# C++

面向对象、泛型编程

注意

swith（）case 必须跟常量

float/double 必须写1.0不然结果会变成整形int

# 一、基础

## hello world

#include <iostream> //标准输入输出流

useing namespace std； //使用命名空间，

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

// 标准的命名空间std

// cout标准输出

// << 左运算符，拼接基础类型，字符串

// endl结束换行

**std::**cout << “hello world” << **std::**endl;

system(“pause”);//阻塞

return EXIT\_SUCCESS; // 返回正常退出

}

using namespace std;

## ：：运算符

作用域运算符

::atc //全局atc

int atc=100;

void function()

{

cout << ::atc <<endl //使用全局atc

std::cout //std作用域下的cout方法

}

## namespace关键字

//命名空间必须定义在全局作用下

//命名空间可以嵌套命名空间

//命名空间可以放函数，变量，结构体类

//命名空间是开放的，可以随时添加进原来的命名空间，**同一个命名空间名会合并**

namespace LOL // 声明一个命名空间

{

void goattc();//函数

int a;//变量

struct Person{};//结构体

class B{};//类

namespace D{};//命名空间LOL::D::

}

//无名称的命名空间

namespace

{

int a; //相当于static int a;

}

## using关键字

using namespace name //using编译指令

using name //use声明，在函数内使用会有二义性，声明原则比编译指令高，

## C++增强

1、全局变量检测增强，不能重新定义已定义的变量

2、函数检测增强，返回值，参数定义

3、类型转换增强，例如malloc必须强转(int \*)

4、struct增强，可在结构体中添加函数；

声明时可不添加struct关键字

1. bool类型增强，布尔类型对于非0的值都会转换为1
2. 三目运算增强，（a>b?a:b） = 100,C语言返回的是值，C++返回的是变量
3. const增强，可以初始化数组 int a[mab],**在C++中尽量使用const代替#define**

## 引用

//1、不要返回局部变量的引用，在函数中运行结束，变量会被释放

//如果返回的值是一个引用，则函数可当左值swap() = 1000

//int &a = 100 //int\* const a = &a

//引用后不能修改

int a[10] = {0};

int (&arr)[10] = a; //给数组起别名,arr指向a的空间

int b = 200;

int &c =b //c操作b的空间， 不能再修改

int &cc(); //引用类型的函数

## 引用传递

void swap(int &a, int &b)//注意这里的&

{

//通过别名的方式操作数据，本质也是操作地址

int temp = a;

a = b;

b = tmp;

}

int main(){

int a=200;

int c=100;

swap(a, b);//引用传递，这里放的是值

}

## 指针引用

二级指针的优化

void allow(Person &p)

{

//引用传值，p指向的是Person的空间

p = (Person\*)malloc(sizeof(Person))

p->m\_age = 1000

}

int main(){

Person \*p = Null;

allow(p) //引用传值

}

# 指针传递/引用传递/值传递区别

typeof struct {

int a;

}student;

int main()

{

student p1 = {111};

value\_pass(p1); // 值传递

//-------------

student p2 = {222};

pass2(**p2**); //引用传递

//-------------

//student p = {222};

// student \*p3 = &p;

student \*p3 = (student\*)molloc(sizeof(student));

p3->a = 333;

pass3(p3); //指针传递

}

## 值传递

void value\_pass(student p)

{

p.a = 222; //这里不会修改值

}

## 引用传递

void pass2(student **&p**){

p.a = 222; //这里会修改值

}

## 指针传递

void pass3(student \*p){

p->a = 222; //这里会修改值

}

# 二、类

## 1、定义类

**NPC.h文件中定义函数**

使用宏定义防止多次加载

#ifndef \_\_xxx

#define \_\_xxx

class Rect{} //第三方类

class NPC

{

NPC（）； //构造方法,与类名相同

~NPC（）； //析构方法

Rect \* a; //引用另外的类

private:

protected:

public:

int Name； //成员属性

void move(int a); //定义一个方法

}

#endif //宏定义

## 2、实现类

**NPC.cpp文件实现方法**

**#include “NPC.h” //引入头文件**

NPC::NPC(){} //实现构造方法

NPC::~NPC(){ //析构方法

delete this->a; //释放空间a

}

void NPC::move(int a){ //::成员操作符

std::cout << “1”;

}

## 3、调用

**main.cpp**

#include “NPC.h”

int main()

{

NPC n1 // 创建一个类，分配空间

n1.move(1) //调用类中的方法

//-------------------------- 动态生成一个对象

NPC \*n2；

n2=&n1 // 可以指向一个已经创建的对象

n2=new NPC() // new一个对象

//-------------------------------

NPC n3(1) //调用构造函数

NPC \* n3 = new NPC(1)

}

## 类中的函数和参数

默认是 private 继承，在 struct 中默认 public 继承：

class Person

{

int age; **//默认情况下，类的所有成员都是私有的，私有属性**

static int m; //静态成员不属于类

static void func3(); //静态成员函数不属于对象

public:

void func(); //这个函数指向的同一个函数空间，非静态函数，不属于对象

}

Person p1;

p1.func(); //调用公有方法，私有方法只能在类内部调用

## 5、this指针/返回this对象

this指针指向被调用的成员函数

不写this编译器也会添加this

class Person

{

public:

int age;

int \_age;

void func(Person \*this){}//this默认添加的，this指针永远指向自己

Person(int age)

{

age = age; //这里编译器不能区分是不是类中的age

\_age = age;//也可以区分，

this->age = age;

}

**// 返回当前对象指针**

// 如果不加& 会返回一个person实体，这个Person实体是新的

Person**&** ss() //返回当前对象的指针

{

return \*this

}

}

Person p1(10);//初始化

## 6、mutable

### 常函数

限制修改类属性的值

class Person

{

int a;

void showInfo() const //长函数，不允许修改指针的值

{

this->a = 100

}

}

mutable int B; //可以修改长函数中的变量

### 长对象

const Person p2; //常对象，不允许修改

cout << p2.name //可以读

常对象可以调用常函数，不能调用普通函数

## 7、重载+号运算

class Person

{

public:

operator+(){} //重载+号运算

}

Person operator+(){}

Person p;

p() //需要重载()方法 ，仿函数调用，因为p是一个Person类型的值，（）是一个运算符

## 友元函数

可以访问指定类中的私有方法

class Buliding

{

friend void goodGay(Buliding\* bul);

friend class goodGy；//类做友元

}

//全局友元函数

void goodGay(buliding\* bul）{

//这里可以调用Buliding访问私有方法属性

}

## 内联函数（C++宏函数）

**定义：**

**//添加（）是为了防止程序运算顺序出错**

// 宏没有作用域

// 宏在运算++运算时可能会超出预期

#define Myadd(x,y) ((x)+(y))

//内联函数为了解决宏直接替换可能产生的各种错误

//函数声明和实现都需要加inline

inline void fun(int a){

return ++a;

}

## 5、函数默认参数\占位符

void func(int a=10, int b=30){}

//int 为占位，但在函数内用不到，因为未定义参数名，调用时必须提供方法

void func(int, int){}

## C++调用C语言代码

extern “C” void aaa();// 告诉编译器以C语言的方法处理

**定义头文件使用C编译**

#ifdef \_\_cplusplus

extern “C” {

#endif

...

#ifdef \_\_cplusplus

}

#endif

## 继承

### 基本

**bash.h文件**

#include “NPC.h” //引入要父类的头文件

class Other : NPC{} //继承NPC，默认是**private**继承

class Other : public NPC,NPC2 {} //继承多个父类

class Other : **public** NPC {} //公众继承

//NPC中的类public protected private不变

class Other : **protected** NPC {} //保护继承

//NPC中的public 变为 protected

class Other : **private** NPC {} //父类属性变为private

//NPC中的private protected变为private

### 调用父类构造函数

class Base

{

int s;

Base(int a){}

}

class Son:public Base

{

int s;

Son(int a) : Base(a){} //调用父类构造函数

}

### 同名函数

// 如果子类与父类的函数同名，子类会把父类的所有同名版本都隐藏掉

// 想调用父类的方法需要添加作用域

Son s1;

s1.Base::s //调用父类Base中的属性

s1.s; //默认调用子类中的属性

## 重载

在**同一个类**中有相同的名称，但参数不同(包含类型不同)，系统会根据参数不同，调用不同的方法

class A

{

void B(int a){};

void B(int a, int b){}; //重载，

}

//-------在同一个作用域

void func(int a, int b){}

void func(){}

func(1,2) //调用有参数的方法

func() //调用无参数方法

## 封装

class与struct基本一样，但struct默认时public，class默认时private

struct Pres

{

int a;

void aa()

{

cont<< a << “pres 的a属性” <<endl;

}

}

## 多态

class Animal

{

**virtual** int speak(){}

}

class A:public Animal

{

int speak(){return 0;}

}

void dospeak(Animal& animal)

{

animal.speak(“ssss”); //动态多态联编

}

int main(){

A cat;

dospeak(cat)

}

多层继承会依次调用各个层的构造方法

## 构造方法与析构方法

**编译器会自动提供这两个函数，但函数体是空的**

**《在栈上可以由系统释放，在堆上要自己释放》**

**在main中定义要在程序完全结束时才清理**

**如果函数中定义,因为函数时在栈中，在函数结束时会调用析构函数**

构造和析构函数必须在pulic下

class Person{

Public:

Person(){} //构造方法，与类名相同，**没有返回值可以重载**，编译器自动调用

Person(int a){} **//有参构造**

Person(const Person &p){} **//拷贝构造函数-需要把类传入**

~Person(){} //析构方法，没有返回值，**没有参数**，内存释放的时候调用

}

Person P1 //默认调用构造函数

Person P2() //函数声明

Person p3 = Person(100) //匿名类

Person p5(p3) //匿名函数不能拷贝构造

Person p6 = 100 //***Person p6 = Person(100) 隐式转换***

## 构造函数的分类和调用

拷贝构造函数使用时机

1、用已经创建好得对象初始化新类(Person p2(p1))

1. 值传递，将类作为参数（值传递，都会调用拷贝构造）
2. 以值得方式返回局部对象（会占用更多内存）

relese与debug模式运行结果不同

## 构造函数的调用规则

当提供了有参得构造函数，系统会调用默认无参构造函数，但会提供拷贝构造

**系统会提供默认构造，拷贝构造，析构函数**

**系统默认得拷贝构造会进行简单得拷贝方法**

当我们提供了拷贝构造，系统就不会提供其他构造方法了

## 构造和析构顺序

1\先构造父类-再构造子类

析构-先析构子类再析构父类（父类设置虚析构）

## 浅拷贝与深拷贝

浅拷贝-并没有正在的拷贝，而是用指针指向了同一块空间，是由于析构（释放）内存时，浅拷贝在第一次就把资源释放，第二次释放就出错了

深拷贝-正在的拷贝了一次

自己提供拷贝构造，（默认拷贝构造只是指向同一块内存空间）

m\_name = (char \*)malloc(strlen(p.name)+1)

strcpy(m\_name, p.name) //堆区复制，=是在栈区

## 初始化列表

class Person

{

//Person中定义参数

int **m\_A**;

/\*

直接实现赋值，不用再实现中赋值

接收a,b,c变量，

将int a的a变量传给m\_A

\*/

Person(int a): **m\_A**(a){}

/\*

直接实现赋值，不用再实现中赋值

将10赋给m\_A

\*/

Person():m\_A(10){} //写死的参数赋值

}

## explicit 隐式转换

class Person

{

explicit Person(int a){} //防止隐式转换

}

Person str = 100 //隐私类型转换

## 内存分配-new

1\malloc必须知道长度

malloc返回一个void\* 必须强转

malloc可能申请失败，严谨时需要判断返回是否为NULL

malloc不会自动调用构造函数

//派皮一个空间

this->name = new char[strlen(s)+1];

char\* a = new char[strlen(2)+1]

**//new出来的对象，都会返回该类型的指针，开辟位置在堆区**

Person \* p = new Person;

Person\* p= new Person[10]; //初始化10，一定要提供默认构造

Person parr[2] = {Person(1), Person(2)}

相当于

Person\* person = (Person\*)malloc(sizeof(Person));

if(person == NULL){return 0;}

person->Init();//调用构造函数

释放空间

**delete p;**

**delete [] p;//释放数组**

## 静态成员变量/静态成员函数

所有相同类名的类都会共享一个static 变量；

静态变量在编译的时候就已经创建

静态变量只能声明

class A

{

public:

static int m\_age; //声明

}

int A::m\_age = 100;//静态成员变量一般只赋值，通过类名进行赋值

//私有权限是无法访问，当可以初始化

## 单例模式

class ChairMan

{

private:

ChairMan(){} //1、私有化构造方法，

ChairMan(const ChairMan& c){} //1、私有化拷贝构造

static ChairMan \* singleMan; //2、创建指针，保存类指针

public:

static ChairMan\* getInstance();//静态成员通过类名访问

{

return singleMan;

}

}

ChairMan \*ChairMan::singeMan = new ChairMan;//3、编译时创建

int main()

{

ChairMan \*cml = ChairMan::getInstance(); //4、程序调用

}

## 虚函数-抽象类

**虚基类解决数据重复问题---菱形继承**

**虚基类不用添加类名，因为只有一份数据，**

**class sheep:virtual public Aninal{} //虚基类**

**class sheep**

**{**

**public:**

**virtual int Sleep(){} //虚函数**

**}**

虚函数永远执行子类中的方法（方法重载）

实现父类指针调用子类方法

如果父类有虚函数，就不能实例化对象

有虚函数就叫抽象类

class NPC

{

virtual void fire(int a); //虚函数定义

virtual void fire(int a)=0; //纯虚函数，子类必须实现

}

class Sound{

virtual void play()=0;

}

void sound::play(){}

#include “sound.h”

## 虚析构和纯虚析构

**通过*父类*指针指向子类对象时释放不完全（因为不会调用子类析构）**

普通析构方法只会调用当前指针类型的析构方法

利用虚构解析解决这个问题

class Animal

{

virtual ~Animal(){}; //父类定义了虚函数，这样

virtual ~Animal() = 0;//纯虚析构

}

//纯虚析构，需要在内中声明，在类外实现

//如果类中有纯虚析构函数--这个类也是抽象类

Animal::~Animal() **//纯虚析构实现**

{

cout << “animal纯虚析构”;

}

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

class Animal

{

public:

virtual void speak(){//说话}

virtual ~Animal(){} //虚析构，防止子类未释放

}

class Cat:public Animal

{

public:

Cat(const\* name){

this->name = new char[strlen(name)+1]; //在堆中开辟空间

strcpy(this->name, name);

}

void speak(){//.....};

~Cat() //因为在堆空间开辟空间，所以要手动释放

{//如果父类没有定义

if (this->name !=NULL){

delete [] this->name;

}

}

char \* Name

}

void test(){

Animal \* animal = new Cat(“TOM”); //父类指针指向子对象

Animal->speak(); //多态，调用Cat类的speak方法

//释放空间---这里会先调用父类animal的析构，

// 如果没有定义虚析构-就不会调用子类析构

delete animal ；

}

## 操作符的重载

class NPC

{

void operator++(); //重载++操作符

}

void NPC::operator++(){} //重新实现++操作符

# 模板-泛型编程

如果普通函数和模板函数发生了重载，优先调用普通函数

如果强制调用模板getMax<>(a,b) ,使用空参数列表

模板可以发生重载

## 模板内部

## 具体化

具体化自定义数据类型

具体化会优先匹配

template<> bool myfunc<Person>(Person &a, Person &b)

{

if(a.name == a.name){}

}

## 模板函数

//定义一个模板函数

// 函数名getMax

template<typename T> //效果与以下一样

template<class T> //T 是一个通用类型，T必须是一个类型，如int，

T getMax(T a, T b) {

return a>b?a:b;

}

int main()

{

// 不用特殊定义属性，模板函数会自动返回T类型的值，输入T类型的变量（T类型，是根据你的输入自动变换的）

getMax(1,2)；

getMax(“a”,”b”)；

getMax<int>(1,2)//显示指定类型

}

## 模板类

模板定义的变量是可变的，其他的都一样

template<class T,class T2=int>//可以添加默认类型

class LinkList{

T name;

T2 age;

LinkList(T name, T2 age){}

}

int main()

{

string a; int 1;

LinkList<string, int>(a, b) //必须使用显示转换，函数模板可以自动推导，类模板不能推导

}

## 模板做函数参数

void doWork(Person<string, int> &p)

{

P.show()

}

//参数模板化

template<class T1, class T2>

void doWork(Person<T1, T2>&p){...}

## 类模板与继承

template<class T>

class Base

{

public:

T m\_A;

}

//必须告诉child类型,否则不能分配内存

class Child: public Base<int> {}

//由用户指定类型

template<class T1, class T2>

class Child: pulbic Base<T2>

{}

## 模板的份文件编写

1. 模板的文件是.hpp
2. 方法实现与声明放到同一个文件中

因为C语言是份文件编译，但模板文件类型是在运行时确定，所以不能使用.h方法引入

可以直接引入.cpp文件，

模板文件编译在运行时会出错

## 模板与友元函数

# 异常

try{} catch(int) {}//捕获int类型异常

try{} catch(){} catch(){} //捕获多个异常

try{} catch(...){} //捕获其他类型异常

throw 1//抛出异常，可被catch抓到

throw 3.14//抛出double异常

自定义异常类

class my Exception

{

public:

void myException();

}

throw MyException(); //抛出异常类

catch(MyException e)

## 栈解旋

从try开始，到throw抛出异常之前，所有栈上的对象都会被释放

## 异常接口声明

void func() throw() //不能抛出任何异常

void func() throw(int) //抛出int类型异常,当前函数只能抛出int错误

{

throw 1;

}

## 异常生命周期

throw new MyException();

catch(MyException \*e)

{

delete e;

}

## 异常多态

## 标准异常库

#include<stdexcept>

throw out\_of\_range(“年龄越界”); //越界异常

e.what() //异常信息

# 标准输入流

cin.get() //获取一个字符，接收换行\n

cin.get(buf, 1024)//将字符放到buf变量中，获取1024个字符,读取字符串不会拿走\n需要在读取一次

cin.get()

cin.getline(buf, 1024)//读取换行符，并把换行扔掉

cin.ignore()//忽略一个字符

cin.ignore(n)//忽略了n个字符

cin.peek()//查看，然后再放入缓存区，用cin.get()可获取

char c = cin.get();

cin.putback(c); //将字符串返回缓冲区

# 标准输出流

cout.flush()//刷新缓冲区linux有用

cout.put(‘a’).put() //放入缓冲区

cout.write()

cout.width(20)

cout.fill(“\*”)

cout.unsetf(ios::dec)//卸载10进制显示

cout.setf(ios::dec)//十进制显示

# 文件读写操作

#include<fstream>

// ios::in 输入方式打开

// ios::out

// ios::

ofstream ofs(“./test.txt”, ios:out | ios::trunc);

ofstream ofs;

ofs.open(“./test.txt”);

if(ofs.is\_open()) //判断打开情况

ofs<< “想要向文本输入的”;

ifstream ifs;

ifs.open(“./test.txt”, ios::in);

char buf[1024];

while(ifs>>buf)

{

couf<<buf<<endl; //读取完成后，输出

}

while(!ifs.eof())

{

ifs.getline(buf,sizof(buf2))

}

# STL（标准模板库）

分为容器，算法，迭代器

## 三大组件-容器-算法-迭代器

迭代器：

include<vector>

vector<int> v;//声明一个容器，int类型，对象名称为v

v.push\_back(10); //尾插

vector<int>::iterator it = v.begin();//it指向容器起始位置

vector<int>::iterator itEnd = v.end();//容器最后位置的下一个

for( vector<int>::iterator it = v.begin();it ！=v.end();i++)

{

cout<< \*it<<endl;

}

while(it != itEnd)

{

cout<< \*it;

it++

}

include<algorithm> //算法头文件

void myPrint(int a);

for\_each(v.begin(), v.end(), myPrint);//遍历

自定义类型

class Person{}

vector<Person> P

vector<vector<int>> v;//容器列表

vector<int> v1;

v.push\_back(v1)

## string容器-类

include<string>

C风格字符串太过复杂，不适合大程序开发

char\* 是一个指针，string是一个类，sting内部管理一个char\*

string封装了很多实用成员方法

不用考虑内存释放和越界

string str;默认构造

string str2(str) //拷贝构造

string str3 = “abcd”; //string str3 = string(“123”)

str = “123”; //重写了=号

str = str4; //多态构造=号，&a

str.size() //字符串大小

str.at(i) == str[i] //表现一样，【】访问越界挂掉，at会抛出异常

str.append(“xxxx”) //尾部追加

str.find(“b”)//查找从0开始，最先查找的位置，没找到返回-1

str.replace(1,2,”111”) //替换

str.compare(str2) //字符串根据ASCII码对比

str.substr(0, 3)//返回从0开始的3个字符串

const char \* p = s.c\_str(); //转换为C风格字符串

string s2(p) //转换为string类型的字符串

//隐式类型能从const char \* 转换为string.但不能从string转换为const char\*

toupper(a) //将字符转换为大写

tolower(a) //转换为小写

注意

string a = “123456”;

a[1] =’a’//这样修其中的字符

a = ‘1234567890’;//这里字符过长，内存重新分配

## vector-栈结构-单钻数组-动态数组

push\_back()

vector<int>v2(v1.begin(), v1.end());

.size()// 返回容器中元素的个数

.empty()//判断容器是否为空

.resize(int num);//重新指定容器长度为num,若容器变长则以默认值填充

capacity()

reserve()

vector<int>(v).swap(v)//节省空间

## deque-队列

s双端数组，没有容量，

push\_front()

push\_back()

pop\_front()

pop\_back();

deque<int> d;

iterator//普通迭代器

reverse\_iterator//逆序迭代器

const\_iterator//只读迭代器

## stack容器

先进后出的容器-不允许有遍历行为-没有迭代器

include<stack>

stack<T> a; //stack采用模板类实现，stack<int> a;

1. push(10) //添加元素
2. pop() //弹出一个数据
3. size() //读取stack大小， while(s.size() !=0 )// empty
4. top() //访问栈顶元素-不弹出

a.empty()//判断是否为空

## queue容器

先进先出-但只能再队尾添加，队头删除-没有迭代器

queue<int> a;

1. push(10) //队尾添加元素
2. pop() //对头弹出
3. empty() //判断是否为空
4. front() //查看队头元素
5. back() //查看队尾元素

## list容器-双向循环链表

动态分配，不会造成内存浪费和溢出-提供双向迭代器

不支持随机访问

#include <list>

list<int> m; //定义一个

list<int> m(10， 10) //构造方式

m.push\_back() //尾插

m.push\_front() //头插

m.pop\_front() //弹出

m.pop\_back() //尾弹出

m.clear()//删除

m.insert(m, 100) //

m.arase()

m.remove(10) //删除指定值，所有值

m.assign()

m.reverse()//反转排序， 所用不支持随机访问的迭代器，不可以用系统提供算法

//对于自定义排序，需要指定排序规则

m.sort(mysort) //排序

bool mysort(Person &p1, Person &2)

{

if(p1.mage> p2.mage)

{

return true

}else{

return false

}

}

## set容器

根据元素进行排序，不允许有2个相同的值，不允许修改

multiset 容器，红黑二叉树

s1.insert()返回值是pair

set<int> s1;

s1.insert() //插入值 -关联式容器，会自动排序

s1.erase() //删除，可传值

s1.find()//返回一个迭代器 set<int>::iterator pos = s1.find(3),未找到返回s1.end()

s1.count()//

s1.lower\_bound(keyElem)

pair对组

pair<set<int>::iterator,set<int>::iterator> res = s1.equal\_range(0)

ret.first //第一个值

ret.second //第二个值

创建队组

pair<string, int> p2=make\_pair(“jjjj”,200)//创建

pair<string, int> p(string(“tom”),100) //创建

p.first

p.second

插入前设置排序规则-插入自定义排序，需要指定排序规则

set<int, mycompare>s1 //默认从小到大排序

class mycompare{

public:

bool operator()(int v1, int v2)

{

return v1>v2;

}

}

## map容器

所有元素会按照key排序，不允许有2个相同的key值-红黑树实现

include<map>

map<int, int> m;

m.insert(make\_pair(2,20));

m[n] = 33; //系统会摩天使用0填充

m.clear() //删除

m.erase()//删除指定值

## 使用时机

vector ke

## 算法-函数对象-仿函数-类-重写了（）

一元谓词 二元谓词

#include<algorithm>

<numeric>小体积

<functional>

//这里function是一个函数对象，

vector<int>::iterator pos = find\_if(begin,end, function())

class function

{

public:

//重载()

}

## 遍历算法

callback可以是函数，仿函数

for\_each(iterator begin, iterator end, callback)

myprint p = for\_each()

p.m\_Count

vector<int>v;

v.resize(v.size()) //开辟空间

transform() //搬运数据，不会开辟空间

## 查找算法

find(start,end, value) 返回一个迭代器

自定义数据

binary\_search()//二分查找，必须有序序列

## 排序算法

sort()

merge() //合并，必须有序

random\_shuffle(begin, end) //将指定数组随机排序，需要自己提供种子

reverse() //反转

## 拷贝和替换

copy()

replace()

replace\_if()

swap()

## 算数生成

include<numeric>

accumulate(begin, end, 0) //计算元素累计总和,累加计算，最后一个是起始值

fill() //向容器中填充元素

## 集合算法

交集，并集，差集（两个集合相减）返回值是集合的最后一个位置

set\_intersection() //交集min(v1.size(), v2.size())

set\_union()//并集vi.size() + v2.size()

## lambda表达式

[](int a){cout<<”ss”}

## 内建函数对象

include<functional>

//template<class T> negate<T>

negate<int> v;//取反

cout<< v(10) ;//-10

plus<int> p;

cout<< p(1,1) //加法

## 适配器

提供函数应提供函数名，提供仿函数function()

# 类型转换

向下转换不安全，线上转换安全

**静态转换**

类属性转换必须有父子关系

基础类型转换

char a=”a”;

doublie b = static\_cast<double>(a);//静态类型转换成double

**动态转换**

**基础类型不允许转换，失去精度或不安全都不可以转换**

**只能向下转换**

char a=”a”;

double d = dynamic\_cast<double>(a);

常量转换

不能直接对非指针或引用使用

const int \*p = NULL;

int\* p = const\_cast<int \*>(p);//去掉const

int\* p2=NULL;

const int p2 = const\_cast<int \*>(p2); //添加const

# C++11 新特性

## 1、auto 自动实现类型转换，

auto 可以指向任何函数，实现类型转换【弱类型语言】

auto 定义的就是，任意类型语言，可以放任何东西

## 2、增强的for循环

**[可以理解成for...in]**

for(auto tmp:allNPC){

//tmp是allNPC中的每一个值

tmp->move(30,40) //移动

}

## lambda 匿名函数

[] 使用外部变量

auto a = **[**=赋值方式、&引用方式**](**参数**){**代码块**}**

a(); //调用

# C++内存管理对象

new 创建对象、delete删除对象