) /\*方法1\*/

{

stu.push\_back(i \* 2);

}

stu = {2,4,6,8}; /\*方法2，会将之前的的所有元素清空，再赋新值\*/

cout << "typeid(vector<int>::size\_type).name() = " << typeid(vector<int>::size\_type).name() << endl; /\*打印结果：unsigned int\*/

//vector元素遍历方法：

for (int i = 0; i < (int)stu.size(); i++) cout << stu[i] << " "; cout << endl; /\*方法1： vector::size()返回类型为size::type,是无符号整型,因此强制转换为int，使<前后的数据类型一致\*/

for (auto i : stu) cout << i << " "; cout << endl;/\*方法2:i即为容器中每一个元素\*/

for (vector<int>::iterator i = stu.begin(); i != stu.end(); i++) cout << \*i << " "; cout << endl;/\*方法3:i为指向容器中元素的指针\*/

for\_each(stu.begin(), stu.end(), [](int i){cout << i << " "; }); cout << endl; /\*方法4，借用algorithm头文件中的for\_each函数和lambda表达式\*/

}

STL算法中for\_each是个模板函数，原型：

template<class InputIterator, class Function>

Function for\_each(InputIterator first, InputIterator last, Function fn)

{

while (first!=last) {

fn (\*first);

++first;

}

return fn; //

}

示例：

void printFun(int i)

{

cout << i << " ";

}

auto print = [](int i){ cout << i << " "; } //lambda 表达式

vector<int> v{1,2,3,4,5};;//容器的初始化

for\_each(v.begin(), v.end(), printFun);//输出1 2 3 4 5

for\_each(v.begin(), v.end(), print);//输出1 2 3 4 5

# std::map

#include<iostream>

#include<map>

#include<string>

**using** std::string;

**using** std::endl;

**using** std::cout;

**void** main()

{

    //map也可以用另一个map来初始化，拷贝构造或移动构造std::map<std::string, int> ma;std::map<std::string, int> ma2(ma);

    std::map<string, **int**> m{ {"meng",32} };

    m={ { "zha",179 },{ "wang",180 } };  //重新给整个map赋值

    m.insert(std::make\_pair<string,**int**>("Li", 21));  //增

    cout << m.begin()->first << "   " << m.begin()->second << endl;  //查

    cout << m["zha"] << endl;  //查，如果没有键为"zha"，则该句无意中增加了{"zha",0}键值对

**for** (const auto& i : m) cout << "(" << i.first << ": " << i.second << ")" << "  "; cout << endl;

    std::map< string, **int** >::iterator it = m.find("wang");  //以key找

    cout << "look for result:" << it->first << "  " << it->second << endl;

    auto iter = m.find("chen");   //find()函数如果找不到，则返回超出末端迭代器

**if** (iter == m.end()) cout << "no exist" << endl;

    m.erase("zha");  //删，传入key

**for** (auto i : m) cout << "(" << i.first << ": " << i.second << ")" << "  "; cout << endl;

    auto d = m.find("wang");

    m.erase(d);  //删，传入迭代器

**for** (auto i : m) cout << "(" << i.first << ": " << i.second << ")" << "  "; cout << endl;

m.erase("zhou");

}

使用count，返回的是被查找元素的个数。如果有，返回1；否则，返回0。注意，map中不存在相同元素，所以返回值只能是1或0

# STL容器综述：

顺序容器包括：vector、list、deque（队列）顺序容器的元素排列次序与元素值无关，而是由元素添加到容器里的次序决定

STL list底层是双向链表, 允许在序列中任何一处位置以常量耗时插入或删除元素,可以双向迭代，访问复杂度O(n)，即不支持随机访问，没有下标运算符

list有size()、empty()、begin()、end()、rbegin()、rend()、push\_back()、push\_front()、pop\_back()、pop\_front()、iterator insert()、iterator erase()、void remove()、sort()稳定排序、reverse()、unique()、merge(std::list)、

splice(iterator position, list& x, iterator first, iterator last)

lst = { 3,1,5,9,7}; sort(std::greater<int>()); //默认是由小到大，如果是用户定义类型，则需要重载operator <

list::erase()有两种重载形式，同vector，可删除一个元素或连续的多个元素，返回迭代器指向最后一个删除元素的下一个元素

iterator erase(iterator \_Where);

iterator erase(iterator \_First, iterator \_Last);

调用lsit::erase(iter)后，iter失效，不能再被使用

**for** (auto i = lst.begin(); i != lst.end();) //删除list中满足某个条件的所有元素

{

**if** (\*i == 13) i = lst.erase(i); //erase()返回迭代器，指向被删除元素的下一个元素，同vector

**else** i++;

}

list::remove()有两种重载形式：

* list::remove(8); //删除所有值为8的元素
* list::remove\_if([](int x) {return x>=15; });//删除所有满足条件(>=15)的元素

list::insert()有多种重载形式，同vector，迭代器指定插入位置，可插入一个元素或多个元素（迭代器指定或{}指定），返回一个迭代器指向插入的第一个元素

std::vector<**int**> vec(lst.begin(),lst.end()); //19 17 15 13 13 11 10 8 5 4 3

auto k = vec.insert(vec.begin() + 3, {88,99});//插入多个元素，返回指向插入的第一个元素的迭代器，也可用迭代器指定范围代替{,}

**for** (auto i : vec)cout << i << " "; cout << endl;//19 17 15 88 99 13 13 11 10 8 5 4 3

vec.insert(k, 100); //插入一个元素

**for** (auto i : vec)cout << i << " "; cout << endl;//19 17 15 100 88 99 13 13 11 10 8 5 4 3

void list::assign( input\_iterator start, input\_iterator end );

void list::assign( size\_type num, const TYPE &val ); assign()重新设置元素

list::assign()函数以迭代器start和end指示的范围为list赋值或者为list赋值num个以val为值的元素。

list::unique()//相邻位置有重复元素，则只保留一个

#include<iostream>

#include<list>

**using** std::cout;

**using** std::endl;

**int** main()

{

    //初始化、赋值

    std::list<**int**> ls(5, 3); //3 3 3 3 3

    std::list<**int**> lst{ 5, 1, 3, 7, 3, 9 };

    lst = { 50, 10, 30, 70, 30, 90 };

**int** a[7] = { 15, 11, 13, 17, 13, 19, 18 };

    lst.assign(a,a+5);

**for** (auto i : lst)cout << i << " "; cout << endl; //15 11 13 17 13

    //erase()

    std::list<**int**>::iterator it = lst.begin();  //list不支持随机存取，因此没有迭代器没有[]操作和+n操作

    lst.erase(it); //如果再it++，则程序崩溃，erase(list<>::iterator x)后迭代器x失效,不能再++，同vector

**for** (auto i : lst)cout << i << " "; cout << endl; //11 13 17 13

    auto start = ++lst.begin();

    auto end = --lst.end();

    lst.erase(start, end);  //lst删除迭代器指定区间的所有元素，返回指向被删除的最后一个元素的下一个元素的迭代器

**for** (auto i : lst) cout << i << " "; cout << endl; //11 13

    lst = { 15, 11, 13, 17, 13, 19, 13 };

**for** (auto i = lst.begin(); i != lst.end();) //删除list中满足某个条件的所有元素

    {

**if** (\*i == 13) i = lst.erase(i); //erase()返回迭代器，指向被删除元素的下一个元素，同vector

**else** i++;

    }

**for** (auto i : lst)cout << i << " "; cout << endl; //15 11 17 19

    //remove()

    cout << "--------------------" << endl;

    lst = { 15, 11, 13, 17, 13, 19, 10 };

    lst.remove(13);

**for** (auto i : lst)cout << i << " "; cout << endl; //15 11 17 19 10

    lst.remove\_if([](**int** x) {**return** x >= 15; });

**for** (auto i : lst)cout << i << " "; cout << endl; //11 10

    //insert

    lst = { 15, 11, 13, 17, 13, 19, 10 };

    auto insertPos = ++lst.begin(); insertPos++; insertPos++;

    insertPos=lst.insert(insertPos,8);  //返回迭代器，指向新插入的元素

**for** (auto i : lst)cout << i << " "; cout << endl; //15, 11, 13, 8, 17, 13, 19, 10

**int** b[] = {1,2,3,4,5};

    auto pos=lst.insert(insertPos, b+2,b+5); //插入迭代器指定范围内的元素,返回迭代器，指向新插入的第一个元素 ，也可用初始化列表代替迭代器，即lst.insert(insertPos, {3,4,5})

**for** (auto i : lst)cout << i << " "; cout << endl; //15, 11, 13, 3, 4, 5, 8, 17, 13, 19, 10

    cout << \*pos << endl;  //3

    lst.reverse();

**for** (auto i : lst)cout << i << " "; cout << endl;//15, 11, 13, 3, 4, 5, 8, 17, 13, 19, 10

    //sort

    lst = { 15, 11, 13, 3, 4, 5, 8, 17, 13, 19, 10 };

    lst.sort();  //由小到大排序

**for** (auto i : lst)cout << i << " "; cout << endl;//3 4 5 8 10 11 13 13 15 17 19

    lst.sort([](**int** a, **int** b) {**return** a > b; }); //由大到小排序

**for** (auto i : lst)cout << i << " "; cout << endl;//19 17 15 13 13 11 10 8 5 4 3

}

关联容器包括：map（映射）、set（集合）、multimap（多重映射）、multiset（多重集合）。元素是有序的集合，默认在插入时自动按升序排列。内部实现是采用红黑树。list 是逐个搜索，它搜索的时间是跟容器的大小成正比，而关联容器 查找的复杂度基本是Log(N)

关联容器对元素的插入和删除操作比vector 要快，因为vector 是顺序存储，而关联容器是链式存储；比list 要慢，是因为即使它们同是链式结构，但list 是线性的，而关联容器是二叉树结构

map即键值对的集合，对于键类型，唯一的约束就是必须支持 < 操作符

map中元素默认按键值从小到大排列

std::map<int, char, std::less<int>> sbmap{ {179,'z'},{180,'d'} }; //第三个参数可省略，默认按key值由小到大

std::map<int, char, std::greater<int>> bsmap{ { 179,'z' },{ 180,'d' } }; //由大到小

for (auto i : sbmap) cout << "("<<i.first<<","<<i.second<<")" << " "; cout << endl; //(179,z) (180,d)

for(auto i: bsmap) cout << "("<<i.first<<","<<i.second<<")" << " ";cout << endl;//(180,d) (179,z)

map中使用下标存在一个很危险的副作用：如果该键不在 map 容器中，那么下标操作会插入一个具有该键的新元素。所以map 容器提供了两个操作：count 和 find，用于检查某个键是否存在而不会插入该键。

m.insert(std::pair<,> kv) 增加一个键值对

m.erase(key) //删除map中某个元素，传入key

m.erase(map<,>::iterator iter) //删除map中某个元素，传入迭代器

m.count(k) 返回 m 中 k 的出现次数

m.find(key) 如果m容器中存在键为k的元素，则返回指向该元素的迭代器。如果不存在，则返回超出末端迭代器

set的含义是集合，它是一个有序的容器，里面的元素都是排序好的，支持插入，删除，查找等操作，就像一个集合一样。所有的操作的都是严格在logn时间之内完成，效率非常高。set和multiset的区别是：set插入的元素不能相同，但是multiset可以相同

set<int> iset(ivec.begin(), ivec.end());//如果ivec中有重复元素，自动删除重复的，且有序排列于set

set有size()、empty()、begin()、end()、rbegin()、rend()、set<>::iterator erase()、void insert()、find()、count()、lower\_bound()、upper\_bound()

set 容器不提供下标操作符。为了通过键从 set 中获取元素，可使用 find运算。

如果只需简单地判断某个元素是否存在，同样可以使用 count 运算，返回 set 中该键对应的元素个数。

当然，对于 set 容器，count 的返回值只能是1（该元素存在）或 0（该元素不存在）

set默认是从小到大排列值，定义时set<int,greater<int>> iset;则此时默认是从大到小排列值

void set::insert({ 1,2,4,6,8,9,10 });//返回类型是void，参数可以是一个数，或两个迭代器指定的一个范围，或初始化列表initializer\_list

list::insert()和vector::insert()，返回一个迭代器指向插入的第一个元素，参数是指定插入位置的迭代器+（一个数，或两个迭代器指定的一个范围，或初始化列表initializer\_list）

set::lower\_bound返回值一般是>= 给定val的最小指针（iterator）。set::upper\_bound返回值则是 > 给定val的最小指针（iterator）

myset = { 1,4,2,6,9,14 }; //1 2 4 6 9 14

cout << \*myset.lower\_bound(4) << endl;//4

cout << \*myset.upper\_bound(4) << endl;//6

std::set<int, std::less<int>> sbset{1,3,5,2,4,6}; //第二个参数可省略，默认由小到大

std::set<int, std::greater<int>> bsset{ 1,3,5,2,4,6 }; //由大到小

for (auto i : sbset) cout << i << " "; cout << endl; //1 2 3 4 5 6

for (auto i : bsset) cout << i << " "; cout << endl; //6 5 4 3 2 1

greater和less是头文件<xfunctional>中的模板类，重载了()运算符，即为函数对象

**int** main()

{

    std::set<**int**> st{ 10,3,15,3,7 };

**for** (auto i : st) cout << i << " "; cout << endl;  //自动去除重复且升序排列，3 7 10 15

    st = { 8,3,5,6 };

**int** a[] = { 80,30,50,60,70};

    std::set<**int**> myset(a + 1, a + 3);

**for** (auto i : myset) cout << i << " "; cout << endl; //30 50

    myset = { 80,30,50,60,70 };  //set没有assign函数

    //set::erase的用法和注意事项同vector、list

    auto eraseIt =++myset.begin();  //set<>::iterator不支持[]和+n操作

    auto next= myset.erase(eraseIt); //erase(iter)后迭代器iter失效,返回迭代器指向被删除的最后一个元素的下一个元素

**for** (auto i : myset) cout << i << " "; cout << endl; //30 60 70 80

    cout << "next ele is:" << \*next << endl;  //60

    cout << \*(myset.erase(++myset.begin(), --myset.end())) << endl;//80

**for** (auto i : myset) cout << i << " "; cout << endl; //30 80

    myset = { 1,4,3,2,5,6,8,7,9 };

**for** (auto i = myset.begin(); i != myset.end();)

    {

**if** (\*i % 2 == 0) i = myset.erase(i);

**else** i++;

    }

**for** (auto i : myset) cout << i << " "; cout << endl; //1 3 5 7 9

    myset.insert({ 1,2,4,6,8,9,10 });//返回类型是void，参数可以是一个数，或两个迭代器指定的一个范围，或初始化列表initializer\_list

    myset.insert(11);

    std::list<**int**> lst{ 20, 30, 40 };

    myset.insert(lst.begin(),--lst.end());

**for** (auto i : myset) cout << i << " "; cout << endl; //1 2 3 4 5 6 7 8 9 10 11 20 30

    //myset和vector没有remove()函数

    //set的元素不能修改，只能先erase再insert

    //set容器不提供下标操作符,因为底层红黑树不能随机存取；map有下标运算符，因为要由key得到value

    //set中的键为const类型，在获得指向set中某元素的迭代器后，只能对其做读操作，而不能做写操作

}

vector 的查询性能最好，并且在末端增加数据也很好，除非它重新申请内存段；适合高效地随机存储。

list 是一个链表，任何一个元素都可以是不连续的，但它都有两个指向上一元素和下一元素的指针。所以它对插入、删除元素性能是最好的，而查询性能非常差；适合 大量地插入和删除操作而不关心随机存取的需求。

deque 是介于两者之间，它兼顾了数组和链表的优点，它是分块的链表和多个数组的联合。所以它有比list 好的随机访问性能，有比vector 好的插入、删除性能。 如果你需要随即存取又关心两端数据的插入和删除，那么deque 是最佳之选

STL 中只有vector、map、deque 可以通过类数组的方式操作元素，即如同ele[1] 方式

# 复制字符串

1. **char**\* p = "hello";
2. **char**\* str = **new** **char**[strlen(p) + 1];
3. strcpy\_s(str, strlen(p)+1,p);

C++中构造函数也可以是private类型

已知p是一个指向类X数据成员m的指针,s是类X的一个对象。如果要给m赋值为3, 则s.\*p=3

polymorphism:多态性[ˌpɒlɪ'mɔ:fɪzəm]

encapsulation：封装性

template<classT1,classT2> 其中，class也可置换为typename

内联函数在编译时是将该函数的目标代码插入每个调用该函数的地方

::x = 6; //通过作用域解析运算符::引用全局变量x,并且改变其值

默认参数与重载函数尽量不要同时采用

在多继承中，公有派生和私有派生对于基类成员的派生类中的可访问性与单继承的规则完全相同

一个虚函数无论被公有继承了多少次，它仍然是虚函数;

syntax：语法

ambiguity：二义性

# 算法技巧：

(1)替换字符串中的空格：知道替换后字符串长度后，从后前边遍历替换源字符串

(2)合并两个有序数组并统计逆序对数目：从后向前遍历

# unsigned与int转换

1、当表达式中存在unsigned int和int时，所有的操作数都自动转换为unsigned int

如：

unsigned int c = 20;

int d = -130;

cout <<"c+d:"<< c + d <<endl; //4294967186

同样：

unsigned int a=20;

signed int b=-130;

a>b? 还是 b>a？实验证明b>a

在运算之前，a＝20，b被转化为4294967166，所以a＋b＝4294967186

* 将一个unsigned型数据赋给一个占据同样长度[存储](http://storage.chinabyte.com/)单元的整型变量时(如：unsigned→int、unsigned long→long,unsigned short→short)，**原值照赋**，内部的存储方式不变，但外部值却可能改变。
* 将一个非unsigned整型数据赋给长度相同的unsigned型变量时，内部存储形式不变，外部值却可能改变，**外部表示时总是无符号的**

2、补码

* 计算机中数据用补码表示，最高位是符号位，为1时表示负值，为0时表示正值
* 补码的表示方法是:

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1.

[+1] = [00000001]原 = [00000001]反 = [00000001]补

[-1] = [10000001]原 = [11111110]反 = [11111111]补

int型-130转换为为unsigned int的结果：2^31+(2^31-1)-130+1=4294967186

long long int x = pow(2, 16)\*pow(2, 16);

cout << "2^32 is:" << x << endl; //4294967296

cout << pow(2, 32) - 1 << endl; //4.29497e+09

cout << pow(2, 16) << endl;

int b = -130;

cout << (unsigned int)(b) << endl; //4294967166

cout << x-130 << endl; //4294967166

cout <<"(unsigned int)((int)(-1)):"<< (unsigned int)((int)(-1)) << endl; //4294967295

# sizeof运算符

sizeof是编译时运算符，编译时就确定了

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

int i;

for (i = -1; i < sizeof(a) / sizeof(a[0]); i++) printf("%d ", a[i]); cout << endl;

结果打印0条语句，分析：

sizeof返回类型为size\_t，size\_t在当前环境下实际为unsigned long long；

i < sizeof(a) / sizeof(a[0])，i为int型，精度小于unsigned long long，i由int型转为unsigned long long，-1转换为2^sizeof(unsigned long long)-1，大于5，不满足循环条件

# malloc/free与new/delete的区别

* malloc、free是标准库函数，new、delete是运算符。new分配空间、调用构造函数，delete调用构造函数、释放空间。对于非内部数据类型的对象而言，光用malloc/free无法满足动态对象的要求。
* new和delete最终调用malloc和free

函数malloc 的原型如下：

**void \* malloc (size\_t size);**

用malloc 申请一块长度为length 的整数类型的内存，程序如下：

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

我们应当把注意力集中在两个要素上：“类型转换”和“sizeof”。

1、malloc 返回值的类型是void \*，所以在调用malloc 时要显式地进行类型转换，将void \* 转换成所需要的指针类型。

2、 malloc 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。

函数free 的原型如下：

void free ( void \* memblock );

为什么free 函数不象malloc 函数那样复杂呢？这是因为指针p 的类型以及它所指的内存的容量事先都是知道的，语句free(p)能正确地释放内存。如果p 是nullptr指针，那么free对p 无论操作多少次都不会出问题。如果p 不是nullptr指针，那么free 对p连续操作两次就会导致程序运行错误

如果用new 创建对象数组，那么只能使用对象的无参数构造函数。例如

Obj \*objects = new Obj[100];       // 创建100 个动态对象

不能写成

Obj \*objects = new Obj[100](1);        // 创建100 个动态对象的同时赋初值1

# new[]、delete[]的原理：

C++ 语言标准库的库函数，原型分别如下：

void\* operator new(size\_t); //allocate an object

void\* operator delete (void \*); //free an object

void\* operator new[] (size\_t); //allocate an array

void\* operator delete[] (void \*); //free an array

* int\* p=new int(10)时：

1. 调用对应的标准库函数operator new申请内存，传入的参数为申请的内存大小；
2. 对内存进行初始化，初始化为10（如果是int p=new int;则不进行初始化）；
3. 返回int\*型指针，指针指向分配空间的起始地址

* int\* p=new int[5]时：

1. 调用标准库函数operator new[]申请内存，传入的参数为申请的内存总大小，即5\*sizeof(int)；
2. 不进行初始化
3. 返回int\*型指针，指针指向分配空间的起始地址

* A\* p=new A时（A为自定义的class）：

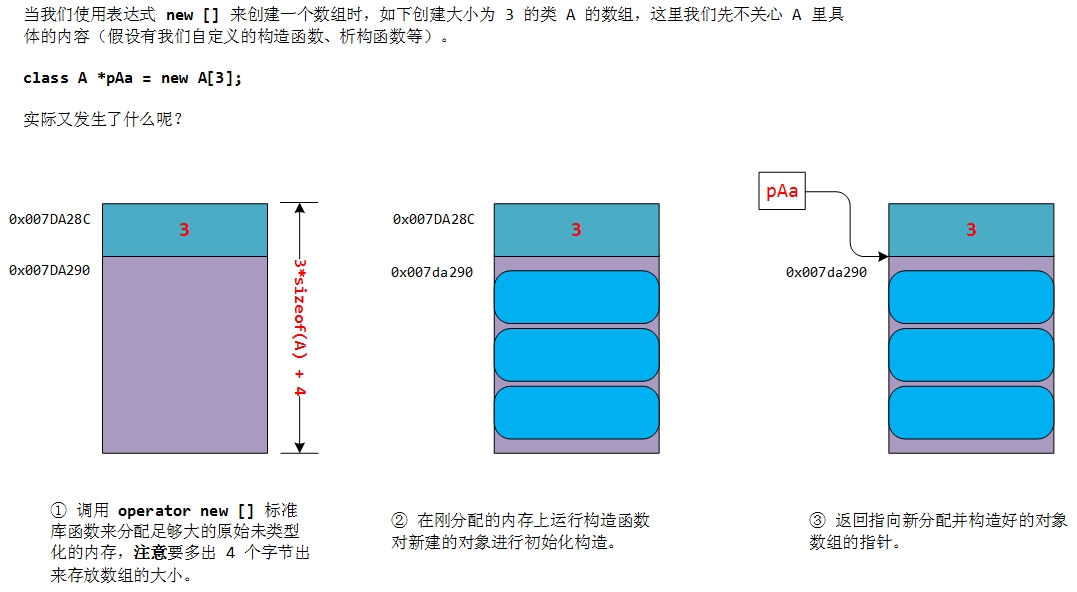
1. 调用标准库函数operator new申请内存，传入的参数为申请的内存大小，即对象所占空间；
2. 调用构造函数；
3. 返回A\*型指针，指针指向对象，也即分配空间的起始地址

调用delete p时：

1. 调用1次析构函数
2. 调用operator delete函数，传入的参数是对象的指针pAa

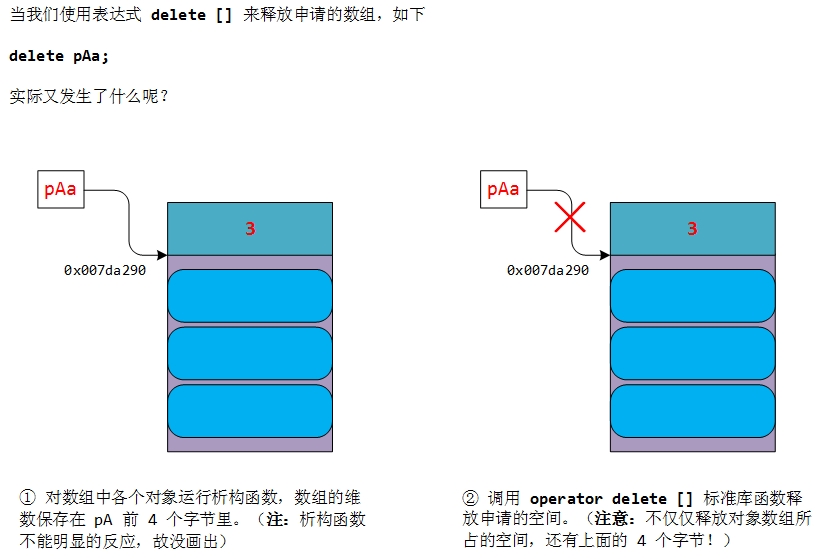
* A\* pAa=new A[5]时：

1. 调用对应的标准库函数申请内存，传入的参数为申请的内存总大小，5个对象+额外4字节；
2. 调用5次构造函数
3. 返回A\*型指针，指针指向第一个对象（但并不是分配空间的起始地址）



从这个图中我们可以看到申请时在数组对象的上面还多分配了4个字节用来保存数组的大小，但是最终返回的是对象数组的指针，而不是所有分配空间的起始地址。

这样的话，释放就很简单了：delete [] pAa; 而不需要写成delete [5] pAa（当然这样写也没问题）



这里要注意的两点是：

* 调用析构函数的次数是从数组对象指针前面的4个字节中取出；
* 传入 operator delete[] 函数的参数不是数组对象的指针 pAa，而是 pAa 的值减 4

new/delete 、new []/delete[] 配对使用：

如果是带有自定义析构函数的类，用new []来创建类对象数组，而用delete来释放会发生什么？用下面的例子来说明：

class A \*pAa = new class A[3];

delete pAa;

那么 delete pAa时 做了两件事：

1. 调用一次 pAa 指向的对象的析构函数；
2. 调用 operator delete(pAa); 释放内存。

显然，这里只对数组的第一个类对象调用了析构函数，后面的两个对象均没调用析构函数

(1)类对象中申请了大量的内存需要在析构函数中释放，而你却在销毁数组对象时少调用了析构函数，这会造成内存泄漏

(2)直接释放pAa指向的内存空间，这个总是会造成严重的段错误，程序必然会奔溃！因为分配的空间的起始地址是 pAa 指向的地方减去4个字节的地方。你应该传入参数设为那个地址！pAa不是某次malloc的返回值，因而它无法被free处理，因为free只接受malloc一系列动态分配函数的返回值,否则出现未定义行为。

# const变量与常量表达式：

C++中的常量表达式包括：字面值常量如42、’a’、42.0，字符串常量，const修饰的常量表达式

字面值常量从其字面形式即可识别其类型，如42是int、’a’是char、42.0是double类型。

常量折叠（const folding）。编译器对简单的const常量表达式进行优化，在预编译期像#define一样进行替换。

全局变量const char\* pstr="hello C++";pstr指向常量区，因为"hello C++"为字符串常量存储在常量区（即.rodata），可用objdump -s -j .rodata main.o查看.rodata段的内容。pstr自身是变量，已被初始化，存在.data段

      常量表达式指初始化后值不会改变，且在编译阶段即可计算出值的表达式。仅一部分const对象是常量表达式，即由常量表达式初始化的const对象是常量表达式。也叫“编译期常量”。

const int a=表达式expr; a是一个const变量，是一个只读变量

const变量不一定是常量表达式，仅当const变量在编译期其值就可以确定的时候，const变量才是常量表达式。expr如果是常量表达式，则a是常量表达式，否则a不是常量表达式，仅是个const变量

      C++ 11 中允许将变量声明为constexpr类型以使编译器在编译时检查该变量是否是常量表达式。**一般而言，如果你认定变量是一个常量表达式，那就把它声明成 constexpr 类型。constexpr 变量在定义时必须初始化**  
例如：

   const int a = 12;    a是常量表达式

   const int b = a+1;  b也是常量表达式

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

   constexpr int c = getsize();  将会报错

# constexpr:

const只能用于非静态成员函数而不是所有函数。它保证成员函数不修改任何非静态数据

constexpr修饰变量，则要求变量是常量表达式

constexpr修饰函数（称之为constexpr function，常值函数），则如果传入的实参是常量，则函数表达式是常量表达式；如果传入的实参不是常量，则函数表达式变为普通函数（相当于没有constexpr，而不是相当于const）

constexpr修饰函数，要求：

* 函数中只能有一个return语句(但允许包含typedefs、 using declaration && directives、静态断言等)
* 只能调用其它constexpr函数
* 只能使用常量表达式，如函数体中不能有int x=10;

constexpr修饰类的构造函数，即保证如果提供给该构造函数的参数都是constexpr，那么产生的对象中的所有成员都会是constexpr，该对象也就是constexpr对象了，可用于各种只能使用constexpr的场合。注意，constexpr构造函数必须有一个空的函数体，即所有成员变量的初始化都放到初始化列表中

constexpr **int** multify(**int** x, **int** y)

{

**return** x\*y;

}

constexpr **int** factorial(**int** n)  //递归函数是constexpr

{

**return** n > 0 ? n\*factorial(n - 1) : 1;

}

**class** Circle

{

**public**:

    constexpr Circle(**int** x\_, **int** y\_, **int** radius\_):x(x\_),y(y\_),radius(radius\_){}

    constexpr **int** getArea()  const

    {

**return** radius \* 2 \* 3;

    }

**void** set\_x(**int** n) { x = n; }

**private**:

**int** x;

**int** y;

**int** radius;

};

constexpr Circle func(**int** x)

{

**return** Circle(x, 4, 5);

}

**int** main()

{

    //val将在编译期计算为15，multify(3, 5)是常量表达式，val是常量表达式

    constexpr **int** val = multify(3, 5);

**int** x = 3, y = 5;

**int** val3=multify(x, y);//正确，传入的参数非（编译器）常量，则constexpr函数视作普通函数

    //constexpr val2 = multify(x, y); //error，x、y不是常量，constexpr相当于普通函数 ，函数表达式不是常量表达式，不能赋值给constexpr val2

    constexpr **int** q = factorial(5);

    //给constexpr构造函数传入常量，则生成的对象是constexpr，对象中的所有数据成员都会是constexpr

    constexpr Circle c(0, 0, 10);

    constexpr **int** area = c.getArea();

    constexpr Circle c2=func(3);

**int** g = 3;

    //constexpr Circle c3 = func(g); error

    func(g).set\_x(5);

}

在C++11中，对成员函数而言constexpr包含const的含义，然而在之后的C++版本中这点已经改变了。如

#include <cassert>

struct A {

    int i = 0;

    constexpr A() {}

    constexpr int a() { return i; }

    constexpr A&  c() { ++i; return \*this; } *//这里显然不能为const成员函数*

};

int main(){

    static\_assert(A{}.c().a()==1);

}

*//g++ -o main main.cpp -std=c++20*

# 移位：

C++中int与double型数据混合运算，int型隐式转换为double型，结果为double

易出现以下错误：

double x = 20;

double y = 1 / 2 \* x;

此时y的值为0.0，而不是10.0。1/2\*x运算时，先算1/2，两个int型运算结果仍为int，所以值为0。再算(1/2）\*x，int 0被转换为double型0.0。0.0\*20结果为0。想求double型数据20的二分之一，正确的表达式是double z = 1 / 2.0 \* x

向下取整 cout << floor(3.4) << " " << floor(3) << endl; //3 3

向上取整 cout << ceil(3.4) << " " << ceil(3) << endl; // 4 3

C++中整形字面值常量的类型默认为int或unsigned int或long或unsigend long

一个正数整型字面值，其值适合int就是int类型，超出int范围就unsigned int，再超出unsigned int范围则是long，再超出long范围则是unsigned long

一个负数整型字面值，其值适合int就是int类型，超出int范围就是long

字面值0x40000082类型为unsigned int，0x40000082即0100 0000 0000 0000 0000 0000 1000 0010

字面值0x80000082类型为int，0x80000082即1000 0000 0000 0000 0000 0000 1000 0010

字面值-2147483388类型为int，-2147483388即10000000 00000000 00000001 00000100

unsigned int右移，最高位补0，按unsigned int翻译移后结果，结果为unsigned int类型

unsigned int左移，最低位补0，按unsigned int翻译移后结果，结果为unsigned int类型。因此左移1位后可能值变小

int右移，最高位补符号位，结果为int类型。具体地，右移，补符号位，按int翻译移后结果，返回int类型

int左移，最低位补0，结果为int类型。因此正数可能变为负数，负数可能变为正数。具体地，左移，最低位补0，按int翻译移后结果，返回int类型。

无论左移右移，移位的结果总是与使用\*、/运算符的结果一致。移位溢出则\*、/也溢出，移位结果“奇怪”则\*、/也“奇怪”。

左移、右移都是所有位一起移动（包括有符号数的符号位，一视同仁），然后再根据有无符号、左右移决定是否补符号位

byte、short和char类型移位，将操作数转为int类型，再移位，移位的结果是int类型

unsigned char x = 0x82; //1000 0010

cout << (int)x << endl; //130

cout << typeid(x << 1).name() << endl;//int类型

cout << (int)(x << 1) << endl; //260

cout << typeid(x >> 1).name() << endl;//对unsigned char移位的结果是int类型

cout << (int)(x >> 1) << endl; //65

unsigned int y = 0x40000082; //0100 0000 0000 0000 0000 0000 1000 0010

cout << y << endl; //4\*16的7次方+8\*16+2=1073741954

cout << typeid(y >> 1).name() << endl;//对unsigned int移位的结果是unsigend int类型

cout << (y >> 1) << endl; //536870977

cout << typeid(y << 1).name() << endl;//unsigend int类型

cout << (y << 1) << endl; //2147483908

//对unsigned int的左移、右移，直接移动，按unsigned int读移后的值，因此左移可能会出界，反倒值变小

unsigned int z = 0x80000082; //0x80000082是unigned int类型 //1000 0000 0000 0000 0000 0000 1000 0010

cout << z << endl; //8\*16的7次方+8\*16+2=2147483778

cout << typeid(z >> 1).name() << endl;//unsigned int

cout << (z >> 1) << endl; //1073741889

cout << typeid(z << 1).name() << endl;//unsigend int

cout << (z << 1) << endl; //260 //出界

cout << 0x80000082 \* 2 << endl; //260

//对int左移、右移，直接移动，按int读移后的值（补码），因此左移可能会整数变负数，负数变整数

int t = 0x40000082; //int类型 //0100 0000 0000 0000 0000 0000 1000 0010

cout << t << endl; //4\*16的7次方+8\*16+2=1073741954

cout << typeid(t >> 1).name() << endl;//int

cout << (t >> 1) << endl; //536870977

cout << typeid(t << 1).name() << endl;//int

cout << (t << 1) << endl; //-2147483388 //10000000 00000000 00000001 00000100

cout << 0x40000082 \* 2 << endl; //-2147483388

int p = -2147483388; //int类型 //10000000 00000000 00000001 00000100

cout << p << endl;

cout << typeid(p >> 1).name() << endl;//int

cout << (p >> 1) << endl; // 11000000 00000000 00000000 1000 0010

cout << typeid(p << 1).name() << endl;//int

cout << (p << 1) << endl; //00000000 00000000 00000010 00001000，即520

cout << -2147483388 \* 2 << endl; //520

# 位运算的应用：

对于除法,我们用>>运算符实现,对于数N，N>>m的结果就是N/(2^m)

对于取模,我们用N=N&((1<<K)-1),这是对2^K取模

N|(1<<m),就可以在N的第m位上置1了

将int型变量a的第k位清0，即a=a&~(1<<k)

加法可用异或和与运算实现，a+b等于(a^b)+(a&b)<<1

void assert( int expression );

assert(expr); //如果expr为false，则

assert宏的原型定义在<cassert >中，其作用是如果它的参数expression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。请看

在调试结束后，可以通过在包含#include <cassert>的语句之前插入 #define NDEBUG 来禁用assert调用

Release模式默认全局定义#define NDEBUG

exit(1);/\*不正常退出程序\*/

“0表示正常退出”。其他的status值以及含义都是由程序员实现自行规定的（implementation-defined)。父进程中可以调用BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode) 得到exit的参数，如果没有这样一个父进程来处理exit()传递的参数，则写几都没什么关系！

将这个文件保存为a.cpp，编译运行后会生成一个a.exe文件。现在打开命令提示符，在命令行里运行刚才编译好的可执行文件，然后输入 echo %ERRORLEVEL% ，回车，就可以看到程序返回 一个0 。如果把 return 0; 改为 return 99; ，那么很显然，再次执行上述步骤以后你可以看到程序返回99

cout对应于标准输出流，默认情况下是显示器。这是一个被缓冲的输出，可以被重定向。

cerr对应标准错误流，用于显示错误消息。默认情况下被关联到标准输出流，但它不被缓冲，也就说错误消息可以直接发送到显示器，而无需等到缓冲区或者新的换行符时，才被显示

# C++中的宏：

## 宏基础知识：

如果我们在一行代码的行尾放置一个反斜杠，c语言编译器会忽略行尾的换行符，而把下一行的内容也算作是本行的内容。这里反斜杠起到了续行的作用

字符串常量跨行书写， 如下所示：

      char letters[] = {"abcdefghijklmnopqrstuvwxyz\

ABCDEFGHIJKLMNOPQRSTUVWXYZ"};

从续行的开始输入字符串，可以避免在整个字符串中加入多于的空格。综上所述，上面的语句定义了一个字符数组letters，并将其初始化为如下的初值："abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"

c语言中还有一种拆分字符串的方法，那就是将其写个多个相邻的字符串。这些字符串之间用0个或者多个空白、制作符以及换行符隔开。c语言编译器会自动将这些字符串连接起来。因此，下面的表达式："one" "two" "three" 实际上相当于 "onetwothree".

因此前面跨行的初始化语句也可以用下面的形式完成：

char letters[] = {"abcdefghijklmnopqrstuvwxyz"

"ABCDEFGHIJKLMNOPQRSTUVWXYZ"}

std::cout<<("1" " " "==" " " "2")<<std::endl; //1 == 2

宏定义有**无参数宏定义**和**带参数宏定义**两种

**无参数的宏定义的一般形式为**

**#define 宏名 字符序列**

要求宏名与字符序列之间用空格符分隔

这种宏定义要求编译预处理程序将源程序中随后所有的宏名 (字符串常量中的除外)均用字符序列替换之

宏定义在源程序中单独另起一行，**换行符是宏定义的结束标志**。如果一个宏定义太长，一行不够时，**可采用续行的方法。续行是在键人回车符之前先键入字符’’’’’’’\’，注意回车要紧接在符号’’\’之后，中间不能插入其它符号。**

宏定义的有效范围**从宏定义的定义结束处开始到其所在的源程序文件末尾，因此**

在新的宏定义中，可以使用前面已定义的宏名。例如，

             # define R 2.5

             # define PI 3.1415926

             # define Circle 2\*PI\*R

             # define Area PI\* R \* R

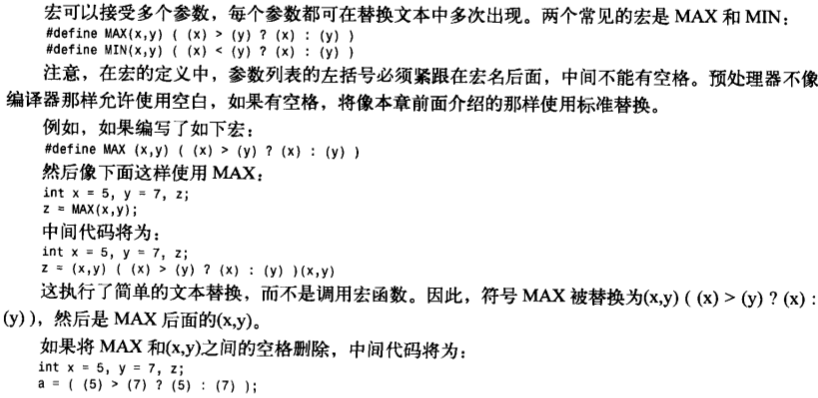
程序中的Circle被展开为2\*3.1415926\* 2.5， Area被展开为3.1415926\*2.5\*2.5。

     如有必要，宏名可被重复定义。被重复定义后，宏名原先的意义被新意义所代替

**带参数宏定义进一步扩充了无参数宏定义的能力，在字符序列替换同时还能进行参数替换。带参数定定义的一般形式为**

**# define 标识符(参数表) 字符序列**

**其中参数表中的参数之间用逗号分隔，字符序列中应包含参数表中的参数。**在定义带参数的宏时，宏名标识符与左圆括号之间不允许有空白符，应紧接在一起，否则变成了无参数的宏定义。



使用含参数宏时，字符序列中出现的每个参数都最好用小括号()括起来，避免意外

#define MIN(a,b) ((a)<=(b)?(a):(b))

std::cout << "less is:" << (10<=9?10:9) << std::endl; //入如果是std::cout << "less is:" <<10<=9?10:9<< std::endl;则报错

std::cout << "less is:" << MIN(18,23)<< std::endl; //因此，#define MIN(a,b) (a<=b?a:b)中必须有字符序列最外边的括号

std::cout << "less is:" << MIN(20<=9, 30>=8) << std::endl; //输出0，正确。

//如果是#define MIN(a,b) (a<=b?a:b)，则输出为1，与要求不符，因为宏被展开为(20<=9<=30>=8?20<=9:30>=8)

函数调用占用程序运行时间，宏调用只占编译时间

#把宏参数变为一个字符串

##把两个宏参数贴合在一起

#define STR(s) #s

#define CONS(a,b) int(a##e##b)

printf(STR(vck)); // 输出字符串"vck"

printf("%d\n", CONS(2,3)); // 2e3 输出:2000

将任意数据类型转换为字符串常量的宏

#define STR(x) #x

std::string self = std::string("my age is ") + std::string(STR(21));

//被展开为std::string(“21”)，而不是std::string(21)

#define STR(s) #s

printf("int max: %s\n", STR(INT\_MAX)); // INT\_MAX #include<climits>

这行会被展开为：

printf("int max: %s\n", "INT\_MAX");

#define INT\_MM 34

printf("int max: %s\n", "INT\_MM"); //打印字符串“INT\_MM”

#define A (2)

#define CONS(a,b) int(a##e##b)

printf("%s\n", CONS(A, A)); // compile error

宏替换后是：

printf("%s\n", int(AeA));

#define x265\_encoder\_glue1(x, y) x ## y 这里x与#、#与y之间有间隔

x265\_encoder\_glue1(a,b); 宏替换后，为ab

## C语言中嵌套宏的展开规律：

1. 一般宏调用的展开规律是，先展开内层宏参数，再展开外层宏函数，所以展开顺序是由内而外。
2. 如果宏定义中某个形参前面有#运算符，则调用此宏定义时，不展开该形参对应的实参，而是直接把这个实参变为字符串。
3. 如果宏定义中某个形参前面有#@运算符，则调用此宏定义时，不展开该形参对应的实参，而是直接把这个实参变为字符。
4. 如果宏定义中某个形参前面或者后面有##运算符，则调用此宏定义时，不展开该形参对应的实参，而是将##运算符前后的实参连接到一起形成一个新的符号，注意不是字符串

#**include**<stdio.h>

#**include**<iostream>

**using** **namespace** std;

#**define** A(x) T\_##x

#**define** B(x) #@x

#**define** C(x) #x

#**define** f(a,b) a##b

#**define** g(a) #a

#**define** h(a) g(a)

#**define** ff(x,y) x+y

#**define** foo abc

#**define** foo2(x) x##123

#**define** foo123 "hehe"

#**define** Str1 "hello"

#**define** Cat2Str(x,y) x#y

**void** **main**()

{

printf("%c\n", B(1)); // '1'

printf("%s\n", C(1)); // "1"

printf("%s\n", h(A(1)));

//输出：T\_1

//宏展开过程为：因h(a)的宏形参前没有#或#@，形参前后没有##，因此先展开宏参数A(1)。

//h(A(1))→h(T\_1)→g(T\_1)→"T\_1"

printf("%s\n", h(f(1, 2)));

//输出:12

//宏展开过程为：因宏h(a)的宏形参前没有#或#@，形参前后没有##，因此先展开宏参数。

//h(f(1, 2))→h(12)→g(12)→"12"

printf("%s\n", g(f(1, 2)));

//输出:f(1,2)

//宏展开过程为：因宏g(a)宏定义的形参前有#，则不展开实参f(1,2)，而是把实参作为字符串。

//g(f(1, 2))→#f(1,2)→"f(1,2)"

printf("%d\n", f(f, f(3, 4)));

//输出：7。

//宏展开顺序为：因宏定义f(a,b)的形参前后有##，因此，不展开实参， f(f, f(3, 4))→ff(3,4)→3+4

printf("%s\n", g(1)g(2)); //输出：12

//这个例子很特殊，我原以为编译器不会识别g(1)g(2),事实上，编译后会识别连在一起的宏，然后分别展开。

//等价于printf("%s\n", "1""2");

printf("%s\n", f(g(1), g(2))); //输出：12

//宏展开过程为：因宏定义f(a,b)的形参前后有##，所以不展开实参g(1)和g(2)，f(g(1), g(2))展开后变为g(1)g(2)，

//f(g(1), g(2))→g(1)##g(2)→g(1)g(2)→“12”

printf("%s\n", Cat2Str(Str1, ",nihao")); //输出：hello",nihao"

printf("%s\n", Cat2Str(Str1, Str1)); //输出：helloStr1

//Cat2Str(Str1, Str1)宏展开过程为：

//因为Cat2Str(x,y)的y参数前有#，直接将后面的参数转为字符串

//Cat2Str(Str1, Str1)→"hello""Str1"→"helloStr1"

printf("%s\n", foo2(foo));

//宏展开过程为：因为foo2(x)的宏定义形参前有##，因此，foo2(foo)对应的实参foo不再展开。

//foo2(foo)→foo##123→foo123→"hehe"

}

## C++编译阶段打印宏定义的内容：

#**include** <iostream>

#**pragma** message("Hello World!")

#**define** PRINT\_MACRO\_HELPER(x) #x

#**define** PRINT\_MACRO(x) #x "=" PRINT\_MACRO\_HELPER(x)

#**define** NUMBER\_MACRO 3.14159

#**define** STRING\_MACRO "This is a string."

#**pragma** message(PRINT\_MACRO(NUMBER\_MACRO))

#**pragma** message(PRINT\_MACRO(STRING\_MACRO))

#**pragma** message(PRINT\_MACRO(UNDEF\_MACRO))

**int** **main**()

{

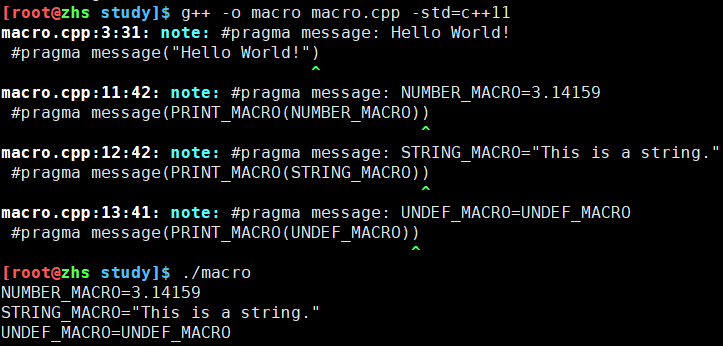
std::cout<<PRINT\_MACRO(NUMBER\_MACRO)<<std::endl;

std::cout<<PRINT\_MACRO(STRING\_MACRO)<<std::endl;

std::cout<<PRINT\_MACRO(UNDEF\_MACRO)<<std::endl;

}

//运行结果：



## 自引用宏：

所谓的自引用宏(Self-Referential Macros)是指在定义中包括自身名字的宏，如下所示：

#define CustomAdd(a,b) 10 + CustomAdd(a,b)

一般认为宏展开是一个替换的过程，考虑对上述宏的调用则是一个无限递归的展开过程：

10 + (10 + (10 + ...))

然而事实是，对CustomAdd(99, 100)的调用产生的结果是：

10 + CustomAdd(99, 100)

为了防止无限递归这种情况的产生，宏展开时会做一个标记，再次展开时就不再对其进行替换，这种情况同样适用于非直接自引用的情况，如

#define x (4 + y)

#define y (2 \* x)

x和y的展开结果如下：

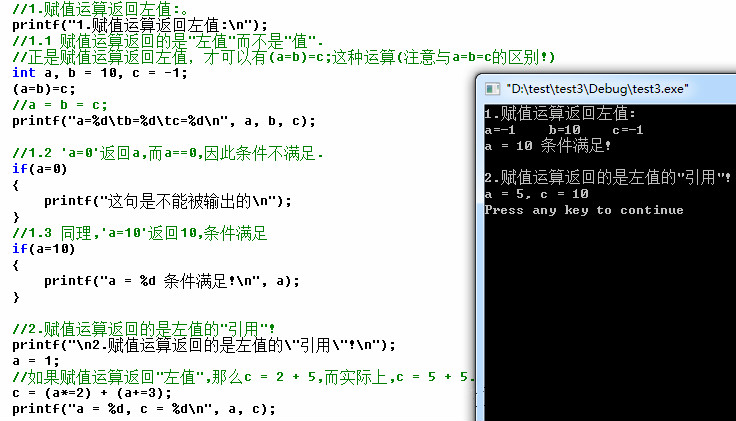
x → (4 + y)

→ (4 + (2 \* x))

y → (2 \* x)

→ (2 \* (4 + y))

# C++赋值表达式的返回值：



int j, m, n;

for (j = 0; (m = 0) && (n = 8);) { cout << "no enter" << endl; break; } //未进入循环

for (j = 0; m = 8, n = 8;) { cout << "enter" << endl; break; } //进入循环

逗号运算符:将相关的表达式链接在一起。由于逗号链接表达式从左到右进行计算，因此最右边的表达式的值即为该组合表达式的值

value = (x = 10, y = 5, x + y); //value = 15;

bool bl=(m = 0&&(n = 8)); //bl=false

bool bl=(m = 0, n = 8); //bl=true

# 结构体对齐：

## 结构体对齐规则

C++标准要求：

* 非 union 的标准布局类中首个成员前无偏移，成员偏移从前往后递增
* 数据成员的偏移都必须使得其对齐得到满足
* 数组类型的对齐同其元素类型

可以根据标准推得：

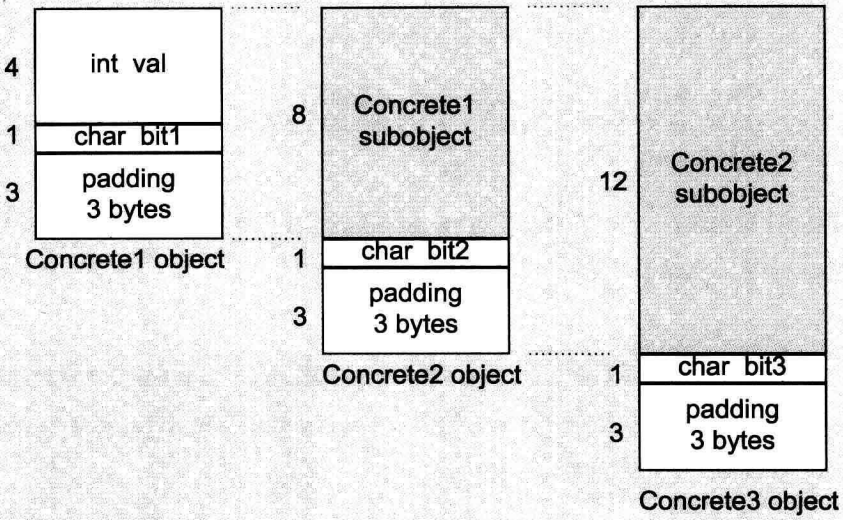
* 类的大小是对齐的整数倍
* 类的对齐不小于各个成员对齐的最大值

主流实现有额外限定：

* 标准布局类中，确定每个成员的偏移时都会使偏移可能小
* 类的对齐等于各个成员对齐的最大值
* 类的尾部填充字节数也要尽可能小
* 非标准的 #pragma pack (N) 使得确定类的布局时只要遇到大于 N 的对齐都当成 N 处理，这可能减少类和数据成员的原生对齐和填充字节数。

以使用LP64数据模型为例，short 大小与对齐均为 2 字节， int 大小与对齐均为 4 字节， long 及指针大小与对齐均为 8 字节。

* struct P1{int a;char b;int c;char d;};a偏移为0，b偏移为4，c偏移为8（大于4+1的4的最小整数倍），d偏移为12。P1对齐为4，大小为16。
* struct P2{int a;char b;char c;long d;};a偏移为0，b偏移为4，c偏移为5，d偏移为8。P2对齐为8，大小为16。
* struct P3{short a[3];char b[3];};a偏移为0，b偏移为6。P3对齐为2，大小为10。
* struct P4{short a[3];char\* b[3];};a偏移为0，b偏移为8（大于0+6的8的最小整数倍）。P4对齐为8，大小为32。
* struct P5{struct P3 a[2];struct P2 t;};a偏移为0，t偏移为24（大于0+20的8的最小整数倍）。P5对齐为8，大小为40。



## 预编译指令#pragma pack

#pragma pack(push) //保存当前对齐状态

#pragma pack(4)//设定为4字节对齐

#pragma pack () 作用：取消自定义字节对齐方式

#pragma pack (push,1) 作用：是指把原来对齐方式设置压栈，并设新的对齐方式设置为1个字节对齐

#pragma pack(pop) 作用：恢复对齐状态

预编译命令#pragma pack(n)，n=1,2,4,8,16， 规则：

不使用#pragma pack(n)时结构体有默认对齐值、各成员有默认对齐值，使用#pragma pack(n)后，结构体的对齐值是min{结构体默认值, n}、结构体成员的对齐值是min{成员默认值对齐值, n}，然后结构体的对齐值和成员的对齐值即可决定内存布局

* + - 结构体作为一个整体自身要对齐，按照#pragma pack指定的数值和最大数据成员长度中，比较小的那个进行，记为m，即首地址位于的m的整数倍
    - 各数据成员要对齐，第一个数据成员放在offset为0的地方，以后每个数据成员的对齐按照#pragma pack指定的数值和这个数据成员自身长度中，比较小的那个进行。
    - 总大小必须是#pragma pack指定的数值和最大数据成员长度中比较小的那个值的整数倍

#pragma pack(1)

**struct** T1{

**char** a;

**double** b;

};   //这里T1的size为9，不能用struct alignas(1) T1代替

#pragma pack()    //如果不使用pragma pack()，则#pragma pack(1)的影响传递到下一个结构体T2

**struct** T2{

**char** a;

**double** b;

};

#pragma pack(push,4)

struct E //alignof为4，sizeof为16

{

char a;

char b;

double c;

char d;

};

#pragma pack(pop) /\*恢复之前的对齐状态\*/

//cout << offsetof(E, b) << " " << offsetof(E, c) << " " << offsetof(E, d) << endl; //1 4 12

结构体E的成员的最大对齐值为8，指定的对齐值为4，因此结构体自身对齐值alignof为4，首地址为4的整数倍

## alignas

联合体，它的所有成员相对于基地址的偏移量都为0，此结构空间要大到足够容纳最“宽”的成员，并且，其对齐方式要适合于联合中所有类型的成员。

union DATE { char a; int i[5]; double b;};

cout <<alignof(DATE) << endl; //8

cout << sizeof(DATE) << endl; //24

该结构要放得下int i[5]必须要至少占4×5=20个字节。如果没有double的话20个字节够用了，此时按4字节对齐。但是加入了double就必须考虑double的对齐方式，double是按照8字节对齐的，所以必须添加4个字节使其满足8×3=24

struct S1{ uint16\_t a; //2 uint64\_t b; //8 uint32\_t c; //4};

cout << sizeof(S1) << endl; //24

struct S2{uint64\_t a; //8 uint16\_t b; //2 uint32\_t c; //4};

cout << sizeof(S2) << endl; //16

* 结构体用alignas(n)，n必须是2的幂，且n必须大于等于结构体中所有元素的对齐值。
* alignas(n) ，则结构体的对齐值为n，那么结构体变量的首地址必须是n的整数倍、总大小sizeof(A)必须是n的整数倍
* 结构体用alignas(n)，n不影响数据成员的对齐
* 内置数据类型默认对齐值与其sizeof相同：如MSVC x64上alignof(char):1、alignof(short):2、alignof(int):4、alignof(float):4、alignof(double):8
* struct或者class或者union的默认对齐值：其成员中自身对齐值最大的那个值
* alignof后面只能跟类型名，不能跟具体的变量名

struct alignas(2) A{ char a; char b; char c; char d;};

alignas修饰结构体时，不是对齐结构体每个成员，而是对齐结构体本身，所以大小不是4\*2=8，而是4不变。如果再增加一个char成员，则大小6， 不是5，因为A对齐了，A+sizeof(A)是下一个A元素的地址，需要同样满足对齐，因此需要补空间

如果alignas()设置的非常大时：struct alignas(32) BB{ int a; alignas(8) char b;};

cout << &bb << endl; //000000AB4718F620

cout << &bb.a << endl; //000000AB4718F620

cout << (void\*)&bb.b << endl;//000000AB4718F628

struct alignas(2) A{ char a; int b; short c;};sizeof(A)不是8，而是12，警告："A"对齐说明符小于实际对齐方式(4)，将被忽略

**struct** X

{

**char** a;

**char** b;

};

**struct** Y

{

    alignas(4) **char** a;

**char** b;

};

**struct** Z

{

    alignas(4) **char** a;

    alignas(4) **char** b;

};

cout << offsetof(X, a) << " " << offsetof(X, b) << endl;  //0 1

cout << offsetof(Y, a) << " " << offsetof(Y, b) << endl;  //0 1

cout << offsetof(Z, a) << " " << offsetof(Z, b) << endl;  //0 4

cout << **sizeof**(X) << endl;  //2

cout << **sizeof**(Y) << endl;  //4

cout << **sizeof**(Z) << endl;  //8

## #pragma pack与align的区别

* #pragma pack只能将结构体自身的对齐值设置的更小（小于等于其数据成员的最大对齐值）

alignas()只能将结构体的自身对齐值设置的更大（大于等于其数据成员的最大对齐值）

* #pragma pack不仅影响结构体自身的对齐，还影响数据成员的对齐

align仅影响结构体自身的对齐，不影响数据成员的对齐值

如果是#pragma pack(2) class B{char a; char b;double c;};则alignof为2，sizeof为10

如果是class alignas(8) B{char a;int b;}; 则alignof为8，sizeof为8

如果是#pragma pack(8) class B{ char a; int b;}; 则alignof为4，sizeof为8

## \_\_declspec(align(n))

struct A {

\_\_declspec(align(16)) int16\_t coeff[3][3][8];

};

struct B {

int i;

\_\_declspec(align(16)) int16\_t coeff[3][3][8];

};

struct C {

\_\_declspec(align(16)) char coeff[3];

};

struct D {

int i;

\_\_declspec(align(16)) char coeff[3];

};

int main()

{

cout << sizeof(A) << endl; //144=3\*3\*8\*2

cout << sizeof(B) << endl; //160

cout << sizeof(C) << endl; //16

cout << sizeof(D) << endl; //32

}

\_\_declspec(align(16))做为数组定义的修释符，这表示该数组是以16字节为边界对齐的

# 数据结构：

构建二叉树：

**char** data[] = { 'A','B','C','D','E','G', '#','#','#','F','#','#','H' };

**void** createTree(Node\*& root, **char** data[], **int** index, **int** len) //以前序遍历的方法构建，len为data[]数组的元素个数

{

**if** (index - 1 >= len || data[index - 1] == '#')

    {

        root = nullptr;

**return**;

    }

    Node\* p = **new** Node;

    p->data = data[index - 1];

    root = p;

    createTree(p->lchild, data, index \* 2, len);

    createTree(p->rchild, data, index \* 2 + 1, len);

}

若有序表的关键字序列为（b,c,d,e,f,g,q,r,s,t），则在二分查找关键字b的过程中，先后进行的关键字依次为：f,c,b

二分查找有一个+1， -1 的操作，如果mid 比 tar 小 就往后找 +1， 反之 往前找-1

二叉树前序遍历：根左右

中序遍历：左根右

后序遍历：左右根

找二叉排序树的双亲结点：

//当tar不是二叉排序树的节点或tar为根节点时，返回nullptr，同时 isLeftChild记录tar是其父节点的左子树还是右子树

**template**<**class** T>

Node<T>\* findParent(Node<T>\* tar, Node<T>\* root,**bool**& isLeftChild)

{

**if** (tar == nullptr) **return** nullptr;

    Node<T>\* p = root;

    Node<T>\* par = nullptr;

**while** (1)

    {

**if** (p == nullptr) **return** nullptr;

**if** (p == tar) **return** par;

**if** (tar->data <= p->data)

        {

            par = p;

            p = p->left;

            isLeftChild = **true**;

        }

**else**

        {

            par = p;

            p = p->right;

            isLeftChild = **false**;

        }

    }

}

中序遍历是递增序列 等价于 二叉排序树

删除二叉排序树的某个结点：

* 删除的是叶子结点：直接删除
* 删除的结点只有左子树或只有右子树：让子树连接在父结点上
* 删除的结点有左子树、右子树：在左子树上找前驱结点，让待删除结点的值为前驱结点的值，删除前驱结点（前驱结点要么为叶结点、要么仅有一棵左子树树）

析构二叉树：采用后序遍历的方式，先析构左子树，再析构右子树，再delete根结点

排序二叉树，找某个结点的前驱结点：该结点的右子树，中序遍历的最后一个元素

找排序二叉树某个结点的前驱结点和前驱结点的父结点：

Node\* pre = tar->lchild; //结点tar的前驱结点

Node\* parent = tar; //结点tar前驱结点的父结点

while (pre->rchild != nullptr)

{

parent = pre;

pre = pre->rchild;

}

文件系统和数据库系统中常用的B/B+ 树

归并排序：

(1)for (int seg = 1; seg < len; seg = i \* 2) mergeOnce(int data[],int len,seg)

(2)mergeOnce(int data[],int len,seg)函数：

计算有多少对长为seg的数组段可以合并，int pair\_num=len/(2\*seg)，

计算剩余的元素数，int rest\_len=len%(2\*seg)

合并剩下的rest\_len个元素：

* 如果0<rest\_len<=seg，直接与前边的合并，merge(int data[], int start, int mid, int end)//[start,mid],[mid+1,end]
* 如果rest\_len>seg，则先合并seg与rest\_len-seg两段，再与前边的合并

(3)merge(int data[], int start, int mid, int end)函数//[start,mid],[mid+1,end]

while (1)

{

while (right >= right\_threshold && copy[right]>copy[left]) data[tar--] = copy[right--];

如果right<right\_threshold，跳出while(1)

while (left >= left\_threshold && copy[left] >= copy[right]) data[tar--] = copy[left--];

如果left<left\_threshold，跳出while(1)

}

if (right<right\_threshold) while (left >= left\_threshold) data[tar--] = copy[left--];

if (left<left\_threshold) while (right >= right\_threshold) data[tar--] = copy[right--];

快速排序：

* 任选一个元素作为基准，一趟排序，使基准左边的元素都比基准小、基准右边的元素都比基准大
* 左区间作为新数组，选基准排序…… 右区间作为新数组，选基准排序……
* 直到新数组只有一个元素

快排每一次确定基准元素位置的方法：挖坑填数

以一个数组作为示例，取区间第一个数为基准数

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 72 | 6 | 57 | 88 | 60 | 42 | 83 | 73 | 48 | 85 |

初始时，i = 0;  j = 9;   X = a[i] = 72

由于已经将a[0]中的数保存到X中，可以理解成在数组a[0]上挖了个坑，等待选取一个合适的其它数据填充到这来。

从j开始向前找一个比X小或等于X的数。当j=8，符合条件，将a[8]挖出再填到上一个坑a[0]中。a[0]=a[8]; i++;  这样坑a[0]被填充，但又形成了一个新坑a[8]。再找数字来填a[8]这个坑。这次从i开始向后找一个大于X的数，当i=3，符合条件，将a[3]挖出再填到上一个坑中a[8]=a[3]; j--

void quickSort(int data[], int start, int end)

{

if (start >= end) return;

int basepos = calBasePos(data, start, end);

quickSort(data, start, basepos - 1);

quickSort(data, basepos + 1, end);

}

int calBasePos(data, start, end)函数选取最左边元素data[start]作为基准，挖坑填充，使基准左边的元素都比基准小、基准右边的元素都比基准大，并返回最后基准元素的正确位置

auto calBasePos = [](**int** data[], **int** start, **int** end)->**int**

{

**if** (start == end) **return** start;

**int** base = data[start]; //挖坑填数，选取区间最左边的元素最为基准，先挖出来产生一个坑

**int** left = start;

**int** right = end;

**while** (1)

    {

        //在坑的右边寻找第一个比基准小的元素

**while** (right > left && data[right] >= base ) right--;//等待填的坑是data[left]，需要一个小于等于基准的数

**if** (right == left) **break**;

**else** data[left++] = data[right];

        //产生一个新坑，data[right]

**while** (right > left && data[left] <= base ) left++; //在坑的左边寻找第一个比基准大的元素

**if** (right == left) **break**;

**else** data[right--] = data[left];

    }

    data[left] = base; //此时left=right

**return** left; //基准的位置

};

希尔排序：先使元素基本有序，再直接插入排序

使元素基本有序的方法：以10(即len/2)为间隔的元素作为一组，各组直接插入排序；以5（即10/2）为间隔的元素作为一组，各组直接插入排序；

……，以1为间隔的元素作为一组，即对整个数组进行一次直接插入排序；

int i, j, gap;

for (gap = len / 2; gap >= 1; gap = gap / 2)

{

for (i = 0; i < gap; i++)

{

for (j = i + gap; j < len; j += gap) //间隔为gap，首元素为data[i]的一个分组，对该分组插入排序

{

插入排序

}

}

}

基于比较的排序算法时间复杂度最小是O(nlogn)的

计数排序是一种非基于比较的排序算法，只能对整数排序，其空间复杂度和时间复杂度均为O(n+k)，其中k是整数的范围，即max-min+1。

计数排序，有max-min+1个坑

桶排序，max-min为元素范围

桶排序，计算元素所属的桶时，必须是double(max-min)除以桶个数得到桶尺寸doube型，再int pos=（x-min）/桶尺寸；同时注意对最大元素的处理

if (max == min) return 0; //桶排序会用max-min作为除数，因此不能为0

int bucketIndex = (data[i] - min) / ((double)(max - min)

if (bucketIndex == bucketNum) bucketIndex = bucketNum - 1; //前边的是左闭右开区间，最后一个是左闭右闭区间，避免数组中最大数落在桶外

实际上，桶排序的f(k)值的计算，其作用就相当于快排中划分，已经把大量数据分割成了基本有序的数据块(桶)。然后只需要对桶中的少量数据做先进的比较排序即可（如使用O(logn)复杂度快排）

(1) 循环计算每个关键字的桶映射函数，这个时间复杂度是O(N)。

(2) 利用先进的比较排序算法对每个桶内的所有数据进行先进的比较排序，其时间复杂度为 ∑ O(Ni\*logNi) 。其中Ni 为第i个桶的数据量

桶排序的平均时间复杂度为线性的O(N+N\*(logN-logM))。如果相对于同样的N，桶数量M越大，其效率越高，最好的时间复杂度达到O(N)。 当然桶排序的空间复杂度 为O(N+M)

普通二叉树，一般以数组形式完全确定了一棵树的拓扑，要求转换为指针形式的二叉树

二叉排序树，为了排序，给定的只是待排序的数组，并不是树的拓扑，因此逐个增加元素，构成一棵排序树

堆排序，将待排序数组视作一棵完全二叉树，总共有len/2个元素需要调整，由后往前调整

外部排序：

实际上归并排序需要额外的存储空间O(n)

例如要对外存中4500个记录进行归并，而内存大小只能容纳750个记录，在第一阶段，我们可以每次读取750个记录进行排序，这样可以分六次读取，进行排序，可以得到六个有序的归并段，如下图：

https://images2017.cnblogs.com/blog/1172457/201708/1172457-20170824204107496-53968655.png

每个归并段的大小是750个记录，记住，这些归并段已经全部写到临时缓冲区（由一个可用的磁盘充当）内了，这是第一步的排序结果。

完成第二步该怎么做呢？这时候归并算法就有用处了，算法描述如下：

1、将内存空间划分为三份，每份大小250个记录，其中两个用作输入缓冲区，另外一个用作输出缓冲区。首先对Segment\_1和Segment\_2进行归并，先从每个归并段中读取250个记录到输入缓冲区，对其归并，归并结果放到输出缓冲区，当输出缓冲区满后，将其写道临时缓冲区内，如果某个输入缓冲区空了，则从相应的归并段中再读取250个记录进行继续归并，反复以上步骤，直至Segment\_1和Segment\_2全都排好序，形成一个大小为1500的记录，然后对Segment\_3和Segment\_4、Segment\_5和Segment\_6进行同样的操作。

2、对归并好的大小为1500的记录进行如同步骤1一样的操作，进行继续排序，直至最后形成大小为4500的归并段，至此，排序结束。

# size\_t：

size\_t类型是一个类型定义，通常将一些无符号的整形定义为size\_t，比如说unsigned int或者unsigned long，甚至unsigned long long。每一个标准C实现应该选择足够大的无符号整形来代表该平台上最大可能出现的对象大小，即size\_t。

size\_t唯一正确的用途就是“代表某块内存的大小”等类似的用途

void \*malloc(size\_t n);

size\_t是sizeof关键字运算结果的类型。所以，应当通过适当的方式声明n来完成赋值：size\_t n = sizeof(object);

# 模板、泛型：

## 模板求和函数

**template**<**class** T1,**class** T2>

auto sum1(T1 a, T2 b)

{

**return** a + b;

}

或：

**template**<**class** T1, **class** T2>

auto sum3(T1 a, T2 b)->decltype(a+b)  //追踪返回类型，

{

**return** a + b;

}

如果是“decltype(tl + t2) Sum(Tl tl, T2 t2)”，似乎更易读，但编译器从左往右读入符号，遇到t1、t2出错，变量应该先声明再使用，因此用追踪返回类型auto sum3(T1 a, T2 b)->decltype(a+b) 则不会出错

## auto类型推导：

**int** x = 78;

**int**\* y = &x;

**double** foo();  //函数声明

**int**& bar();  //函数声明

auto\* a = &x; //int\*

auto f = &x;  //int\*

auto c = y;   //int\*

auto& b = x;  //int&

auto g = x;   //int

auto\* d = y;  //int\*

**const** **double**& p = foo();//如果double& x = foo() 则报错:非常量左值引用不能绑定右值

**const** auto& q = foo();//如果是auto& x = foo()则报错:非常量左值引用不能绑定右值

auto m = bar(); //int

auto& n = bar(); //int&

**int** v = 100;

**int** &rv = v;

auto arv = rv; //int

auto& brv = rv; //int&

cout << arv++ <<" "; cout << rv << endl; //100 100

cout << brv++ << " "; cout << rv << endl; //100 101

对于指针类型，声明为 auto、auto\* 是没有区别的。

声明引用类型，必须使用 auto &

对于cv限定符const 、volatile：

（1）当不声明为指针或引用时，auto的推导结果和初始化表达式抛弃引用和cv限定符后的类型一致；

（2）当声明为指针或引用时，auto的推导结果将保留初始化表达式的cv限定符

**double** foo=43.96;

**float** vbar = 3.14f;

**float**\* bar=&vbar;

**const** auto a = foo;  //const double

**const** auto& b = foo; //const double&

**const** auto\* c = bar; //const float\*

auto d = a; //double

auto& e = a; //const double&

auto f = c; //const float\*

**const** auto& g = c;//const float\*&,是const引用，如果再g = nullptr则报错

auto& h = c;//float\*&

d = 10.8;  //如果是e = 10.8则报错

\*(**const\_cast**<**float**\*>(c)) = 29.0;//如果是\*c = 29.0则报错

cout << vbar << endl;

h = nullptr;

auto z = new auto(43.96);

cout << typeid(z).name() << endl; //double\*

auto不能作为函数形参类型，如void func(auto x=1){}

auto不能作为类的非静态成员变量类型，如struct str{auto age=23;}

## typeid().name()

typeid()和sizeof()类似，是操作符，而不是函数

type\_info类提供了public虚析构函数，以使用户能够用其作为基类。它的默认构造函数和拷贝构造函数及赋值操作符都定义为private，所以不能定义或复制type\_info类型的对象。程序中创建type\_info对象的唯一方法是使用typeid操作符

typeid()函数返回一个type\_info类的对象的引用

type\_info的name成员函数返回C-style的字符串

用途：

template <typename T>

struct TypeName

{

static const char\* Get() { return typeid(T).name();} /\*typeid(对象).name()返回对象的实际类型，返回类型是字符串char const\* \*/

};

对引用类型，typeid().name()不能区分，如：

int bad = 10;

int& rb = bad;

cout << "typeid(rb).name()" << typeid(rb).name() << endl; //int

追踪返回类型，增加函数定义可读性：

/\*

 函数名为pf，参数列表为空

 返回类型，返回一个函数指针

 该函数的返回类型为int(\*p)()型函数指针，参数列表为double

\*/

**int**(\*(\*pf())(**double**))   ()

{

**return** nullptr;

}

//等价于：

auto pf1()->auto (\*)(**double**)->**int**(\*)()

{

**return** nullptr;

}

**int** main()

{

    cout << (**typeid**(pf).hash\_code() == **typeid**(pf1).hash\_code()) << endl; //1

}

函数引用：int (&p)(double,int)

## traits技术：

Traits并不是 C++ 关键字或一个预先定义好的构件；它们是一种技术，也是一个C++ 程序员共同遵守的协议

#include<type\_traits>

std::iterator\_traits<IterT>::value\_type：IterT类型所指对象的类型，对内置类型和用户定义类型都有效

std::iterator\_traits<int\*>::value\_type x = 10;

用于获取STL容器的元素类型，用decltype(\*iter) value ;替换也可以

template<class C>

void print2nd(const C& container)

{

typename C::const\_iterator iter(container.begin());

++iter;

typename std::iterator\_traits<typename C::const\_iterator>::value\_type value = \*iter;

std::cout << value << std::endl;

}

迭代器共5种：

* 输入迭代器(Input Iterator):只能向前单步迭代元素，只能读，仅能读一次，如istream\_iterator
* 输出迭代器(Output Iterator):只能向前单步迭代元素，只能写，仅能写一次，如ostream\_iterator
* 向前迭代器(Forward Iterator):单步向前迭代，可读可写，可读写多次
* 双向迭代器(Bidirectional Iterator):在向前迭代器的基础上增加了向后单步迭代元素的能力。如被list、map、set使用
* 随机访问迭代器(Random Access Iterator):在双向迭代器的基础上，还可以像指针那样进行算术计算。被std::vector、std::deque、std::string使用

迭代器操作                      说明  
(1)所有迭代器  
p++                              后置自增迭代器  
++p                              前置自增迭代器  
(2)输入迭代器  
\*p                                 复引用迭代器，作为右值  
p=p1                             将一个迭代器赋给另一个迭代器  
p==p1                           比较迭代器的相等性  
p!=p1                            比较迭代器的不等性  
(3)输出迭代器  
\*p                                 复引用迭代器，作为左值  
p=p1                             将一个迭代器赋给另一个迭代器  
(4)正向迭代器  
提供输入输出迭代器的所有功能  
(5)双向迭代器  
--p                                前置自减迭代器  
p--                                后置自减迭代器  
(6)随机迭代器  
p+=i                              将迭代器递增i位  
p-=i                               将迭代器递减i位  
p+i                                在p位加i位后的迭代器  
p-i                                 在p位减i位后的迭代器  
p[i]                                返回p位元素偏离i位的元素引用  
p<p1                             如果迭代器p的位置在p1前，返回true，否则返回false  
p<=p1                           p的位置在p1的前面或同一位置时返回true，否则返回false  
p>p1                             如果迭代器p的位置在p1后，返回true，否则返回false  
p>=p1                           p的位置在p1的后面或同一位置时返回true，否则返回false

向vector v中插入vector v1的第二个和倒数第二个之间的元素

vector<int> v1{ 30,40,50,60,70 };

vector<int> v{3,4,5,6,7};

v.insert(v.begin() + 1, v1.begin() + 1, v1.end()-1); //3 40 50 60 4 5 6 7

结构体iterator\_traits的实现 ：

(1)对用户类型的处理

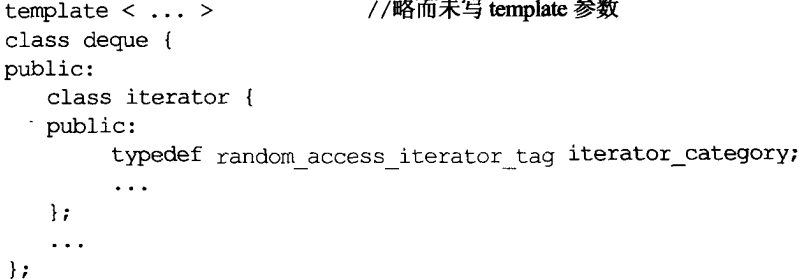
std::iterator\_traits<std::list<int>::iterator> x;

//STL容器如list模板类内嵌一个类iterator，iterator类中对list使用的迭代器（5种迭代器之一）用typedef重命名为x

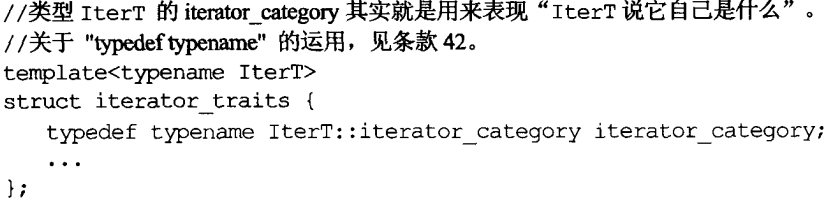
//模板结构体template<Iter> struct iterator\_traits对Iter::x用typedef重命名为x

//因此在struct iterator\_traits中x即为模板类型使用的迭代器

一个针对 deque 迭代器而设计的 class 看起来会是这样子：

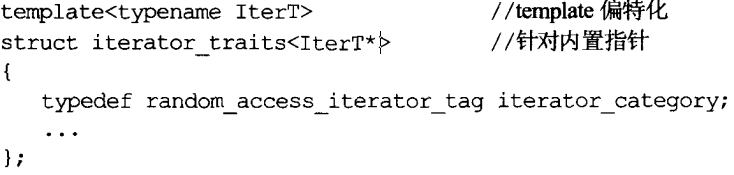


iterator\_traits结构体看起来是这样子：



(2)对指针类型的处理：

为了支持指针迭代器， iterator\_traits 特别针对指针类型提供一个偏特化版本。由于指针的行径与随机访问迭代器类似，所以 iterator\_traits为指针指定的迭代器类型是：



## 模板的类型参数推导：

std::move()，即static\_cast<T&&>将非常量左值转为非常量右值，将常量左值转为常量右值

class A{};

A a; std::move(a);//非右值常量

const A b; const A&& r3 = std::move(b); //常量右值

（1）模板函数的形参形如T&&

template<class T>

void g(T && value) {}

可以传入非常量左值、常量左值，T分别推导为int&、const int&，value分别为int&、const int&

可以传入非常量右值、常量右值，T分别推导为int、const int，value分别为int&&、const int&&

（2）模板函数的形参形如T&

template<class T>

void g(T& value) {}

可以传入常量左值、非常量左值，T分别推导为int、const int，value分别为int&、const int&

不可以传入右值（包括常量右值、非常量右值）

std::remove\_reference<T>::type去掉引用修饰

std::decay<T>::type 退化类型的修饰，如把cosnt int&退化为int

template<class T>

typename std::remove\_reference<T>::type&& move(T&& a)

{

return static\_cast<typename std::remove\_reference<T>::

type&&>(a);

}

//使用时指定模板参数T，仅用于完美转发，因此传入的一定是左值（左值引用或右值引用）

//T被推导为int或int&

template<class T>

typename static\_cast<T&&> forward(typename std::remove\_reference<T>::type& a)

{

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

}

//remove\_refernce为模板类，由左值引用或右值引用得到对应的非引用类型

template<class \_Ty>

struct remove\_reference

{ // remove reference

typedef \_Ty type;

};

template<class \_Ty>

struct remove\_reference<\_Ty&>

{ // remove reference

typedef \_Ty type;

};

template<class \_Ty>

struct remove\_reference<\_Ty&&>

{ // remove rvalue reference

typedef \_Ty type;

};

## std::ref模板函数：

std::ref模板函数的定义为

template<class \_Ty> inline

reference\_wrapper<\_Ty> ref(\_Ty& \_Val) \_NOEXCEPT

{return (reference\_wrapper<\_Ty>(\_Val));}

因此，只能传入的实参类型为左值（包括常量左值、非常量左值）

对形如

template<class T>

void com(T arg)

{

cout << "com=" << &arg << endl;

arg++;

}的模板函数，参数为值传递，如果想起到引用传递的效果，则用std::ref包装要传递的左值

int cnt = 10;

int& rcnt = cnt;

com(cnt); //com=000000590C2FF770

cout << cnt << endl; //10

com(rcnt); //com=000000590C2FF770

cout << "main=" << &rcnt << endl; //main=000000590C2FF794

cout << cnt << endl; //10

com(std::ref(rcnt)); //000000590C2FF770

cout << cnt << endl; //11

对com(std::ref(rcnt))，实际上，传入模板函数com()的是另一个类的对象，也是值传递

标准库中std::ref的实际要考虑各种情况，较复杂。如以下实现一个简易版的myref，用com(myref(rcnt))可以起到com(std::ref(rcnt))的效果

**template**<**class** T>

**class** my\_reference\_wrapper

{

**public**:

    my\_reference\_wrapper(T& val) :ptr\_(&val) {}

    operator T& ()**const**

    {

**return** \*ptr\_;

    }

**private**:

    T\* ptr\_;

};

**template**<**class** T>

my\_reference\_wrapper<T> myref(T& Val)

{ **return** (my\_reference\_wrapper<T>(Val)); }

# 模板特化

模板为什么要特化，因为编译器认为，对于特定的类型，如果你能对某一功能更好的实现，那么就该听你的。

函数模板、类模板都可以全特化

#include<iostream>

using namespace std;

template <typename T>

struct Aa { void f() { cout << "T" << endl; }};

template<>

struct Aa<int> { void f() { cout << "Int" << endl; }};

void main()

{

Aa<float> a;

Aa<int> b;

a.f(); //没有特化，输出 T

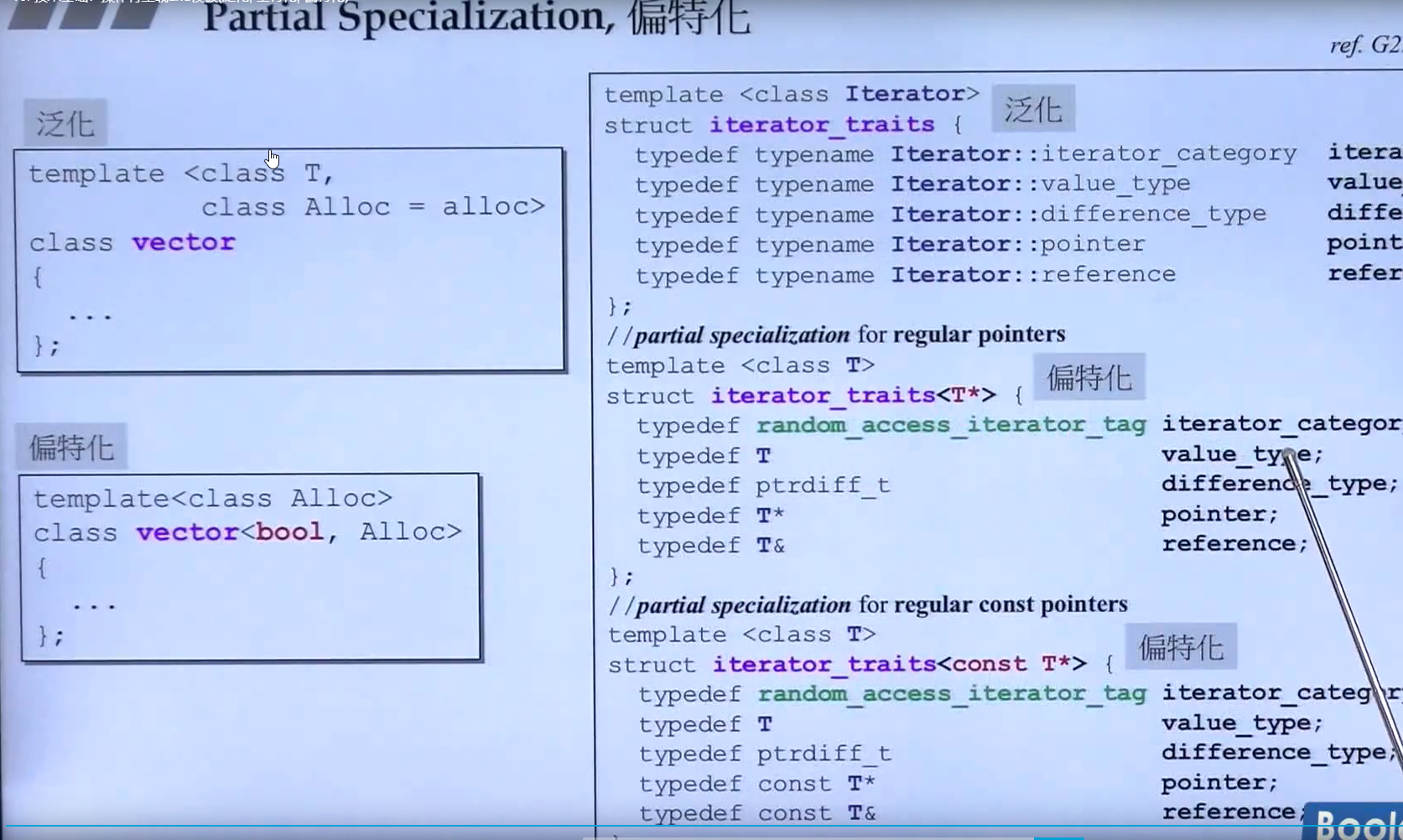
b.f(); //有特化， 输出 Int

}

# 类模板偏特化：

包括模板参数个数的偏特化和范围的偏特化

类模板偏特化不能有默认模板参数



# 模板特化、重载：

类模板不能重载，能全特化、能偏特化

函数模板能重载，能全特化，不能偏特化

“偏特化一个函数模板”的功能，可以用函数模板重载来实现

**template**<**class** T>

**class** B {};

**template**<**class** T>

**inline** **void** myfunc(**const** T& a, **const** T&b)

{

    cout << "my std::myfunc" << endl;

}

//以下程序报错，企图偏特化（partially specialize)一个 function template(std::swap)

**template**<**class** T>

**inline** **void** myfunc<B<T>>(**const** B<T>& a, cosnt B<T>&b)

{

    cout << "my std::myfunc" << endl;

}

//重载函数模板，正确

**template**<**class** T>

**inline** **void** myfunc(**const** B<T>& a, **const** B<T>&b)

{

    cout << "my std::myfunc" << endl;

}

用户可以全特化std命名空间的函数模板（不能重载，函数模板本来就不允许偏特化）

用户可以全特化、偏特化std命名空间的类模板，类模板本来就不能重载

*//以下程序正常，全特化std命名空间的函数模板*

**class** A{};

**namespace** std

{

**template**<>

**inline** **void** swap<A,**void**>(A& a, A&b)

    {

        cout << "my std::swap" << endl;

// 实际上，自C++11后std::swap()已经是移动语义实现，以前是拷贝语义

        A temp(std::move(a));

        a = std::move(b);

        b = std::move(temp);

    }

}

*//以下程序理论上应该报错，std命名空间不能重载*

template<class T>

class B {};

namespace std

{

 template<class T>

 inline void swap (B<T>& a, B<T>&b)

 {

  cout << "my std::swap" << endl;

  B temp(std::move(a));

  a = std::move(b);

  b = std::move(temp);

 }

}

除此之外，不能添加重载到std、不能添加新的 template或class或 function或其他任何东西）到 std里。但VS2015和GCC8.2.0向std中添加函数模板的重载都能通过，如下

class B {};

namespace std

{

    void swap(B& a, B& b)         *//对std中函数模板的重载，添加到std*

    {

        cout << "my std::swap" << endl;

        B temp(std::move(a));

        a = std::move(b);

        b = std::move(temp);

    }

}

int main()

{

    B x;

    B y;

    std::swap(x, y);*//my std::swap*

}

# 函数模板无法偏特化的解决方案

假设想实现一个destroy\_at函数，用来调用对象的析构函数。特别的，如果析构函数是trivial的，那就什么都不做。

直觉上会写出下述代码：

template<typename T, bool value = std::is\_trivially\_destructible\_v<T>>

void destroy\_at(T\* p) { p->~T(); }

template<typename T>

void destroy\_at<T, true>(T\* p) {}

上述代码编译会报错，如gcc编译器提示：error: non-class, non-variable partial specialization ‘destroy\_at<T, true>’ is not allowed。函数模板不能偏特化，解决方案：

1. 通过包一层类模板解决

#include <iostream>

#include <cassert>

template<typename T, bool value = std::is\_trivially\_destructible\_v<T>>

struct wrap{

    static void destroy\_at(T\* p) {

        printf("not trivially\_destructible\n");

        p->~T();

    }

};

template<typename T>

struct wrap<T, true>{

    static void destroy\_at(T\* p) {

        printf("is trivially\_destructible\n");

    }

};

struct A{

    double x=10;

    ~A(){printf("A destruct\n");}

};

int main()

{

    void\* ap = malloc(sizeof(A));

    void\* bp = malloc(sizeof(int));

    A\* a=new (ap) A;

    int\* b=new (bp) int(100);

    wrap<A>::destroy\_at(a);

    wrap<int>::destroy\_at(b);

    free(a);

    free(b);

}

1. 使用编译期if constexpr

template<typename T>

void destroy\_at(T\* p) {

    if constexpr(!std::is\_trivially\_destructible\_v<T>){

        p->~T();

    }

}

**if** **constexpr** (...){...}

**else** **if** **constexpr** (...){...}

**else** {...}

# 模板显示实例化和外部模板

显式实例化，a.h中template <typename T> void fun(T) {}，

在b.cpp中

#include”a.h”

template void fun<int>(int); //全特化的语法为template<> void fun<int>(int)

void test1(){

fun(3);

}

则使编译器在本编译单元显示实例化出一个fun<int>(int)版本的函数

外部模板

c.cpp中

#include”a.h”

extern template void fun<int>(int); //如果注释这句，则会在当前编译文件中实例化函数模板

void test2()

{

fun(4);

}

则本编译单元不会实例化模板，而是在其它编译单元寻找符号void fun<int>(int)

template class std::vector<Magic>; //显式实例化

extern template class std::vector<Magic>; //不在该编译文件中实例化模板

//typedef可以为新类型定义一个新名称，但是没有办法为模板定义一个新的名称，因为模板不是一个类型。模板是用来产生类型的。using可以

template<class T>

using my\_vector=std::vector<T,std::allocator<T>>;

template<class T>

using my\_map = std::map<T, std::string>;

int main()

{

my\_vector<int> v1{1,3,5};

my\_map<long> m{ {1L,"zh"},{2L,"li"}};

}

在定义函数函数指针的时候，using看起来可读性要稍微好一点

typedef void(\*func)(int, int);

using func = void(\*)(int, int);

**inline与模板：**

C++ 中的inline，标准中规定的语义是，允许一个函数的在多个源文件中定义。被inline标记的函数可以直接定义在头文件中，即使被多个源文件#include，也不会导致链接错误

*// src1.cpp*

inline int add(int a, int b) {

    return a + b;

}

int g1(int a, int b) {

    return add(a, b);

}

*// src2.cpp*

inline int add(int a, int b) {

    return a + b;

}

int g2(int a, int b){

    return add(a, b);

}

*// main.cpp*

#include <cstdio>

extern int g1(int, int);

extern int g2(int, int);

int main() {

    return g1(1, 2) + g2(3, 4);

}

不开优化编译，g++ -c src1.cpp -o src1.o，g++ -c src2.cpp -o src2.o，则src1.o和src2.o中都有一份add函数的实例（可由objdump -d src1.o | c++filt查看）。g++ -o main main.cpp src1.o src2.o生成可执行文件时，链接器保留了两份add实例中的一份。C++标准要求，inline函数在不同编译单元的定义必须相同，否则违反ODR。此时如果main.cpp中声明并调用add()函数，正常运行。

打开-O2优化编译，g++ -O2 -c src1.cpp -o src1.o，g++ -O2 -c src2.cpp -o src2.o，则add()调用被完全内联，编译器都未给add生成符号。此时如果main.cpp中声明并调用add()函数，则报错"undefined reference to add(int, int) "

*// src1.cpp*

template <typename T>

int add(T a, T b) { return a + b; }

float g1() {

    return add(1, 2) + add(3.0, 4.0);

}

*// src2.cpp*

template <typename T>

int add(T a, T b) { return a + b; }

float g2() {

    return add(1, 2) + add(3.0, 4.0);

}

*// main.cpp*

extern float g1();

extern float g2();

int main() {

    return g1() + g2();

}

不开优化时，两个编译单元都实例化了add(int, int)和add(double, double)，各有一份代码。然后在最终链接的时候，链接器只为每个模板实例化保留了一份代码。在main.cpp中如果要调用add(int,int)函数，则可

template <typename T>

int add(T a, T b);

extern template int add<int>(int,int);

，或声明为extern "C" int \_Z3addIiEiT\_S0\_(int,int);也可以

打开-O2优化后，add调用完全被内联，连实例化出的函数的符号都没有。main.cpp中无法调用add()函数。

如果在src1.cpp中显示实例化，即在add模板定义后加一行template int add<int>(int,int);则无论是否开优化，都会生成add(int, int)符号。此时在main.cpp中

template <typename T>

int add(T a, T b);

extern template int add<int>(int,int);

然后就可以调用add(int, int)函数了

# 资源管理

资源，指一些对象，这些对象使用后必归还给系统，否则糟糕的事情会发生。

如堆内存、std::mutex对象、std::ofstream对象、数据库连接、网络sockets等

int\*p=new int(10);delete p;

ofstream file; file.open("a.txt"); file.close();

std::mutex mu; mu.lock(); mu.unlock()

手动释放资源容易被遗忘，因此创建自定义类型管理资源，“以对象管理资源”

* RAII （Resource Acquisition Is Initialization）获得资源后立即交给管理对象
* 管理对象有个私有数据成员是资源类型
* 管理对象的析构函数中释放资源

如管理指针的对象：

**class** SmartPtr

{

**private**:

**int**\* ptr\_;

**public**:

    SmartPtr(**int**\* ptr) :ptr\_(ptr) {}

    ~SmartPtr() { **if** (ptr\_ != nullptr) **delete** ptr\_; }

    SmartPtr(**const** SmartPtr&) = **delete**;

    SmartPtr& operator=(**const** SmartPtr&) = **delete**;

};

RAII对象（即用来管理资源的对像）要小心拷贝行为（拷贝构造、拷贝赋值函数）：

* 禁止使用，设置=delete
* 可以拷贝，但采用引用计数，RAII类的多个对象管理同一份资源，仅当最后一个对象销毁时，才释放资源（而不是在析构函数中释放资源）。如shared\_ptr的实现

实现引用计数拷贝的方法：

通常只要内含一个shared\_ptr 成员变量，RAII类便可实现出reference-counting copy行为

shared\_ptr允许在构造函数中指定所谓的“删除器”（deleter) , 那是一个函数或函数对象（fimction object) ，当引用次数为 0 时便被调用

//如自定义一个管理mutex资源的类，可以引用计数拷贝：

**class** shared\_mutex

{

**public**:

    explicit shared\_mutex(std::mutex\* mut) :mut\_(mut, [](std::mutex\* p) {(\*p).unlock();})//自定义shared\_ptr的删除器

    {

        (\*mut\_).lock();

    }

**private**:

    std::shared\_ptr<std::mutex> mut\_;

};

需要调用shared\_mutex的拷贝构造函数（或拷贝赋值函数）时，编译器为shared\_mutex类自动生成，调用shared\_ptr类的拷贝构造（或拷贝赋值函数）对私有数据成员mut\_实现拷贝。shared\_ptr类的拷贝是引用计数拷贝，这样则有shared\_mutex类的两个对象，每个都有一个数据成员mut\_。当最后一个shared\_mutex类对象销毁时，其数据成员mut\_销毁，引用计数显示这是管理同一资源的最后一个shared\_ptr对象，因此调用删除器

shared\_ptr可被用来自动解除互斥锁。可定制型删除器（custom deleter) ，这可防范 DLL 问题。

DLL问题：“对象在动态连接程序库（DLL) 中被 new 创建，却在另一个 DLL 内被 delete销毁”。在许多平台上，这一类“跨 DLL 之 new/delete 成对运用”会导致运行期错误。shared\_ptr 没有这个问题，因为它缺省的删除器是来自 “trl::shared\_ptr诞生所在的那个 DLL”的delete。

# 异常安全：

异常安全函数应保证以下两点之一：

* **不抛掷（nothrow) 保证，承诺绝不可能抛出异常**。作用于内置类型（例如 ints, 指针等等）身上的所有操作都提供nothrow 保证。
* 或者，**强烈保证：可能抛出异常，运行时抛出异常则程序回到“调用函数之前”的状态**，运行时没抛出异常则表示函数完全成功
* 或者，**基本承诺：如果异常被抛出，程序内的任何事物仍然保持在有效状态下**。如我们可以撰写 changeBackground 使得一旦有异常被抛出时，PrettyMerm 对象可以继续拥有原背景图像，或是令它拥有某个缺省背景图像

一般，不抛出异常的程序很难做到，任何使用动态内存的东西（例如所有 STL 容器）如果无法找到足够内存以满足需求，通常便会抛出一个 bad\_alloc 异常

典型的非异常安全程序：

**class** Image {};

Image init\_img;

Image\* bgImge = &init\_img;

std::mutex mut;

**void** changeBackground(**const** Image& imgsrc)  //该函数可能抛出异常的只有new Image(imgsrc)，其余代码都提供nothrow 保证

{

    mut.lock();

**delete** bgImge;

    bgImge = **new** Image(imgsrc);

    mut.unlock();

}

异常安全函数，抛出异常时，应做到：

* 不泄露资源，如果” new Image(imgsrc)”抛出异常，则mut.unlock()不会被执行，资源永远得不到释放
* 不允许数据败坏，如果” new Image(imgsrc)”抛出异常，bglmage 就是指向一个己被删除的对象

修改程序，使之提供强烈保证：

**class** Image {};

Image\* bgImge;

Image init\_img;

Image\* bgImge = &init\_img;

std::mutex mut;

**void** changeBackground(**const** Image& imgsrc)

{

    std::unique\_lock<std::mutex>(mut);

    Image\* temp = **new** Image(imgsrc);  //如果抛出异常，则unique\_lock销毁，自动释放互斥量mut，同时程序回到调用本函数前的状态

**delete** bgImge;

    bgImge = temp;

}

更应该，使用智能指针代替内置指针：

**class** Image {};

Image init\_img;

std::shared\_ptr<Image> bgImge(&init\_img);

std::mutex mut;

**void** changeBackground(**const** Image& imgsrc)

{

    std::unique\_lock<std::mutex>(mut);

    bgImge.reset(**new** Image(imgsrc));

}

这里不再需要手动 delete 旧图像，因为这个动作已经由智能指针内部处理掉了。此外，删除动作只发生在新图像被成功创建之后。更正确地说，shared\_ptr::reset函数只有在其参数（也就是 "newImage(imgSrc)"）被成功生成之后才会被调用。delete 只在 reset 函数内被使用，所以如果从未进入那个函数也就绝对不会使用 delete.

**feof函数：**

检测流上的文件结束符，如果文件结束，则返回非0值，否则返回0

feof判断文件结束是通过用户上次调用函数fread/fscanf等的返回错误来识别的，故而判断文件是否结束应该是在读取函数之后进行判断

#include <stdio.h>

int main ()

{

FILE \*fp;

int c;

fp = fopen("file.txt","r");

if(fp == NULL) {perror("打开文件时发生错误");return(-1);}

while(1)

{

c = fgetc(fp);

if( feof(fp) ){ break ;}

printf("%c", c);

}

fclose(fp);

return(0);

}

void test()

{

int i = 100;

}

# 命名空间：

using namespace std;可以写在#include语句后（对整个文件都有效），也可以写在函数里（仅在函数里起作用）

#include <iostream>

void testNameSpace()

{

using namespace std; //或者using std::cout; using std::endl;

cout << "test" << endl;

}

int main()

{

testNameSpace();

//cout << "main" << endl; //错误，未定义标识符

}

全局作用域包含程序所有源文件、头文件和库，对每个程序唯一，类和非static的独立函数都是全局域

* 同层次的作用域相互独立
* 不同层次域中的同名元素不冲突，而是遵循**向上就近原则**决定引用对象

    extern int c         //①

    static int c;         //②

    class A

   {

    public:

       int c;          //③

       void fun1();

    };

    A::fun1()

   {

      int c;             //④

      int a,b;

      c = a+b;          //⑤

   }

在A::fun1()内的局部域④、 classA的类域③、文件内部②及外部全局域①均定义有int c。引用c的语句④c=a+b位于fun1内，因此就近选④，如未定义④，次近就是类域里的③，如③也没有，再选文件域内的②，最后是全局的①。换种说法：局部域、类域、文件域到全局域逐层向上屏蔽。

**域限定符**::，可用该符号指定引用哪个域的元素，如前例fun1中可用::c=a+b;或A::c=a+b分别指代全局①和类域内③处的c

注意对A::fun1()这种类成员函数定义形式，限定符::之后包括函数名、参数类型、函数定义在内均被拖回了类域作用范围，因此A::fun1()定义部分可直接使用类成员而不必再用::限定，但**函数返回值类型位于::号之前，不受影响，仍位于类域之外**。因此：

    typedef string Mytype;  //①

    class B

    {

    typedef int MyType;   //②

    public:

       MyType fun2(MyType a);  //③

    };

    MyType A::fun2(MyType a)  //④::使所有阴影部分位于类域，而位于::前的返回值除外

    {

        ……

    }

    ④处A::fun2不只把fun2名字纳入类域，其后的形参MyType、函数定义体{}部分都进入类域，即④处形参MyType为类域中的int型②。但返回值MyType型位于类域之外，为①处的string型，这样函数返回值与③处声明不同。应改为：B::MyType A::fun2(MyTypea)

* 一个命名空间可以在两个地方被定义，即在全局范围、在另一个命名空间中 (这样形成嵌套命名空间)，不能在函数和类的内部定义
* 一个命名空间可以分散多个文件中，不同的文件中命名空间的定义是累积的。通常将命名空间的声明放到头文件中，实现放到源文件中。可以将不相关的成员放到不同的头文件中。

//a.cpp

**namespace** Ui //命名空间Ui的声明

{

**class** MainWindow;   //命名空间Ui中的class MainWindow

}

**class** MainWindow

{

**public**:**void** show() { cout << "全局范围的class MianWindow" << endl; }

};

//b.cpp

**class** ui\_MainWindow

{

**public**:**void** show() { cout << "全局范围的class ui\_MainWindow" << endl; }

};

**namespace** Ui  //命名空间Ui的实现

{

**class** MainWindow :**public** ui\_MainWindow {};

}

**int** main()

{

    Ui::MainWindow().show();//打印出：全局范围的class ui\_MainWindow

    MainWindow().show(); //打印出：全局范围的class MianWindow

}

（1）命名空间没有名字，其他文件显然无法引用，只能在当前文件起效。使用其成员无需也无法用UDNS::限定，直接调用fun1();即可。与此类似，static也限制元素作用域到当前文件，只不过无名空间能一次包含多个成员

#include<iostream>

using namespace std;

namespace

{

int a = 10;

}

void main()

{

cout << a << endl; //10

}

（2）UDNS名::成员名是最直接调用成员的方式，如testl::MyClass obj1;，但每次都这样重复写很麻烦，可先用using UDNS名::成员名的形式声明某成员，然后代码里就可以不加前缀直接使用，如：

using testl::MyClass; //声明其后出现的MyClass属于命名空间testl中的MyClass

MyClass obj1; //相当于testl::MyClass obj1

MyClass obj2;

（3）用using namespace UDNS名的方式则可一次声明全部UDNS的成员，如:

using namespace test1;

……

MyClass obj1;

fun1();

即用using namespace UDNS名声明后可直接使用该UDNS中所有成员而不必再用::限定。这种方式更简单，但别忘了创建UDNS的本意是防止命名冲突，轻易放开UDNS的全部成员又会增加冲突的概率

**printf格式化字符串：**

“%zu”用于打印size\_t类型

%zd"描述一个有符号数ssize\_t，其与无符号类型size\_t对应。如size\_t是unsigend long long，则%zd描述signed long long

%hu打印unsigned short，%lu打印unsigned long，%llu打印unsigned long long，%zu打印ssizet\_t

%hd打印short，%d打印int，%ld打印long int，%lld打印long long int，%zd打印size\_t

%f打印float

%lf打印double

size\_t test = 0xffffffffffffffff;

printf("test is %d\n", test); //-1

printf("test is %ld\n", test); //-1

printf("test is %zd\n", test); //-1

printf("test is %u\n", test); //4294967295

printf("test is %lu\n", test);//4294967295

printf("test is %zu\n", test); //18446744073709551615

# fread、fwrite、sprintf、sscanf、strspn、time函数：

#include <stdio.h>

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

fread和fwrite用于读写记录，这里的记录是指一串固定长度的字节，比如一个int、一个结构体或者一个定长数组。参数size指出一条记录的长度，而nmemb指出要读或写多少条记录，这些记录在ptr所指的内存空间中连续存放，共占size \* nmemb个字节，fread从文件stream中读出size \* nmemb个字节保存到ptr中，而fwrite把ptr中的size \* nmemb个字节写到文件stream中

例如   从文件fp里读取100个字节   可用以下语句    
fread(buffer,100,1,fp)

fread(buffer,50,2,fp)

fread(buffer,1,100,fp)

sprintf函数的功能与printf函数的功能基本一样，只是它把结果输出到指定的字符串中了，看个例子就明白了：

例：将”test 1 2”写入数组s中

char s[40]; sprintf(s,"%s%d%c","test",1,'2'); /\*第一个参数就是指向要写入的那个字符串的指针，剩下的就和printf()一样了

const char\* ip\_start = "192.165.56.97:80";

unsigned short m\_port = 80;

char hostname[64];

sscanf(ip\_start, "%[^:]:%hu", hostname, &m\_port);

std::cout << "hostname=" << hostname << std::endl; //hostname=192.165.56.97

std::cout << "m\_port=" << m\_port << std::endl; //m\_port=80

%hu代表以unsigned short格式输出整数

格式[]表示输入的字符是中括号中包含的几种du字符，比如%[A-Z]就是仅接收大写zhi字母A～Z的输入其他一概忽略

那么如果只接收冒号就要写%[:]

在符号前面加^，比如%[^:]就是除了冒号其他字符都可以输入

size\_t strspn(const char \*str1, const char \*str2) 检索字符串 str1 中第一个不在字符串 str2 中出现的字符下标。

# time、date函数：

time\_t now = time(nullptr);

std::cout << "now is: "<< now << " " << std::endl; // now is: 1592982675

time\_t preset = 1234567;

time\_t ret = time(&preset); // ret is 1592982675 preset is 1592982675

struct tm

{

int tm\_sec; /\* 秒 – 取值区间为[0,59] \*/

int tm\_min; /\* 分 - 取值区间为[0,59] \*/

int tm\_hour; /\* 时 - 取值区间为[0,23] \*/

int tm\_mday; /\* 一个月中的日期 - 取值区间为[1,31] \*/

int tm\_mon; /\* 月份（从一月开始，0代表一月） - 取值区间为[0,11] \*/

int tm\_year; /\* 年份，其值等于实际年份减去1900 \*/

int tm\_wday; /\* 星期 – 取值区间为[0,6]，其中0代表星期天，1代表星期一，以此类推 \*/

int tm\_yday; /\* 从每年的1月1日开始的天数–取值区间为[0,365]，其中0代表1月1日，1代表1月2日，以此类推 \*/

int tm\_isdst; /\* 夏令时标识符，该字段意义不大，很好用用夏令时。 \*/

};

//用time()取得时间 (秒数), 利用localtime() 转换成struct tm 再利用mktine()将structtm 转换成原来的秒数

time\_t timep;

time(&timep);//获取从1970年1月1日零时到现在的秒数

printf("time func return %d\n", timep); //1593310795

tm\* tt = localtime(&timep);

printf("%d-%d-%d %d:%d:%d\n", tt->tm\_year + 1900, tt->tm\_mon + 1, tt->tm\_mday, tt->tm\_hour, tt->tm\_min, tt->tm\_sec);//2020-6-28 10:19:55

time\_t cvted= mktime(tt);

printf("mktime func return %d\n", cvted); //1593310795

# nullptr函数传递给const char\*字符串参数的函数

const char\* src = nullptr;

const char\* x = strdup(src); //nullptr

const char\* path = nullptr;

FILE\* file = fopen(path, "wb");//会崩溃

char a[1024];

memcpy(a,nullptr,0);不会引起崩溃

# 按指定格式从字符串中读数据、将数据写入字符串

sscanf()：从一个字符串(指字符数组)中读进与指定格式相符的数据

fscanf()：从一个文件中读进与指定格式相符的数据

scanf()：从键盘读进与指定格式相符的数据

int a;

double b;

scanf\_s("%d%lf",&a,&b);

cout << a << " " << b << endl;

#include <iostream>

#include <cstdio>

#pragma warning(disable:4996) //屏蔽4994错误，否则提示sscanf应换为sscanf\_s

int main()

{

int i, j;

float x, y;

char str1[10] = { 0 }, str2[4] = {0};

char warr[4] = {0};

char input[] = "25 54.32E-1 Thompson 56789 0123 56hello";

// parse as follows:

// %d: an integer

// %f: a floating-point value

// %9s: a string of at most 9 non-whitespace characters

// %2d: two-digit integer (digits 5 and 6)

// %f: a floating-point value (digits 7, 8, 9)

// %\*d an integer which isn't stored anywhere，读一个整数但不会被赋给某个参数

// ' ': all consecutive whitespace

// %3[0-9]: a string of at most 3 digits (digits 5 and 6)

// %3s: 字符串，但字符数不能超过3

int ret = std::sscanf(input, "%d%f%9s%2d%f%\*d %3[0-9]%3s",

&i, &x, str1, &j, &y, str2, warr); //返回成功解析出的参数个数

std::cout << "Converted " << ret << " fields:\n"

<< "i = " << i << "\nx = " << x << '\n'

<< "str1 = " << str1 << "\nj = " << j << '\n'

<< "y = " << y << "\nstr2 = " << str2 << '\n'

<< "warr="<< warr << '\n';

}

/\*

Converted 7 fields:

i = 25

x = 5.432

str1 = Thompson

j = 56

y = 789

str2 = 56

warr=hel

\*/

#include <iostream>

#include <cstdio>

int main()

{

//无符号的整数（unsigned, 最大值是4294967295， 是个10位数）转为字符串， 不足10位时， 前面补0

char ret[11] = { 0 };

sprintf\_s(ret,sizeof(ret),"%010u",34); //0,表示不够宽度则补零，10表示宽度为10，u表示无符号整数

std::cout << ret << std::endl; //0000000034

std::cout << std::hex << std::numeric\_limits<unsigned>::max() << std::dec << " " << std::numeric\_limits<unsigned>::max() << std::endl; //ffffffff 4294967295

}

int snprintf(char\* dest\_str,size\_t size,const char\* format,...);

(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

**wb+/rb+/ab+**

FILE\* of = fopen(path, "wb+");

wb+读或写一个二进制文件，没有则创建

rb+读或写一个二进制文件，要求文件必须存在

ab+读或追加一个二进制文件，没有则创建

ofstream ofs("result.txt", ios::app);追加写

**以“wb”参数打开文件：**

回车、\r、0x0d

换行、\n、0x0a

LF：换行\n，0x0a

CRLF：回车换行\r\n 0x0d0a

windows使用CRLF作为文本文件换行符，linux使用LF作为文本文件换行符

abcd

1234

windows上新建txt，输入，则共10个字节，为0x61 62 63 64 0D 0A 31 32 33 34

linux上vim新建，输入，则共10个字节，为0x61 62 63 64 0A 31 32 33 34 0A，注意默认情况下，vim会在文件末尾自动添加换行符

linux下hexdump -C a.txt可以以16进制形式查看文件内容

linux下cat -A a.txt可以显示出非打印字符

vim编辑器可以自动识别LF或CRLF格式的文件

二进制方式（使用"b"参数）读文件时，会原封不动地读出文件的全部內容，写的時候，也是把內存缓冲区的內容原封不动地写到文件中。

Windows系统用”b”参数来表示Binary mode，也就是untranslated mode。那么translate的什么呢？translate的\n。说直白了就是windows为了兼容自己\r\n结尾和其它\n结尾的系统的差异，默认(不使用b参数）的情况下会将\n转换为\r\n。如下程序，

char str[] = "abcd\n";

FILE\* fp = fopen("file.txt", "w");

fwrite(str, 1, strlen(str), fp);

fclose(fp);

fopen时不使用"b"参数，那么这个程序在Linux上编译执行会写入"abcd\n"（即0x61 62 63 64 0A），在Windows上编译执行会写入"abcd\r\n"（即0x61 62 63 64 0D 0A）

以文本模式（不带”b"参数）写文件时，windows下会将0x0a或者说”\n”换成0x0d0a，因此如果是”\r\n”，则写到文件里是0x0d0d0a；

linux下会将”0x0a或者说”\n”原封不动写入，即写到文件里还是0x0a，因此如果是”\r\n”，则写到文件里是0x0d0a

调用std::ofstream::write想原封不动地写二进制数据时必须是std::ios::binary格式，否则可能出错，因为windwos上会将0x0a写为0x0d0a

ofstream与fopen表现一致

std::endl与”\n”表现一致

win10以二进制形式（带"b"参数）用ofstream向文件中写入换行：ofs << "\r\n"；不能是"\n"，也不能是ofs <<endl;

以文本模式用ofstream ofs("hello")写换行，则可以endl，可以”\n”、不可以”\r”或”\r\n”

int main()

{

std::ofstream ofs("out\_n.txt");

ofs << std::string("abcd\n1234");

std::ofstream ofs2("out\_rn.txt");

ofs2 << std::string("abcd\r\n1234");

std::ofstream ofs3("out\_endl.txt");

ofs3 << std::string("abcd");

ofs3 << endl;

ofs3 << std::string("1234");

}

/\*

windows上，输出的三个文件内容分别为：

61 62 63 64 0D 0A 31 32 33 34

61 62 63 64 0D 0D 0A 31 32 33 34 （用记事本或vscode看abcd与1234中空了一行）

61 62 63 64 0D 0A 31 32 33 34

Linux上，输出的三个文件内容分别为：

61 62 63 64 0a 31 32 33 34

61 62 63 64 0d 0a 31 32 33 34

61 62 63 64 0a 31 32 33 34

（用vim看三者显示都一样，vim应该是把第二种情况识别为CRLF格式）

\*/

文本模式读文件时：

windows会将0x0d0a全部识别为”\n”；

linux会将0x0a全部识别为”\n”；

int main()

{

std::ifstream ifs1("out\_n.txt");

std::string str1((std::istreambuf\_iterator<char>(ifs1)), std::istreambuf\_iterator<char>());

for (int i = 0; i < str1.size(); i++) cout << std::hex << (int)str1[i] << " ";

cout << endl;

cout << str1 << endl;

std::ifstream ifs2("out\_rn.txt");

//……

}

windows上读取windows上之前生成的文件，linux上读取linux上之前生成的文件，

运行结果，windows上与linux上一致：

/\*

61 62 63 64 a 31 32 33 34

abcd

1234

61 62 63 64 d a 31 32 33 34

abcd

1234

\*/

使用另一种解析方式：

int main()

{

std::ifstream ifs("out\_n.txt");

std::string str1, str2;

ifs >> str1 >> str2;

for (int i = 0; i < str1.size(); i++) cout << std::hex << (int)str1[i] << " ";

cout << endl;

cout << str1 << endl;

for (int i = 0; i < str2.size(); i++) cout << std::hex << (int)str2[i] << " ";

cout << endl;

cout << str2 << endl;

std::ifstream ifs2("out\_rn.txt");

if (ifs2)

{

//同上……

}

}

windows上读取windows上之前生成的文件，linux上读取linux上之前生成的文件，

运行结果，windows上与linux上一致，可以看到已经不能看到0x0a的踪迹了：

/\*

61 62 63 64

abcd

31 32 33 34

1234

61 62 63 64

abcd

31 32 33 34

1234

\*/

**文本结束符(ASCII为26)文件：**

文本结束符，ascii为26，即0x1a，即ctrlz，即^z

以文本模式读文件时，如果文件中有0x1a，windows上则认为文件已结束，即如果下一个字符时文本结束符，则ifs.peek()返回EOF

对Linux，将0x1a当做普通字符，不认为文件已经结束

旧操作系统DEC的TOPS-10的文件系统的最初版本并不记录文件的精确长度，用0x1a标记文件末尾。虽然现代的操作系统能精确记录文件长度，不再用0x1a标记文件末尾，现代编辑器也不再在文件末插入0x1a，但某些操作系统上（今天Windows版的C标准库）为了能正确读取旧系统上的文本文件，依然将0x1a作为文件结束标记对待。

测试程序：

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

{

ifstream ifs("a.txt");

if (!ifs) { cout << "not read" << endl; return -1; }

while (ifs.peek() != EOF) //不能使用while (ifs) 读完最后一个字符后，ifs仍未设置failbit，while(ifs)仍为真，因此会多读一个字节

{

cout << ifs.peek() << endl;

char x = ifs.get();

cout << "x:" << (int)x << endl;

cout << "ifs:" << (bool)ifs << endl;

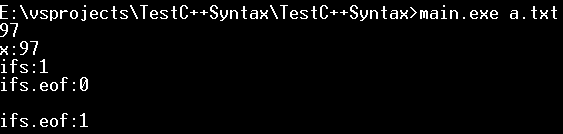
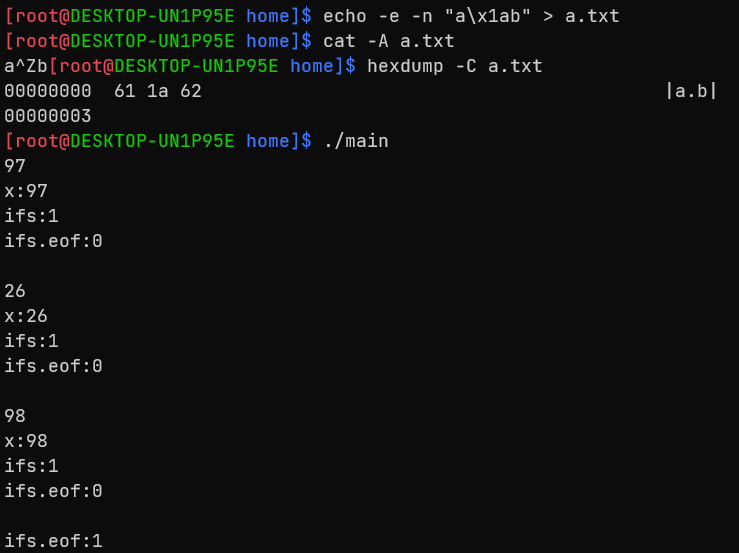
cout << "ifs.eof:" << ifs.eof() << endl << endl;

}

cout << "ifs.eof:" << ifs.eof() << endl << endl;

return 0;

}



**vsprintf：**

#include <stdio.h>

int printf(const char \*format, ...);

int fprintf(FILE \*stream, const char \*format, ...);

int dprintf(int fd, const char \*format, ...);

int sprintf(char \*str, const char \*format, ...);

int snprintf(char \*str, size\_t size, const char \*format, ...);

#include <stdarg.h>

int vprintf(const char \*format, va\_list ap);

int vfprintf(FILE \*stream, const char \*format, va\_list ap);

int vdprintf(int fd, const char \*format, va\_list ap);

int vsprintf(char \*str, const char \*format, va\_list ap);

int vsnprintf(char \*str, size\_t size, const char \*format, va\_list ap);

要封装sprintf类似的变参函数，则必须使用 stdarg里的对应版本:

void mysprintf(const char\* fmt, ...)

{

char s[32];

va\_list vas;

va\_start(vas,fmt);

vsprintf\_s(s,sizeof(s),fmt,vas);

va\_end(vas);

cout << s << endl;

/\*

//以下封装将不能处传递的变参

char s[32];

sprintf\_s(s, sizeof(s), fmt);

cout << s << endl;

\*/

}

int main(){

mysprintf("%d,%lf", 1, 2.3);

}

**ifdef：**

#if defined(A) || defined(B)

#if defined(A) && defined(B)

#if !defined(C) && !defined(D)：既没定义C又没定义D

#if !defined(C) || !defined(D)：没定义C或没定义D

**按位或时有负数且bit数不相同：**

unsigned short x0 = 0x0100;

unsigned char a1 = 0xe4; //内存中存储的是0xe4

unsigned short x1 = x0 | a1;

printf(" x1=%lld\n",(long long)x1); //x1=484

unsigned short y0 = 0x0100;

//char b1 = 0xe4; //内存中存储的是0xe4，但编译器认为b1是char类型的28

char b1 = a1; //内存中存储的是0xe4，但编译器认为b1是char类型的28

unsigned short y1 = y0 | b1; //y0是2字节，b1是1字节，先补为2字节，注意是2字节的负数short，然后按位或，然后再翻译为unsigned short，作为y1的取值

printf(" y1=%lld\n", (long long)y1); //y1=65508

short s1 = b1; //内存中存储的是0xffe4

printf("s1=%lld\n", (long long)s1); //s1=-28

/\*

按位与或非运算是按照数据的内部二进制形式进行运算的。

如果左右操作数位数不相同，则先补全至相同

若是负数，则是按二进制补码形式进行按位与或非。

所得结果若用有符号整型变量存储，则内部形式仍看作二进制补码

整数数值类型强转，如int、unsigend int时，如果前后bit数相同，则内存中布局不变，如果前后bit数不同，则会补0或补1

\*/

**运算符优先级问题：**

加减乘除优先级高于左移右移

(gdb) p 4 << 1 + 1

$13 = 16

(gdb) p (4 << 1) + 1

$12 = 9

**std::string的默认构造：**

由标准库的std::string实现我们知道，默认构造生成的字符串地址，均指向同一个全局静态变量。

std::string默认构造等价于std::string a=””;

因为是全局变量，所以每次跑程序它的地址都一样，因为这个地址是写死在exe文件里的

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

{

std::string tmp1;

std::string tmp2;

std::string tmp3="";

std::string tmp4="";

printf("t1=%p\n",(void\*)tmp1.c\_str()); //t1=0x7fcea9061338

printf("t2=%p\n",(void\*)tmp2.c\_str()); //t2=0x7fcea9061338

printf("t3=%p\n",(void\*)tmp3.c\_str()); //t3=0x7fcea9061338

printf("t4=%p\n",(void\*)tmp4.c\_str()); //t4=0x7fcea9061338

}

**空基类优化OBP：**

class A{};

printf("sizeof(A)=%zu\n", sizeof(A) ); //1

class B :public A{char m;};

printf("sizeof(B)=%zu\n", sizeof(B)); //1

class C :public A {};

printf("sizeof(C)=%zu\n", sizeof(C)); //1

class D{A a;char m;};

printf("sizeof(D)=%zu\n", sizeof(D)); //2

空基类优化：没有任何非static数据成员的基类，则称为空基类。对于继承空基类的子类，基类部分不会产生内存占用

组合里的空类能否优化？不能。上边class D的size不能是1，否则对于D的实例d，d.a和d.m是同一个地址，无法用地址来区分

**定制点与ADL：**

定制点：库提供并希望用户为自己的类型定制的定制点

ADL：实参依赖查找（Argument-Dependent Lookup）。简单的来说，ADL让你在调用函数的时候，只要有一个参数的类型属于函数所在的命名空间，那么调用的时候就不用加命名空间前缀（当然，实际的查找规则要复杂得多）。

namespace pet

{

    struct meow {

        friend void swap(meow&, meow&);

        friend void show(meow&, meow&)  {*//声明和定义都是inline的友元函数，是隐藏友元函数*

            printf("execute pet::show\n");

        }

    };

    void swap(meow&, meow&) {

        printf("execute pet::swap\n");

    }

}

int main(){

    pet::meow kit1, kit2;

    pet::swap(kit1, kit2); *//有限定调用*

    swap(kit1, kit2); *//无限定调用，ADL机制允许“与实参类型相关的函数（只能是函数或函数模板）”被找到，因此这里pet::swap是可以被找到并选择的，即使前面没有写 using namespace pet; 或者 using pet::swap，编译器通过ADL规则从kit们的类型签名 pet::meow 中发现了 namespace pet 并找到了里面的 pet::swap*

*//pet::show(kit1, kit2); //报错，对于隐藏友元函数，没有ADL以外的方式能够找到这一重载*

    show(kit1, kit2);

}

Q：为什么要使用ADL，在要用到class meow的每个地方都写pet::swap，而用到标准库设施的地方都写std::swap不就行了？

A：在泛型语境下，一个一致通用的调用格式是必须的。借助ADL，则可以通过无限定调用的格式查找并调用匹配的重载，不论函数模板接受的是std中的类型还是pet中的类型，ADL都可以通过类型签名决定调用的是std::swap还是pet::swap。

template<typename T>

void GenericOperation(T&& first, T&& second){

*//...*

*/\*这里应该是什么？\*/*::swap(first, second);

*//...*

}

但是基本类型怎么办？

int a = 0, b = 0;

swap(a, b); //报错

虽然标准库在namespace std下提供了基本类型的swap，但是基本类型的签名里并没有std或者别的什么名字提供给ADL查找信息。这一写法对于基本类型失效了，work around方案如下：

int a = 0, b = 0;

using std::swap;

swap(a, b);

综上，一个丑陋但完善的泛型示例如下：

template<typename T>

void GenericOperation(T&& first, T&& second)

{

    using std::swap;

    swap(first, second);

}

int main()

{

    pet::meow kit1, kit2;

    GenericOperation(kit1, kit2); *//打印出execute pet::swap*

    std::vector<int> va = { 1,2,3 };

    std::vector<int> vb = { 4,5,6 };

    GenericOperation(va, vb);

    printf("va[0]=%d\n", va[0]); *//va[0]=4*

    int a = 1, b = 2;

    std::swap(a, b);

    GenericOperation(a, b);

    printf("a=%d\n", a);  *//a=1*

}

一个前现代C++的定制点典范是对于iostream的输入输出操作的定制。众所周知，只要为自己的类型添加

friend std::ostream& operator<<(std::ostream&, const MyType&);

friend std::istream& operator>>(std::istream&, const MyType&);

自己的类型就能和iostream交互了！这些运算符的重载会通过MyType的签名被ADL发现并使用，而对于基本类型，他们的重载可以通过流对象的签名被看到。

int main(){

    std::cout << "hello";

    operator<< (std::cout, "hello"); *//与上一句等价*

*// 由于ADL，因此可以找到std中的std::operator<< (std::ostream&, const char\*)*

*// 如果没有ADL，就必须写std::operator<<(std::cout, "hello")*

}

隐藏友元函数的优势之一即是将作用域限制在class里

//https://www.zhihu.com/question/645167095/answer/3403862858

struct Age {

    friend void show(const Age& a) { std::cout << "age is " << a.year << std::endl; }

    friend void meet() {std::cout << "hello\n";}

    friend void apologize();

 int year=5;

};

void apologize() { std::cout << "sry\n"; }

int main() {

    Age f;

    show(f); *//隐藏友元，ADL*

*//meet(); //隐藏友元，符号对外不可见，报错找不到标识符*

    apologize();

}