visual studio快捷键ctrl k + c一键注释， ctrl k + u一键取消注释

1.内存四区：

代码区：存放函数体的二进制代码、注释等，由操作系统进行管理

1）代码区的特点：共享的、只读的

全局区：存放全局变量和静态变量和常量（字符串常量、用const修饰的全局变量），该区域的数据在程序结束后由操作系统释放

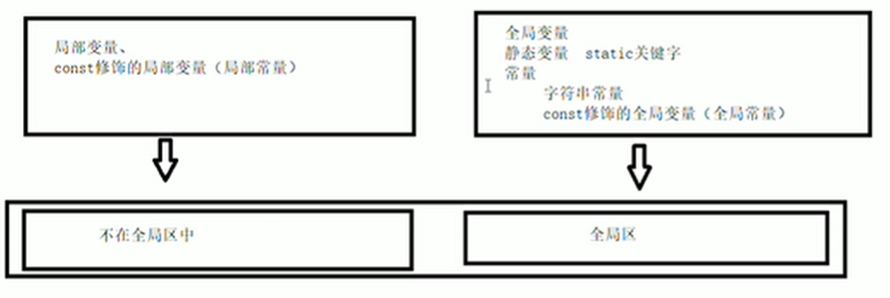
1）静态变量只被初始化一次，静态变量其内存空间是只有在程序结束后才由系统释放

2）全局变量和静态全局变量的区别：

非静态全局变量的作用域是整个源程序，

静态全局变量的作用域只在定义该变量的源文件内有效，在同一源程序的其他源文件不能使用它。静态局部变量虽然只能在函数体内可以使用，但其生命周期是只有在程序结束后才由系统释放，所以静态局部变量可以通过引用返回

3）用const修饰的变量可分为const修饰全局变量和const修饰局部变量，只有const修饰的全局变量和字符串常量才放在全局区中。



栈区：由编译器自动分配和释放，存放函数的形参和局部变量等

堆区：由程序员手动分配和释放，若程序员不释放，程序结束时由操作系统回收

2.new运算符，在堆区创建数据

int \*p = new int(10); //new返回的是该数据类型的指针

delete p; //释放堆区的数据

int \*p = new int(10); //创建一个int类型变量在堆区，并赋值为10

int \*p = new int[10]; //创建10个整型数据的数组在堆区，这里的10代表数组有10个元素，其返回值返回的是数组的首地址，按类似栈上使用数组的方式区使用 p[3]/p[5]

释放堆区数组 delete[] p

3.引用

1）引用必须初始化，并且初始化后就不可以改变了，引用就是给变量起别名，不要返回局部变量的引用。

2）引用的本质在C++内部实现是一个指针常量,即：int &ref = a; <==> int \* const ref = &a

3）void ShowValue(const int& v) 常量引用是用来修饰形参，防止函数内部的误操作

4.指针常量和常量指针

指针常量：其本质上是一个常量，指针用来说明常量的类型 int \* const p = &a //指向的地址不可改变，当其内容可以修改

常量指针：其本质上是一个指针，常量表示指针指向的内容 const int \*p = &a //指向的地址可以改变，但其内容不可以改变

const在\*的左边，则指针指向的变量的值不可直接通过指针改变（可以通过其他途径改变）；

const在\*的右边，则指针的指向不可变。简记为“左定值，右定向”。

5.指针函数和函数指针

指针函数：其本质是一个函数，只是其返回值是一个指针 int \* func\_sum(int n)

函数指针：其本质是一个指针，只是其指向的是一个函数 int (\*p)(int, int); //函数指针的定义

typedef int(\*pSum)(int a,int b); //有typedef时，pSum是一个指针类型，他可以初始化实例；int(\*pSum)(int a,int b); //无typedef时，pSum仅仅是一个指针。

6.函数默认参数

声明和实现只能有一个默认参数

int func(int a, int b = 10);

int func(int a, int b = 10) //错误  
{

}

如果某个位置有默认参数，那么从这个位置往后，从左向右都必须有默认值

int func(int a, int b = 10, int c) //错误

7.占位参数

int func(int a, int)

占位参数也可以有默认参数

int func(int a, int = 10)

8.函数重载

1）同一作用域下

2）函数名称相同

3）函数参数类型不同，或者个数不同，或者顺序不同

int func(int a)

int func(double a)

使用函数重载的时候尽量不要使用默认参数，否则可能会发生二义性

8.sizeof()是一个操作符，其作用是返回一个对象或类型的字节数

9.

char \*strcpy(char \*dest, const char \*src)

void \*memcpy( void \*dest, const void \*src, size\_t count );

strcpy和memcpy的不同

1）复制的内容不同。strcpy只能复制字符串，而memcpy可以复制任意类型的内容。strcpy只用于字符串复制，并且还会复制字符串的结束符。memcpy对于复制的内容没有限制，用途更广。

2）复制的方法不同。strcpy不需要指定长度，遇到结束符’\0’才会结束，所以容易溢出。memcpy则是根据第三个参数决定复制的长度

3）用途不同。通常在复制字符串时用strcpy，在复制其他类型数据时一般用memcpy。

10.

三种访问权限

公有权限：public 成员类内可以访问，类外也可以访问

（类外访问就比如说s1.m\_name=”132”）

保护权限：protected 成员类内可以访问，类外不可以访问

继承方式：private 成员类内可以访问，类外不可以访问

继承方式包括 public（公有的）、private（私有的）和 protected（受保护的），此项是可选的，如果不写，那么默认为 private。不同的继承方式会影响基类成员在派生类中的访问权限。

（1）public继承方式

基类中所有 public 成员在派生类中为 public 属性；

基类中所有 protected 成员在派生类中为 protected 属性；

基类中所有 private 成员在派生类中不能使用。

（2）protected继承方式

基类中的所有 public 成员在派生类中为 protected 属性；

基类中的所有 protected 成员在派生类中为 protected 属性；

基类中的所有 private 成员在派生类中不能使用。

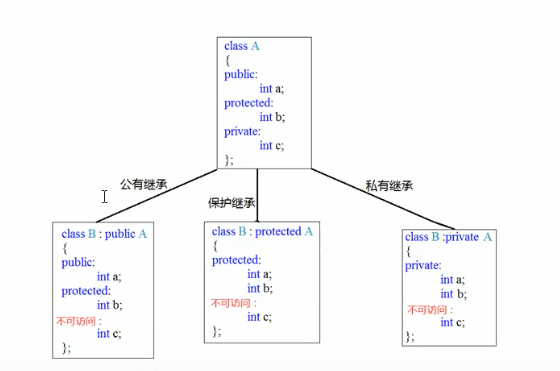
（3）private继承方式

基类中的所有 public 成员在派生类中均为 private 属性；

基类中的所有 protected 成员在派生类中均为 private 属性；

基类中的所有 private 成员在派生类中不能使用。

注意，我们这里说的是基类的 private 成员不能在派生类中使用，并没有说基类的 private 成员不能被继承。实际上，基类的 private 成员是能够被继承的，并且（成员变量）会占用派生类对象的内存，它只是在派生类中不可见，导致无法使用罢了，需要调用基类的公有和保护成员来访问。（父类中所有非静态成员属性都会被子类继承）



11.C++ 中struct和class的唯一区别就是默认访问权限不同

区别：

struct默认权限为公共

class默认权限为私有

12. #pragma once // 可以防止头文件重复包含

13.默认构造函数是无参构造函数，调用默认构造函数时，不要加（）

Person p1；//调用默认构造函数

Person p1(10); //调用有参构造函数

Person(const Person &p) //拷贝构造函数写法

Person(10) //匿名对象，不要利用拷贝构造函数初始化匿名对象

14.

拷贝构造函数调用时机：1.使用一个已经创建完毕的对象来初始化一个新对象2.值传递的方式给函数参数传值3.以值方式返回局部对象

15.在默认情况下，创建一个类，C++编译器至少给一个类添加四个函数1.默认构造函数2.析构函数3.拷贝构造函数,4.赋值运算符operator=，对属性进行值拷贝

如果有定义有参构造函数，编译器就不会提供无参构造函数，

如果有定义拷贝构造函数，编译器就不会提供其他构造函数

16.

浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作

如果属性有在堆区开辟的，既要自己提供拷贝构造函数也要提供赋值运算符重载函数，防止浅拷贝带来的问题，因为如果是浅拷贝，然后在析构函数里又会释放堆区的空间，就会导致堆区内容的重复释放。（赋值运算符重载函数比拷贝构造函数要多一步，先判断指针指向的数据是否为空，不为空要释放其内存空间。）

有在堆区分配空间的时候要在析构函数内释放其空间，

17.初始化列表

Person（int a, int b）：m\_A(a), m\_B (b) {}

18.当其他类对象作为本类成员，构造时候先构造其他类对象，再构造自身，析构的顺序和构造函数相反

19.静态成员变量1.所有对象共享同一份数据2，再编译阶段分配内存3类内声明，类外初始化

class Person

{

Public:

staic int m\_A;

}

int Person::m\_A = 100;

静态成员变量可以通过两种方式来访问

Person p；

p.m\_A = 10； //通过对象来进行访问

Person::m\_A = 10； //通过类名来进行访问

静态成员变量，它是所有对象所共享的，类内只是声明，类外实现定义。静态成员变量不属于对象，是属于类级别的。 当我们再次计算对象大小的时候，静态成员变量不纳入对象内存的。

父类的static变量和函数在派生类中依然可用，但是受访问性控制（比如，父类的private域中的就不可访问），而且对static变量来说，派生类和父类中的static变量是共用空间的，这点在利用static变量进行引用计数的时候要特别注意。

20.静态成员函数所有对象共享同一个函数，静态成员函数只能访问静态成员变量

也同静态成员变量一样可以通过两种方式来访问1. 通过对象来进行访问2. 通过类名来进行访问

21.C++中，类内的成员变量和成员函数是分开存储的，只有非静态成员变量才属于类的对象，每一个非静态成员函数也只会诞生一份函数实例，所以C++为了知道到底是哪一个对象调用自己，就通过this指针来实现

this指针指向被调用的成员函数所属的对象，this指针隐含在每一个非静态成员函数内。

this指针的本质是指针常量，指针的指向是不可以修改的

this指针的用途：

当形参和成员变量同名时，可以用this指针来区分

在类的非静态成员函数中返回对象本身，可以使用 return \*this；

Person &PersonAddAge（Person &p）

{

return \*this //返回对象本身的时候要注意返回值为引用

}

在 C++ 中，每一个对象都能通过 this 指针来访问自己的地址，为 this 的目的总是指向“这个”对象，所以 this 是一个常量指针

this只能在成员函数中使用。

全局函数，静态函数都不能使用this。

实际上，成员函数默认第一个参数为T\*const register this。（隐藏的）

22.用const修饰成员函数，其称为常函数，常函数内不可修改成员属性，在成员属性声明时加上mutable后，在常函数中依然可以修改。

在声明对象前加const称该对象为常对象，常对象只能调用常函数，在成员属性声明时加上mutable后，常对象依然可以修改

void showPerson() const //常函数

{

}

const Person p; // 常对象

23. 如果是友元就可以访问类中的私有成员。

友元有三种实现，1.全局函数做友元2.类做友元3.成员函数做友元。如果是友元就可以访问类中的私有成员

class Building

{

friend void goodGay(building \*building); //全局函数做友元 （放在类的最前面声明）

};

void goodGay(building \*building)  
{

}

///////////////////////////////////////////////////////////////////////////////////////////

class Building

{

friend class Goodgay; //类做友元 （放在类的最前面声明）

}；

class Goodgay

{

};

///////////////////////////////////////////////////////////////////////////////////////////

class Building

{

friend void Goodgay::visit(); //成员函数做友元（放在类的最前面声明）

prvate:

string m\_Building;

}；

class Goodgay

{

public:

Goodgay()

{

m\_building = new building;

}

public:

void visit()

{

cout << m\_building->m\_BedRoom >> endl;

}

private:

Building \*m\_building;

};

24.对于加号运算符，如果是内置的数据类型，编译器知道如何进行运算，而如果是自定义类型，编译器就不知道如何进行运算，这时候就可以使用运算符重载

//加号运算符重载 1成员函数重载加号运算符， 2.全局函数重载加号运算符

class Person

{

public:

Person operator+(Person &p) //成员函数重载加号运算符

{

m\_A += p.m\_A;

m\_B += p.m\_B;

}

int m\_A;

int m\_B;

}

Person operator+(Person &p1, Person &p2) //全局函数重载加号运算符

{

p1.m\_A += p2.m\_A;

p1.m\_B += p2.m\_B;

}

///////////////////////////////////////////////////////////////////////////////////////////

利用全局函数重载左移运算符(用成员函数无法实现)

ostream &operator<<(ostream &cout, Person &p)

{

cout << “m\_A” << p.m\_A << “m\_B” << p.m\_B << endl;

return cout;

}

///////////////////////////////////////////////////////////////////////////////////////////

class MyInteger

{

public:

MyInteger &operator++() //重置前置++运算符

{

m\_Num++;

return \*this

}

MyInteger operator++(int) //重置后置++运算符

{

MyInteger temp = \*this;

m\_Num++;

return temp;

}

private:

int m\_Num;

}

///////////////////////////////////////////////////////////////////////////////////////////

class Person

{

public:

Person(int age)

{

m\_age = new int(age)

}

~Person()

{

delete m\_age;

m\_age = NULL;

}

Person &operaor=(Person &p)

{

if(m\_age != NULL)

{

delete m\_age;

m\_age = NULL;

}

m\_age = new int(\*p.m\_age)

return \*this

}

private:

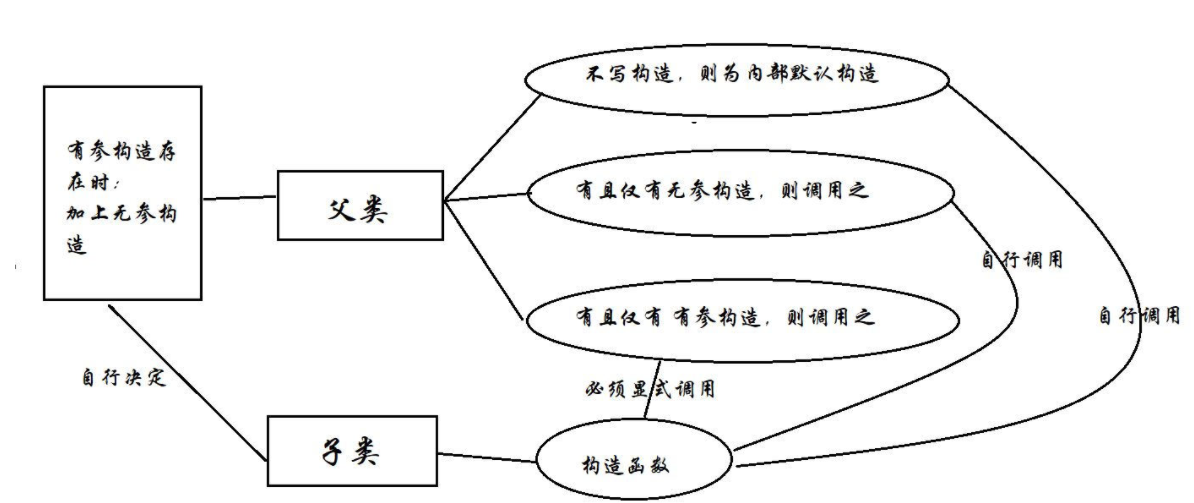
int \*m\_age；

}

///////////////////////////////////////////////////////////////////////////////////////////

重载（）运算符也称为仿函数

25.继承中构造和析构的顺序，先构造父类，再构造子类，析构顺序和构造顺序相反，子类不能继承父类的构造函数



如果父类有参构造需要在子类构造函数中显示调用

Carton(double lv, double wv, double hv, const string desc) : Box(lv, wv, hv ）, material（ desc ）

子类会自动调用父类的析构函数，如果是new出来的对象，必须delete指针才会调用其析构函数

26.继承中同名成员属性处理方式，通过子类对象放问父类中的同名成员需要加作用域

s.m\_A //访问的是子类中的同名成员属性

s.Base::m\_A//访问的是父类中的同名成员属性

继承中同名成员函数处理方式，（如果子类中出现和父类同名的成员函数，子类的同名函数会隐藏调父类中所有的同名成员函数，如果想调用父类的同名成员函数需要加作用域）

s.func()//调用的是子类中的同名成员函数

s.Base::func()//调用的是父类中的同名成员函数

继承中同名静态成员的处理方式于非静态成员一致，唯一要注意的是对于静态成员的第二种访问方式（通过类名进行访问）

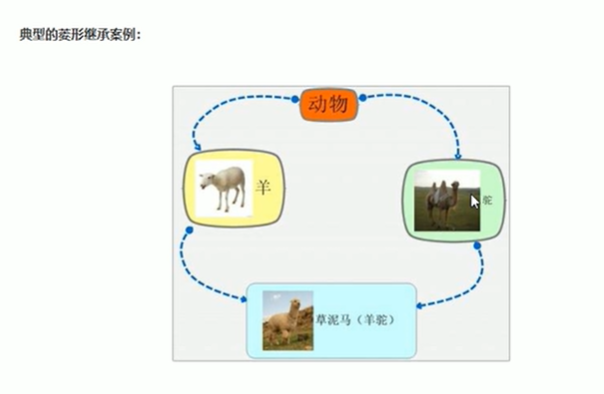
Son::func(); //访问的是子类中的同名静态成员函数

Son::Base::func();//访问父类中同名的静态成员函数

27.多继承语法

Class son :public base1, public base2

28.菱形继承（两个子类继承同一个父类，而又有子类同时继承这两个子类。），当出现菱形继承中，如果有两个父类拥有相同数据，需要用作用域加以区分，而菱形继承还会导致数据有两份的情况，所以要利用虚继承来解决此类问题（Class son :virtual public base1）,使用虚继承后虚基类的数据就只有一份了



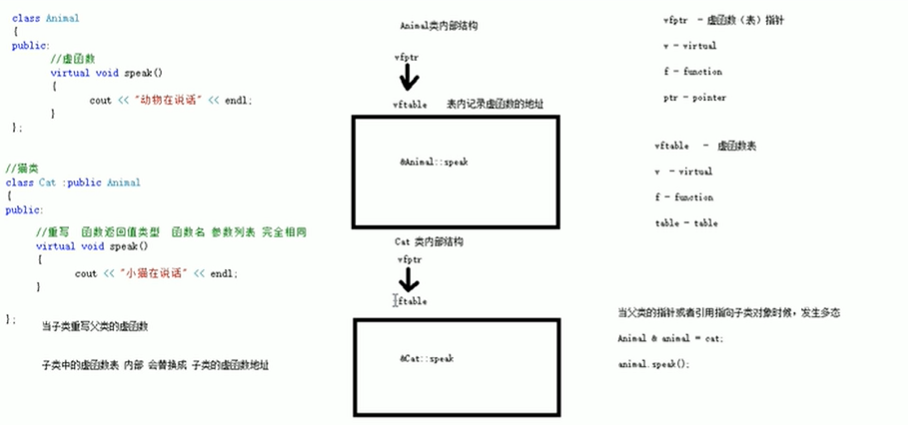
29.多态分为两类：1.静态多态：函数重载和运算符重载属于静态多态2.动态多态：派生类和虚函数属于动态多态。

静态多态和动态多态的区别：静态多态的函数地址早绑定，编译阶段确定函数地址，动态多态函数地址晚绑定，运行阶段确定函数地址。

动态多态需要满足两个条件，1.有继承关系。2.子类重写父类的虚函数

动态多态的使用：父类的指针或者引用指向子类对象

虚函数是C++实现动态多态的一种方式，可以使得父类指针指向子类对象，调用虚函数时是调用子类的同名成员函数。



30.纯虚函数：virtual void func() = 0;当类中有了纯虚函数，这个类也就称为抽象类，抽象类无法实例化对象，当子类继承抽象类时，子类必须重写抽象类的纯虚函数，否则子类也属于抽象类

纯虚函数是在声明虚函数时被“初始化”为0的函数。声明纯虚函数的一般形式是

virtual 函数类型 函数名 (参数表列) =0;

注意: ①纯虚函数没有函数体；②最后面的“=0”并不表示函数返回值为0，它只起形式上的作用，告诉编译系统“这是纯虚函数”; ③这是一个声明语句，最后应有分号。

纯虚函数只有函数的名字而不具备函数的功能，不能被调用。它只是通知编译系统: “在这里声明一个虚函数，留待派生类中定义”。在派生类中对此函数提供定义后，它才能具备函数的功能，可被调用。

31.多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码，解决方法：将父类中的析构函数改为虚析构或者纯虚析构。  
纯虚析构既需要声明也需要实现

class Aninal

{

public:

Animal(){}

virtural ~Animal() = 0; //纯虚析构声明

};

Animal::~Animal() //纯虚析构实现

{

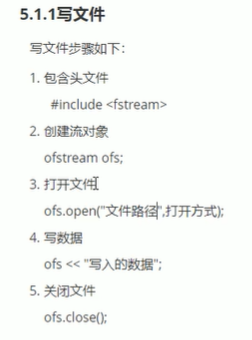
}

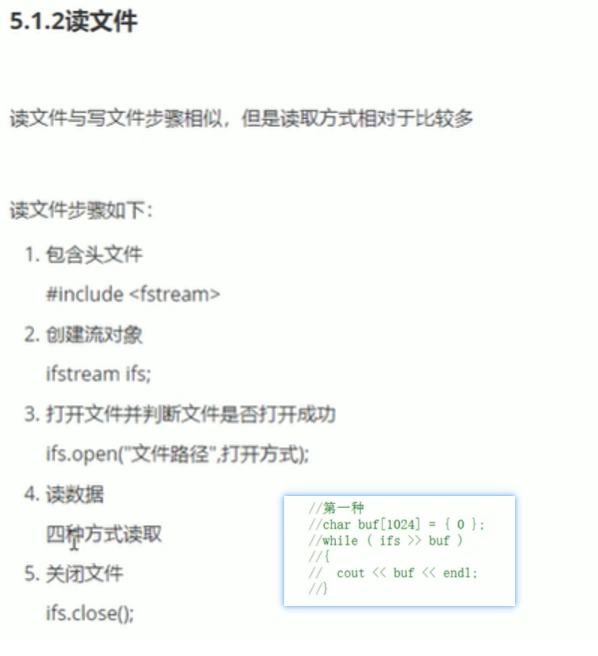
32.文件操作

C++对文件操作需要包含头文件<fstream>

文件类型分为两种：1.文本文件，文件以文本的ASCII码形式存储在计算机中2.二进制文件：文件以文本的二进制形式存储在计算机中

该头文件提供操作文件的三种类：1.ofstream：写操作 2.ifstream：读操作 3.fstream：读写操作







EOF文件尾部标志

二进制方式写文件主要利用流对象调用成员函数write

函数原型： ostream &write(const char \*buffer, int len);

buffer：指向内存中的一段存储空间，len：是写入的字节数

二进制方式读文件主要利用流对象调用成员函数read

函数原型： istream &read (const char \*buffer, int len);

buffer：指向内存中的一段存储空间，len：是读出的字节数

打开文件后可以调用流对象的成员函数is\_open()来判断文件是否打开成功

函数原型：bool is\_open();

成功返回true

失败返回false

使用举例：

if(! ifs.is\_open())

{

cout << “文件打开失败” << endl;

return;

}

33.C++排序函数,（只适用于排序非自定义类型的数据，且容器类型要是支持随机访问的迭代器）

sort(v.begin(),v.end()); //从小到大排序

sort(v.rbegin(),v.rend()); //从大到小排序

34.



list的iterator是双向的，只支持++、--，不支持iter+1之类的操作，如果要移动多个元素应该用next：  
#include <iterator>  
using std::next;  
iter = next(iter, 2);

for (auto iter1 = m\_datalist.begin(); iter1 != m\_datalist.end(); iter1++)

{

for (auto iter2 = next(iter1, 1); iter2 != m\_datalist.end(); iter2++)

{

if ((\*iter1)->GetNumber() > (\*iter2)->GetNumber())

{

iter\_swap(iter1,iter2);

}

}

}

vector的iterator 才支持 iter + 1

for (auto iter1 = m\_datalist.begin(); iter1 != m\_datalist.end(); iter1++)

{

for (auto iter2 = iter + 1; iter2 != m\_datalist.end(); iter2++)

{

if ((\*iter1)->GetNumber() > (\*iter2)->GetNumber())

{

iter\_swap(iter1,iter2);

}

}

}

std::list<AbstractStaff \*> m\_datalist; 如果容器list存的数据类型是指针，在析构函数除了要m\_datalist.clear()还要清空指针指向的内存空间（使用delete）

35.C++的泛型编程思想主要通过模板来实现，C++模板提供两种机制1.函数模板2.类模板

typename 和class关键字都一样，使用哪一个都行

函数模板语法：

template<typename T>

void mySwap(T &a, T &b)

{

T temp = a;

b = a;

a = temp;

}

有两种方式使用函数模板1.自动类型推导2.显示指定类型mySwap<int>(a, b)

通过空模板参数列表，可以强制调用函数模板myPrint<>(a, b);

模板具体化：

template<typename T>

template<> void mySwap(person &a, peson & b)

类模板语法：

template<class NameType, class AgeType>

class Person

{

public:

NameType m\_Name;

AgeType m\_Age;

}

Person<string, int>p1; //使用类模板类,类模板只能用显示指定类型，不能自动类型推导

类模板的成员函数在调用的时候才会创建。

通过类模板创建的对象，可以有三种方式向函数中进行传参，这边只记录一种

指定传入类型

void PrintPerson1(Person<string, int>&p)

{

}

void test01()

{

Person<string, int>p(“123”, 100)

printPerson1(p);

}

36.类模板于继承

当子类继承的父类是一个类模板时，子类在声明的时候，需要指出父类中T的类型

template<class T>

class Base

{

T m;

}

class Son :public Base<int>

{

}

37.类模板成员函数类外实现

template<class T1, class T2>

class Person

{

public:

void setPerson(T1 a, T2 b);

T1 m\_Name;

T2 m\_Age;

}

template<class T1, class T2>

void Person<T1, T2>::setPerson(T1 a, T2 b)

{

this->m\_Name = a;

this->m\_Age = b;

}

类模板分文件编写：类模板的成员函数创建时机是在调用阶段，导致分文件编写时链接不到，解决方法，将声明和实现写在同一个文件中，并更改后缀名为hpp，hpp是约定的名称（写hpp代表写的是一个类模板）

38.STL(标准库模板)，从广义上分为容器、算法、迭代器

STL六大组件分别是：容器、算法、迭代器、仿函数、适配器（配接器）、空间配置器

39.迭代器

vector<int> v;

vector<int>::iterator itBegin = v.begin();

vector<int>::const\_iterator itBegin = v.begin() //不可修改迭代器指向元素的值

40.string本质是一个类，类的内部封装了char\* ，管理这个字符串，是一个char\*型的容器

string字符串拼接有两种方式：1.使用+=。2.使用成员函数append（）

string str = “a”;

str += “bcd”; //使用+=拼接

string str = “a”;

str.append( “bcd”); //使用append（）拼接

string字符串查找和替换

查找：int find(const string &str, int pos = 0)const； 查找str第一次出现位置，从pos开始

找到返回下标值，没找到返回-1；(rfind从右往左)

string str = “abcdefg”;

int pos = str.find(“df”);

替换：string &replace(int pos, int n, const char \*s); 替换从pos开始的n个字符为字符串s。

string str = “abcdefg”;

str.replace(1, 3, “fad”);

string字符串比较

int compare(const string &s) const字符串的比较是按字符的ASCII码进行比较的，相等返回0，第一个字符串大于第二个字符串返回1，小于返回 -1

string str1 = “hello”;

string str2 = “hello”;

str1.compare(str2);

string 字符串存取

string str = “hello”;

str[2] = f;

string字符串插入和删除

插入：

string &insert(int pos, cons char \*s); //插入字符串

string str1 = “aas”

str1.insert(1,”sss”);

删除：

string &erase(int pos, int n = npos)//删除从Pos开始的n个字符

string str1 = “adfafds”;

str1.erase(1, 3);

string从字符串中获取想要的子串

string substr(int pos = 0, int n = npos) const; //返回由pos开始的n个字符组成的字符串

string str = “abcde”;

string substr = srt.substr(1, 3);

41.vector容器，数据结构与数组相似，也称为单端数组，可动态扩展，动态扩展不是在原空间之后续接新空间，而是找更大的内存空间，将原数据拷贝进新空间，释放原空间

vector的迭代器是支持随机访问的迭代器，可以+n

函数原型：

empty(); //判断容器是否为空，为真代表容器为空

capacity();//容器的容量

size(); //返回容器中的元素个数

resize(int num，elem);//重新指定容器的长度num，若容器变长，则以elem值填充新位置，如果容器变短，则末尾超出容器长度元素被删除。

push\_back(ele);//尾部插入元素ele

pop\_back();//删除最后一个元素

insert(const\_iterator pos, ele); //迭代器指向位置pos插入元素ele

insert(const\_ierator pos, int count, ele)//迭代器指向位置pos插入count个元素ele

erase(const\_iterator pos);//删除迭代器指向的元素，返回下一个元素的迭代器

erase(start, end);//删除迭代器从start到end之间的元素，返回下一个元素的迭代器

clear();//删除容器中所有元素

operator[];//重载了[]所以可以通过[]返回索引idx所指向的数据

front();//返回容器中的第一个元素

back();//返回容器中最后一个元素

swap(vec); //将vec与本身的元素互换，巧用swap可以收缩内存空间

例子：vector<int>v;

vector<int>(v).swap(v);

reserve(int len);//容器预留len个元素长度，预留位置不初始化，元素不可访问

42.deque容器，称为双端数组，可以对头端进行插入删除操作

deque与vector的区别

vector对于头部的插入删除效率低，deque对头部的插入删除效率高，但vector访问元素的速度会比deque快

deque的迭代器是支持随机访问的迭代器，可以+n

deque容器使用的函数跟vector容器差不多，只有些许区别，以下只列出有区别部分

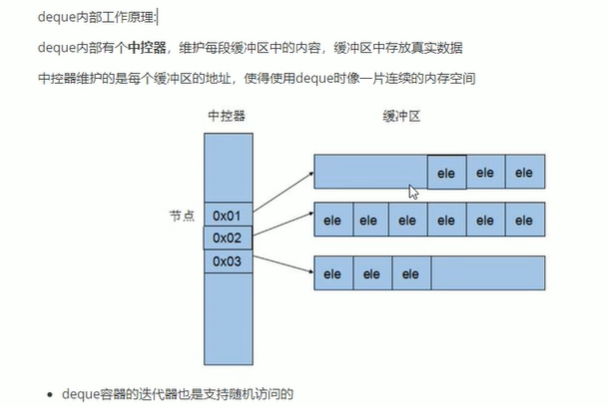
相比于vector没有的函数：

deque容器没有capacity()函数，因为其没有容量的概念

相比于vector新增的函数：

push\_front();//在容器的头部插入一个数据

pop\_front();//删除容器第一个数据



43.stack容器是一种先进后出的数据结构，它只有一个出口，栈不允许有遍历的行为

数据存取

push(elem); //向栈顶添加元素

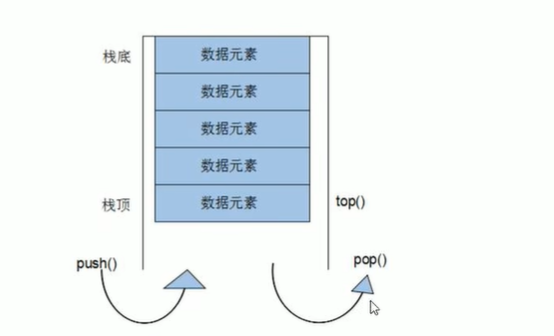
pop();//从栈顶移除第一个元素

top();//返回栈顶元素

大小操作

empty();//判断堆栈是否为空

size();//返回栈的大小



44.queue容器是一种先进先出的数据结构，它有两个口，一端只允许新增数据，另一端只允许移除数据，只有队头队尾才可以被外界访问。

数据存取：

push(elem);//往队尾添加元素

pop();//从对头移除第一个元素

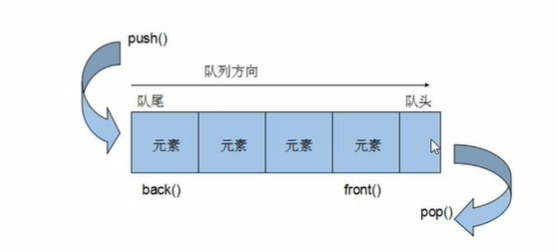
back();//返回最后一个元素

front();返回第一个元素

大小操作：

empty();//判断堆栈是否为空

size();//返回栈的大小



45.链表容器，是一种物理存储单元上非连续的存储结构，链表是由一系列的结点组成的。

结点由数据域和指针域所组成，是一个双向循环列表

1.std::list定义对象

list<A> listname; 创建空list

list<A> listname(size);创建含有5个元素的list

list<A> listname(size,value); 创建含有3个元素的list，其值为value

list<A> listname(elselist);使用elselist初始化listname

list<A> listname(elselist.begin(),elselist.end());同上一个

2.std::list添加元素

list1.push\_front(const T& x); // 头部添加

list1.push\_back(const T& x); // 尾部添加

3.std::list删除元素

list1.pop\_front(); // 头部删除

list1.pop\_back(); // 尾部删除

4.std::list容器容量

list1.size() const; // 返回元素个数

list1.max\_size() const; // 返回list对象最大允许容量

resize(int num，elem);//重新指定容器的长度num，若容器变长，则以elem值填充新位置，如果容器变短，则末尾超出容器长度元素被删除。

list1.empty()//检测列表容器是否为空，如果列表容器为空，则返回true，否则返回false

5.插入

insert(const\_iterator pos, ele); //迭代器指向位置pos插入元素ele

insert(const\_ierator pos, int count, ele)//迭代器指向位置pos插入count个元素ele

6.std::list迭代器

list1.begin() // 返回指向容器中第一个元素的双向迭代器。

list1.end() // 返回指向容器中最后一个元素所在位置的下一个位置的双向迭代器。

list1.rbegin() // 返回指向最后一个元素的反向双向迭代器。

list1.rend() // 返回指向第一个元素所在位置前一个位置的反向双向迭代器。

list1.cbegin() // 和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

list1.cend() // 和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

list1.crbegin() // 和 rbegin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

list1.crend() // 和 rend() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。

swap(L); //将L与本身的元素互换

reverse();//元素反转

listname.clear() // clear()函数用于删除列表容器的所有元素，从而使其大小为0。

list1.front() //返回列表容器中第一个元素的引用

list1.back () //返回列表容器中最后一个元素的引用

erase( pos);//删除迭代器指向的元素，返回下一个迭代器所指向的位置

remove(elem);//删除容器中所有与elem值匹配的元素

使用方法：对于删除操作要注意使用这样for循环的写法

for(iter = iaddress.lt\_stPerson.begin(); iter != iaddress.lt\_stPerson.end();)

{

if(1)

{

iter = iter.erase(iter);

}

else

iter++;

}

iter\_swap(iter1,iter2) //用于交换两个迭代器所指向的值

所有不支持随机访问迭代器的容器，不可以使用标准算法（即头文件algorithm），所以list不能用sort，不支持随机访问迭代器的容器，内部会提供对应一些算法。

就比如list不能用sort(l.begin(), l.end()),只能用它内部提供的l.sort()

l.sort();//默认升序排列，从小到大

实现从大到小排序需要两步：

bool myCompare(int v1, int v2)

{

return v1 > v2;

}

l.sort(myCompare);

vector和list的选用：

（1）如果需要高效的随机存取，而不在乎插入和删除的效率，使用vector；  
（2）如果需要大量高效的删除插入，而不在乎存取时间，则使用list；

46.set/multiset容器，两个容器的头文件都是<set>所有元素都会在插入时自动被排序，底层结构是通过二叉树来实现的

set容器和multiset容器唯一的区别就是set不允许容器中由重复的元素，multiset容器允许容器中由重复的元素

empty(); //判断容器是否为空，为真代表容器为空

size(); //返回容器中的元素个数

swap(s); //将两个集合容器进行交换

插入数据只有insert方式

insert(elem);//在容器中插入元素

erase( pos);//删除迭代器指向的元素，返回下一个迭代器所指向的位置

erase(start, end);//删除迭代器从start到end之间的元素，返回下一个迭代器所指向的位置

erase(elem);//删除容器中值为elem的元素

clear();//删除容器中所有元素，类似remove()操作

find(key);//查找key是否存在，若存在，返回该键的元素迭代器，若不存在，返回set.end()

count(key);//统计key元素的个数

修改set容器的排序规则(从大到小排)，利用仿函数

class Mycompare

{

public:

bool operator()(int v1, int v2)

{

return v1 > v2;

}

}

set<int, MyCompare>s2; //set容器存放内置数据类型排序,在初始化的时候就要修改排序规则，后面在插入的时候就是按从大到小排序

/////////////////////////////////////////////////////////////////////////////////////

自定义数据类型都要指定排序规则

class Person

{

public:

string m\_name;

int m\_age;

}

class Mycompare

{

public:

bool operator()(const Person &v1,const Person &v2)

{

return v1.m\_age > v2.m\_age

}

}

set<Person, MyCompare>s2; //set容器存放内置数据类型排序

47.pair对组创建，成对出现的数据，利用队组可以返回两个数据

创建方式

pair<type, type> p (value1, value2);

例子：

pair<string, int>p(“Tom”, 20);

cout << “姓名： ” << p.first << “年龄： ” << p.second << endl;

47.map容器/multimap容器，（map容器和multimap容器的区别是map容器不允许有重复key值元素，multimap允许容器中有重复key值元素）

map中所有容器都是pair，

pair中第一个元素为key（键值），起到索引作用，第二个元素为value（实值）

所有元素都会根据元素的键值自动排序，

本质map/multimap属于关联式容器，底层结构是用二叉树实现的

map<int, int>m; //创建map容器

m.insert(pair<int, int>(1, 10)) //插入数据

m[4] = 20//可以利用[]来访问元素,[]里填写的是键值（但是不建议用[]插入元素）

empty(); //判断容器是否为空，为真代表容器为空

size(); //返回容器中的元素个数

swap(m); //将两个集合容器进行交换

insert(elem);//在容器中插入元素

erase(pos);//删除迭代器指向的元素，返回下一个元素的迭代器

erase(start, end);//删除区间（start，end）的所有元素，返回下一个元素的迭代器

erase(key);//删除容器中值为key的元素

clear();//清除所有元素

find(key); //查找key是否存在，若存在，返回该键的元素的迭代器，若不存在，返回set.end();

count(key);//统计key的元素个数

map容器默认排序规则为按照key值从小到大，通过使用仿函数改变排序规则,按照从大到小排序

class Mycompare

{

public:

bool operator()(int p1, int p2)

{

return p1 > p2;

}

}

map<int, int, Mycompare> m; //按照从大到小排序，自定义类型的数据也类似可参考set容器

48.随机数

包含<ctime>头文件

srand((unsigned int)time(NULL)); //种随机数种子

rand()%max //这个表达式会生成[0..max-1]之间的随机数

49.函数对象

重载函数调用操作符的类，其对象常称为函数对象

函数对象使用重载的（）时，行为类似函数调用，也叫做仿函数

本质：函数对象（仿函数）是一个类，不是一个函数

特点：

函数对象在使用是，可以像普通函数那样调用，可以有参数，返回值

函数对象超出普通函数的概念，函数对象可以有自己的状态

函数对象可以作为参数传递

50.谓词

返回bool类型的仿函数称为谓词

如果operator()接受一个参数，那么叫做一元谓词

如果operator()接受两个参数，那么叫做二元谓词

例子：

class GreaterFive

{

bool operator()(int val)

{

return val > 5;

}

};

vector<int> v;

//find\_if() 是algorithm头文件下的函数，返回值为查找到的迭代器位置

// GreaterFive()匿名函数对象

vector<int>::iterator iter = find\_if(v.begin(), v.end(), GreaterFive());

51.内建函数对象

STL内建了一些函数对象

使用内建函数对象需要包含头文件#include<functional>

52.STL常用算法

算法主要是由头文件<algorithm>、<functional>、<numeric>组成

<algorithm>是所有STL头文件，范围涉及到比较、交换、查找、遍历操作、复制、修改等等

<numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数

<functional>定义了一些模板类，用以声明函数对象

for\_each(iterator beg, iterator end, \_func)//实现遍历容器

beg：开始迭代器

end：结束迭代器

\_func：函数或函数对象

for\_each(v.begin(); v.end(); print) //第三个参数传递函数

for\_each(v.begin(); v.end(); print()) //第三个参数传递函数对象

transform (iterator beg1, iterator end1, iterator beg2, \_func)//搬运容器到另一个容器种

beg1：源容器开始迭代器

end1：源容器结束迭代器

beg2：目标容器开始迭代器

\_func：函数或函数对象

注意点：目标容器要提前开辟空间

v2.resize(v1.size());

find(iterator beg, iterator end, value);//按值查找元素，找到返回指定元素的迭代器，找不到返回结束迭代器位置

beg：开始迭代器

end：结束迭代器

value：查找的元素

注意点：如果要查找自定义数据类型，类中需要重载==，底层find才知道如何对比person数据类型

bool operator==(const Person &p)

{

if(this->m\_name == p.m\_name && this->m\_age = p.m\_age)

{

return true;

}

else

return false;

}

find\_if(iterator beg, iterator end, \_Perd)// 按条件查找元素，找到返回指定元素的迭代器，找不到返回结束迭代器位置

beg：开始迭代器

end：结束迭代器

\_Perd：谓词

adjacent\_find（iterator beg, iterator end）//查找相邻重复元素，返回相邻重复元素的第一个位置的迭代器

beg：开始迭代器

end：结束迭代器

bool binary\_search(iterator beg, iterator end, value);//通过二分查找法，查找指定元素是否存在，查到返回true，否则返回false

beg：开始迭代器

end：结束迭代器

value：查找的元素

注意点：查找的容器元素必须是有序序列

int count(iterator beg, iterator end, value);//统计元素出现次数，返回值为出现次数

beg：开始迭代器

end：结束迭代器

value：统计的元素

注意点：统计自定义数据类型，类中需要重载==运算符

bool operator==(const Person &p)

{

if(this->m\_age == p.m\_age)

{

return true;

}

}

int count\_if(iterator beg, iterator end, \_Pred)//按条件统计元素个数，返回值为出现次数

beg：开始迭代器

end：结束迭代器

\_Pred：谓词

sort(iterator beg, iterator end, \_Pred)//对容器内元素进行排序，第三个参数可填可不填

beg：开始迭代器

end：结束迭代器

\_Pred：谓词

random\_shuffle(iterator beg, iterator end)指定范围内的元素随机调整次序

beg：开始迭代器

end：结束迭代器

注意点：使用时还要加上随机数种子

merge(iterator beg1, iterator end1, iterator beg2, iterator end3, iterator dest)//容器元素合并，并存储到另一容器中

beg1：容器1开始迭代器

end1：容器1结束迭代器

beg2：容器2开始迭代器

end2：容器2结束迭代器

dest：目标容器开始迭代器

注意点：两个容器必须是有序的，且需要提前给目标容器分配内存（ vtarget.resize(v1.size()+v2.size()) ）

reverse(iterator beg, iterator end);//反转指定范围内的元素

beg：开始迭代器

end：结束迭代器

copy(iterator beg, iterator end, iterator dest);//容器内指定范围的元素拷贝到另一容器中

beg：开始迭代器

end：结束迭代器

dest：目标容器开始迭代器

注意点：目标容器需要提前分配内存，（ vtarget.resize(v.size()) ）

replace(iterator beg, iterator end, oldvalue, newvalue)//将区间内旧元素替换为新元素

beg：开始迭代器

end：结束迭代器

oldvalue：旧元素

newvalue：新元素

replace(iterator beg, iterator end, \_pred, newvalue)//按条件替换元素，满足条件的替换为指定元素

beg：开始迭代器

end：结束迭代器

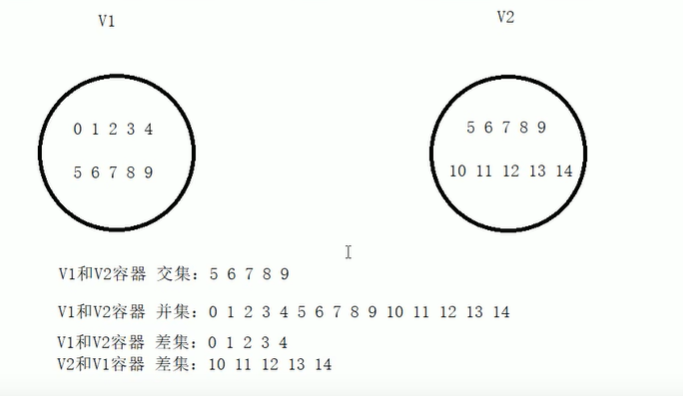
\_pred：谓词

newvalue：替换的新元素

swap(container c1, container c2);//互换两个容器的元素

c1：容器1

c2：容器2



set\_intersection(iterator beg1, iterator end1, iterator begin2, iterator end2, iterator dest)//求两个集合的交集，返回值为交集中最后一个元素的迭代器位置

beg1：容器1开始迭代器

end1：容器1结束迭代器

beg2：容器2开始迭代器

end2：容器2结束迭代器

dest：目标容器开始迭代器

注意点：两个集合必须是有序序列，目标容器需要提前开辟空间，开辟的空间可以为两个容器中小的那一个容器大小

set\_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)//求两个集合的并集，返回值为并集中最后一个元素的迭代器位置

beg1：容器1开始迭代器

end1：容器1结束迭代器

beg2：容器2开始迭代器

end2：容器2结束迭代器

dest：目标容器开始迭代器

注意点：两个集合必须是有序序列，目标容器需要提前开辟空间，开辟的空间为两个容器的大小之和

set\_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)//求两个集合的差集，返回值为差集中最后一个元素的迭代器位置（差集：不是交集的部分）

beg1：容器1开始迭代器

end1：容器1结束迭代器

beg2：容器2开始迭代器

end2：容器2结束迭代器

dest：目标容器开始迭代器

注意点：两个集合必须是有序序列，目标容器需要提前开辟空间，开辟的空间可以为两个容器中大 的那一个容器大小

53.常用算术生成算法，算术生成算法属于小型算法，使用时包含的头文件为<numeric>

accumulate(iterator beg, iterator end, value)//计算容器元素累加总和

beg：开始迭代器

end：结束迭代器

value：起始值

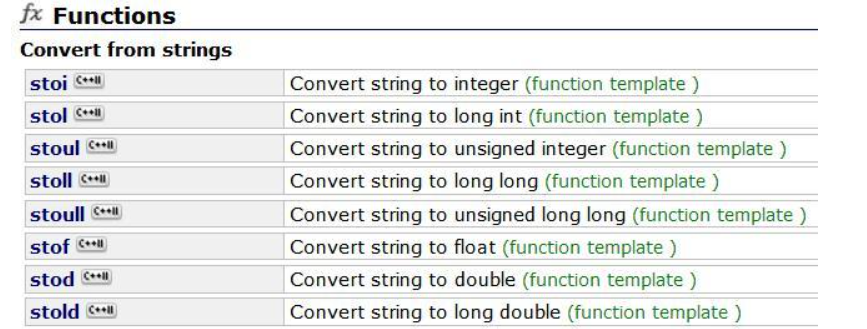
fill(iterator beg, iterator end, value)//向容器中填充元素

beg：开始迭代器

end：结束迭代器

value：填充的值

54.string转换为int、float



55.

min\_element(begin, end);

参数

begin : 序列起始地址（迭代器）

end : 序列结束地址（迭代器）

返回值 : 序列中最小元素地址（迭代器）

max\_element(begin, end);

参数

begin : 序列起始地址（迭代器）

end : 序列结束地址（迭代器）

返回值 : 序列中最大元素地址（迭代器）

distance( begin, end)

参数

begin : 序列起始地址（迭代器）

end : 序列结束地址（迭代器）

返回值 :返回两个迭代器范围内包含的元素个数

advance (InputIterator& it, Distance n)

参数

it：迭代器

n：为一个常数

函数作用：用于将迭代器前进（或者后退）指定长度的距离

accumulate(vec.begin() , vec.end() , 42);

accumulate带有三个形参：头两个形参指定要累加的元素范围，第三个形参则是累加的初值。

56. static\_cast< 想要的类型> (原数据)， 是C++中的关键字。相当于C语言的强制类型转换

57. string是对象，不是一段[内存](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98&spm=1001.2101.3001.7020)，不适合放在结构体中。

对于string、列表、容器是不能使用memset的

58.单例模式：程序在设计上必须保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

饿汉模式：指全局的单例实例在类加载时就被构建

class Singleton

{

public:

static Singleton &getObject()//给外界的唯一接口

{

return instance;

}

void set(int t)

{

data = t;

}

void print()

{

cout << "data == " << data << endl;

}

//其他成员函数

private:

Singleton(){};

~Singleton(){};

Singleton(const Singleton &rhs); //拷贝构造函数

Singleton &operator=(const Singleton &rhs); //拷贝赋值运算符

Singleton(const Singleton &&rhs); //移动构造函数

Singleton &operator=(const Singleton &&rhs); //移动赋值运算符

static Singleton instance;

int data;

};

Singleton Singleton::instance; //注意需要在类外定义

int main(void)

{

Singleton::getObject().set(666);

Singleton::getObject().print();

return 0;

}

懒汉模式：指全局的单例实例在第一次被使用时构建。

class Singleton

{

public:

static Singleton &getObject() //给外界的唯一接口

{

static Singleton instance; //staic 离开函数仍然存在

return instance;

}

void set(int t)

{

data = t;

}

void print()

{

cout << "data == " << data << endl;

}

//其他成员函数

private:

Singleton(){};

~Singleton(){};

Singleton(const Singleton &rhs); //拷贝构造函数

Singleton &operator=(const Singleton &rhs); //拷贝赋值运算符

Singleton(const Singleton &&rhs); //移动构造函数

Singleton &operator=(const Singleton &&rhs); //移动赋值运算符

int data;

};

int main(void)

{

Singleton::getObject().set(445555454);

Singleton::getObject().print();

return 0;

}

59. 静态函数:使某个函数只在一个源文件中有效，不能被其他源文件所用。

60.C++在构造函数中调用虚函数，是静态绑定不是动态绑定。

61.面向对象设计原则：

（1）依赖倒置原则（DIP）

高层模块（稳定）不应该依赖于低层模块（变化）二者都应该依赖于抽象（稳定）

抽象（稳定）不应该依赖于实现细节（变化）实现细节应该依赖于抽象（稳定）。

（2）开放封闭原则（OCP）

对扩展开放，对更改封闭。

类模型应该是可扩展的，但是不可修改的。

（3）单一职责原则（SRP）

一个类应该仅有一个引起它变化的原因。

变化的方向隐含着类的责任。

（4）Liskov替换原则（LSP）

子类必须能够替换他们的基类（IS-A）。

继承表达类型抽象。

（5）接口隔离原则（ISP）

不应该强迫客户程序依赖它们不用的方法。

接口应该小而完备。

（6）优先使用对象组合，而不是类继承

类继承通常为“白箱复用”，对象组合通常为“黑箱复用”。

继承在某种程度上破坏了封装性，子类父类耦合度高。

而对象组合则只要求被组合的对象具有良好定义的接口，耦合度低。

（7）封装变化点

使用封装来创建对象之间的分界层，让设计者可以在分界层的一侧进行修改，而不会对另一侧产生不良的影响，从而实行层次间的松耦合。

（8）针对接口编程，而不是针对实现编程

不将变量类型声明为某个特定的具体类，而是声明为某一个接口。

客户程序无需获知对象的具体类型，只需要知道对象所具有的接口。

减少系统中各部分的依赖关系，从而实现“高内聚、松耦合”的类型设计方案.

61.一般不推荐多继承，但推荐一个是主的继承类，其他都是接口或者是抽象基类

62.C++中循环语句for(auto elem:range)

1） for(auto elem : range)

创建range的拷贝，遍历时无法修改range中的元素。

2） for(auto& elem : range)

不创建range的拷贝，可以直接修改range中的元素，但一般用以下这种形式。

3） for(const auto & elem : range)

不创建range的拷贝，只读range中的元素。

63.

模式划分：

“组件协作”模式：模板方法、策略模式、观察者模式

模板方法：定义一个操作中的算法的骨架（稳定），而将一些步骤延迟（变化）到子类中。模板方法使得子类可以不改变（复用）一个算法的结构即可重定义（override重写）该算法的某些特定步骤。

动机：在软件构建过程中，对于某一项任务，它常常有稳定的整体操作结构，但各个子步骤却有很多改变的需求，或者由于固有的原因（比如框架与应用之间的关系）而无法和任务的整体结构同时实现。

策略模式：定义一系列算法，把它们一个个封装起来，并且使他们可以相互替换（变化）。该模式使的算法可独立于使用他的客户程序（稳定）而变化（扩展、子类化）---有需要使用if-else或switch-case语句时可以考虑是否用策略模式，策略模式是应对变化的，如果确定完全不变不必考虑策略模式

动机：在软件构建过程中，某些对象使用的算法可能多种多样，经常改变，如果将这些算法都编码到对象中，将会使对象变得异常复杂，而且有时候支持不使用的算法也是一种性能负担。

观察者模式：定义对象间的一种一对多（变化）的依赖关系，以便当一个对象（subject）的状态发生改变时，所有依赖于它的对象都得到通知并自动更新。

动机：在软件构建过程中，我们需要为某些对象建立一种“通知依赖关系”---一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知。如果这样的依赖关系过于紧密，将使软件不能很好地抵御变化。

“单一职责”模式：装饰模式、桥模式

装饰模式：动态（组合）地给一个对象增加一些额外的职责。就增加功能而言，装饰模式比生成子类（继承）更为灵活（消除重复代码&减少子类个数）。

动机：在某些情况下我们可能会“过度地使用继承来扩展对象的功能“，由于继承为类型引入静态性质，使得这种扩展方式缺乏灵活性；并且随着子类的增多（扩展功能的增多），各种子类的组合（扩展功能的组合）会导致更多子类的膨胀。

桥模式：将抽象部分（业务功能）与实现部分（平台实现）分离，使它们都可以独立地变化。

动机：由于某些类型固有的实现逻辑，使得它们具有两个变化的维度，乃至多个维度的变化。

“对象创建“模式：工厂方法、抽象工厂、原型模式、构建器

工厂方法：定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使得一个类的实例化延迟（目的：解耦、手段：虚函数）到子类。

动机：在软件系统中，经常面临着创建对象的工作；由于需求的变化，需要创建的对象的具体类型经常变化。

抽象工厂：提供一个接口，让该接口负责创建一系列“相关或者相互依赖的对象“，无需指定它们具体的类。

动机：在软件系统中，经常面临着“一系列相互依赖的对象“的创建工作；同时，由于需求变化，往往存在更多系列对象的创建工作。

原型模式：

使用原型实例指定创建对象的种类，然后通过拷贝这些原型来创建新的对象。

动机：在软件系统中，经常面临着“某些结构复杂的对象“的创建工作；由于需求的变化，这些对象经常面临着剧烈的变化，但是它们却拥有比较稳定一致的接口。

构建器：将一个复杂对象的构建与其表示相分离，使得同样的构建过程（稳定）可以创建不同的表示（变化）。

动机：在软件系统中，有时候面临着“一个复杂对象“的创建工作，其通常由各个部分的子对象用一定的算法构成：由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是它们组合在一起的算法却相对稳定。

对象性能模式：单件模式、享元模式

单件模式：保证一个类仅有一个实例，并提供一个该实例的全局访问点。

动机：在软件系统中，经常有这样一些特殊的类，必须保证它们在系统中只存在一个实例，才能确保它们的逻辑正确性、以及良好的效率。

享元模式：运用共享技术有效地支持大量细粒度的对象。

动机：在软件系统采用纯粹对象方案的问题在于大量细粒度的对象会很快充斥在系统中，从而带来很高的运行时代价——主要指内存需求方面的代价。

接口隔离模式：门面模式、代理模式、适配器、中介者

门面模式：为子系统中的一组接口提供一个一致（稳定）的界面，门面模式定义了一个高层接口，这个接口使得这一子系统更加容易使用（复用）

动机：组件的客户和组件中各种复杂的子系统有了过多的耦合，随着外部客户程序和各子系统的演化，这种过多的耦合面临很多变化的挑战。

代理模式：为其他对象提供一种代理以控制（隔离、使用接口）对这个对象的访问。

动机：在面向对象系统中，有些对象由于某种原因（比如对象创建的开销很大，或者某些操作需要安全控制，或者需要进程外的访问等），直接访问会给使用者、或者系统结构带来很多麻烦。

适配器：将一个类的接口转换成客户希望的另一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

动机：在软件系统中，由于应用环境的变化，常常需要将“一些现存的对象“放在新的环境中应用，但是新环境要求的接口是这些现存对象所不满足的。

中介者：用一个中介对象来封装（封装变化）一系列的对象交互。中介者使各对象不需要显示的互相引用（编译时依赖->运行时依赖），从而使其耦合松散（管理变化），而且可以独立地改变它们之间的交互。

动机：在软件构建过程中，经常会出现多个对象互相关联交互的情况，对象之间常常会维持一种复杂的引用关系，如果遇到一些需求的更改，这种直接的引用关系将面临不断的变化。

“状态变化”模式：状态模式、备忘录

状态模式：允许一个对象在其内部状态改变时改变它的行为、从而使对象看起来似乎修改了其行为。

动机：在软件构建过程中，某些对象的状态如果改变，其行为也会随之而发生变化，比如文档处于只读状态，其支持的行为和读写状态支持的行为就可能完全不同。

备忘录：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可以将该对象恢复到原先保存的状态。

动机：在软件构建过程中，某些对象的状态在转换过程中，可能由于某种需要，要求程序能够回溯到对象之前处于某个点时的状态。如果使用一些公有接口来让其他对象得到对象的状态，便会暴露对象的细节实现。

“数据结构“模式：组合模式、迭代器、职责链

组合模式：将对象组合成树形结构以表示“部分-整体“的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性（稳定）。

动机：在软件的某些情况下，客户代码过多地依赖于对象容器复杂的内部实现结构，对象容器内部实现结构（而非抽象接口）的变化将引起客户代码的频繁变化，带来了代码的维护性、扩展性等弊端。

迭代器模式：提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露（稳定）该对象的内部表示。

动机：在软件构建过程中，集合对象内部结构常常变化各异。但对于这些集合对象，我们希望在不暴露其内部结构的同时，可以让外部客户代码透明地访问其中包含的元素；同时这种“透明遍历“也为”同一种算法在多种集合对象上进行操作“提供了可能。

职责链：使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递请求，直到有一个对象处理它为止。

动机：在软件构建过程中，一个请求可能被多个对象处理，但是每个请求在运行时只能有一个接受者，如果显示指定，将必不可少地带来请求发送者与接收者的紧耦合。

“行为变化”模式：命令模式、访问器

命令模式：将请求（行为）封装为一个对象，从而使你可用不同的请求对客户进行参数化；对请求排队或记录请求日志，以及支持可撤销的操作。（与函数对象类似）

动机：在软件构建过程中，“行为请求者”与“行为实现者”通常呈现一种“紧耦合”。但在某些场合——比如需要对行为进行“记录、撤销、事务”等处理，这种无法抵御变化的紧耦合是不合适的。

访问器：表示一个作用于某对象结构中的各元素的操作。使得可以在不改变（稳定）各元素的类的前提下定义（扩展）作用于这些元素的新操作（变化）。

动机：在软件构建过程中，由于需求的改变，某些类层次结构中常常需要增加新的行为（方法），如果直接在基类中做这样的更改，将会给子类带来很繁重的变更负担，甚至破坏原有设计。

“领域规则“模式：解析器

解析器： 给定一个语言，定义它的文法的一种表示，并定义一种解释器，这个解释器使用该表示来解释语言中的句子。

动机：在软件构建过程中，如果某一特定领域的问题比较复杂，类似的结构不断重复出现，如果使用普通的编程方式来实现将面临非常频繁的变化。

64.转型构造函数就是只有一个参数的构造函数。

# C

1.Windows DLL编程（DLL动态链接技术）

\_\_declspec(dllexport)导出到dll

\_\_declspec(dllimport)从dll导入

2. %hd是一个短整数

%ld是一个长整数

%x的意思是以十六进制数形式输出整数

3. #define SQUARE(x) x\*x //定义宏函数

#define QUOTE(name) #name //也是一个宏函数，#name的作用就是将传入的name进行字符串化

4. atoi(s)函数用于把一个char型的[字符串](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020)转换成一个整型数据，

例如：b = aito（“12345.67”） 现在就等于12345

5. property\_get("vendor.camera.calibration.mode", szProp, "0");

每个属性都有一个名称和值，他们都是字符串格式。属性被大量使用在Android系统中，用来记录系统设置或进程之间的信息交换。属性是在整个系统中全局可见的。每个进程可以get/set属性。

Int8 szProp[PROPERTY\_VALUE\_MAX];

memset(szProp, 0, sizeof(szProp));

property\_get("iwb.removestill", szProp, "0");

Int32 iIsOpenRemoveStill = atoi(szProp);

iwb.removestill这相当于一个全局变量，调试的时候可以手动设置这个全局变量，"0"表示默认设置这个全局变量是0，

6.

[\_\_DATE\_\_](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.codersrc.com%2Farchives%2F9308.html) 当前日期，一个以 “MMM DD YYYY” 格式表示的字符串常量。

\_\_TIME\_\_ 当前时间，一个以 “HH:MM:SS” 格式表示的字符串常量。

[\_\_FILE\_\_](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.codersrc.com%2Farchives%2F9281.html) 这会包含当前文件名，一个[字符串](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.codersrc.com%2Farchives%2F7815.html)常量。

[\_\_LINE\_\_](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.codersrc.com%2Farchives%2F9270.html) 这会包含当前行号，一个[十进制](https://links.jianshu.com/go?to=https%3A%2F%2Fwww.codersrc.com%2Farchives%2F8488.html)常量。

\_\_FUNCTION\_\_ 当前函数名，一个字符串常量

OSA\_Printf ("[%s]", \_\_FUNCTION\_\_,);

7. memcpy是内存拷贝，什么数据都可以

8.

强转变量时，定义变量要考虑对应，一级指针是可以转换为二级指针，其语法上没问题，但是使用时好像有问题，目前还没想通

比如：int \*retval;

(void \*\*)&retval //这边就是一个二级指针转二级指针

9.

printf(“你好 %d”, a)

sprintf函数的格式：int sprintf( char \*[buffer](https://so.csdn.net/so/search?q=buffer&spm=1001.2101.3001.7020), const char \*format [, argument,...] );

除了前两个参数固定外，可选参数可以是任意个。buffer是[字符数组](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E6%95%B0%E7%BB%84&spm=1001.2101.3001.7020)名；format是格式化字符串

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](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020)中了

fprintf函数的格式：int fprintf( FILE \*stream, const char \*format, [ argument ]…)

将结果输出到对应的流文件中。

fscanf 函数原型为 int fscanf(FILE \* stream, const char \* format, [argument...]); 其功能为根据数据格式(format)，从输入流(stream)中读入数据，存储到argument中。

10.

int pthread\_setaffinity\_np(pthread\_t thread, size\_t cpusetsize， const cpu\_set\_t \*cpuset); //设置线程在某个CPU核心上跑

int pthread\_getaffinity\_np(pthread\_t thread, size\_t cpusetsize, cpu\_set\_t \*cpuset);

//得到CPU核心上跑了那些线程

从函数名以及参数名都很明了，唯一需要点解释下的可能就是cpu\_set\_t这个结构体了。这个结构体的理解类似于select中的fd\_set，可以理解为cpu集，也是通过约定好的宏来进行清除、设置以及判断：

void CPU\_ZERO (cpu\_set\_t \*set); //初始化，设为空

void CPU\_SET (int cpu, cpu\_set\_t \*set); //将某个cpu加入cpu集中

void CPU\_CLR (int cpu, cpu\_set\_t \*set); //将某个cpu从cpu集中移出

int CPU\_ISSET (int cpu, const cpu\_set\_t \*set); //判断某个cpu是否已在cpu集中设置了

11.

int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5);

prctl(PR\_SET\_NAME, “process\_name”)

第一个参数是操作类型，指定PR\_SET\_NAME，即设置进程名  
第二个参数是进程名字符串，长度至多16字节

12.

回调函数就是一个通过函数指针调用的函数。

13.

restrict关键字：用来限定指针变量，被该关键字限定的指针变量所指向的内存操作，必须由本指针完成

14. int system(const char \* string);

system()会调用fork()产生子进程，由子进程来调用/bin/sh-c string来执行参数string[字符串](https://so.csdn.net/so/search?q=%E5%AD%97%E7%AC%A6%E4%B8%B2&spm=1001.2101.3001.7020)所代表的命令。

15. int fcntl(int fd, int cmd);  
int fcntl(int fd, int cmd, long arg);

函数作用：可以用来对已打开的文件描述符进行各种控制操作以改变已打开文件的的各种属性



16.fabsf 函数 返回浮点的绝对值

17.

union联合体

联合的成员是共用一块内存空间的，这样一个联合变量的大小，至少是最大成员的大小

在计算联合体大小之前我们必须知道两个知识点：  
1.联合的大小至少是最大成员的大小  
2.当最大成员大小不是最大对齐数的整数倍的时候，就要对齐到最大对齐数的整数倍。

18. 使用c语言编写的文件要用以下格式框起来

#ifdef \_\_cplusplus

extern "C"{

#endif

xxxxx //C语言的n个函数实现

#ifdef \_\_cplusplus

}

#endif

原因是在C++的环境中\_\_cplusplus这个宏定义会自动被定义，其作用就可以将此块被框起来的代码按C语言的方式去编译

19. malloc分配得到的地址也可以用下标来进行访问：

uint64\_t \*chunk = (uint64\_t \*)malloc(65536);

printf("chunk%u\n",chunk[8191]);

20.

cin.clear(); /\*\* 清除输入错误标志 \*\*/比如要输入一个整数int型，但是输入了一个字符，这时候cin流中就会置一个错误标志，再输入其他数据都会受到影响，简单的理解，流错了，咋弄都白扯（典型的错误就是你输错之后，如果是循环输入的，后面都不提示你输入了，死循环）

cin.ignore(1024, '\n'); cin.ignore()函数中有两个参数，分别为数值型的a 和 字符型的 ch ，即cin.ignore( a, ch )。它表示从输入流 cin 中提取字符，提取的字符被忽略，不被使用。

21.

文件操作整理：

FILE \* fopen ( const char \* filename, const char \* mode );

参数：

filename：文件名

mode：打开方式 “rb“打开二进制文件只读

函数作用：打开文件



int fclose ( FILE \* stream );

参数：

stream：文件名

函数作用：关闭文件

size\_t fread( void \*buffer, size\_t size, size\_t count, FILE \*stream );

参数：

buffer:存放读取文件内容的地址

size:读取数据块的字节数

count:要读取的数据块数

stream:文件指针

函数作用：读文件

size\_t fwrite( const void \*buffer, size\_t size, size\_t count, FILE \*stream );

参数：

buffer:存放需写入文件内容的地址

size:写入数据块的字节数

count:要写入的数据块数

stream:文件指针

函数作用：写文件

int fseek( FILE \*stream, long offset, int origin );

参数：

stream：文件指针

offset：偏移量。（可正可负）

origin:从哪个位置开始计算偏移量。位置可取3种值：文件首部、当前位置和文件尾部，实际表示时分别对应值0、1、2，或常量SEEK\_SET、SEEK\_CUR、SEEK\_END

函数作用：调节文件指针位置

int fflush(FILE \*stream)

参数：

stream：文件指针

函数作用：可以实时将缓冲区中的数据写入磁盘中

22. 不要在switch语句内定义变量 //其提示错误为jump to case label [-fpermissive]

23. 字符串比较（char类型）

函数原型

extern int strcmp(const char \*s1, const char \*s2)

当s1 < s2 返回值：负数

当s1 = s2 返回值：0

当s1 > s2 返回值：正数

24. 枚举类型的初始化：后一个会自动比前一个名称大1，也就是CC\_STRATEGY\_LEFT\_FIRST是1，CC\_STRATEGY\_RIGHT\_FIRST是2以此类推

typedef enum tagCcStrategeyEn /\* 左右先\*/

{

CC\_STRATEGY\_NONE = 0,

CC\_STRATEGY\_LEFT\_FIRST ,

CC\_STRATEGY\_RIGHT\_FIRST ,

CC\_STRATEGY\_MIDDLE\_FIRST ,

CC\_STRATEGY\_MAX

} CC\_STRATEGY\_E;

C语言中枚举类型的值是暴露在全局作用域下的

25. 正负数在if语句中均为真

26. 套接字相关

fd\_set是一组文件描述字(fd)的集合，

fd\_set set;

FD\_ZERO(&set); //用于清空指定的文件描述符集合

FD\_SET(0, &set); //用于在文件描述符集合中添加一个新的文件描述符

FD\_CLR(4, &set); //用于在文件描述符集合中删除一个文件描述符

FD\_ISSET(5, &set); //用于测试指定的文件描述符是否在该集合中

int select(int nfds, fd\_set \*readset, fd\_set \*writeset,fd\_set\* exceptset, struct tim \*timeout);

参数：

nfds：为需要检查的三组文件描述符集合中所含的最大fd值+1

readset：用来检查可读性的一组文件描述符集合

writeset：用于检测可写性的一组文件描述符集合

exceptset：用于检测是否有异常条件出现的文件描述符集合

timeout：用于描述一端时间长度，如果在这个时间内，需要监视的描述符没有事件发生则函数返回，返回值为0

返回值：成功时：返回三中描述符集合中”准备好了“的文件描述符数量。  
超时：返回0  
错误：返回-1，并设置 errno

函数作用：测试指定的文件描述符集合可读？可写？有异常条件待处理？

int socket(int domain, int type, int protocol);

参数：

domain：

参数:

domain: 使用的地址族协议

AF\_INET: 使用 IPv4 格式的 ip 地址

AF\_INET6: 使用 IPv4 格式的 ip 地址

type:

SOCK\_STREAM: 使用流式的传输协议

SOCK\_DGRAM: 使用报式 (报文) 的传输协议

protocol: 一般写 0 即可，使用默认的协议

SOCK\_STREAM: 流式传输默认使用的是 tcp

SOCK\_DGRAM: 报式传输默认使用的 udp

返回值:

成功：可用于套接字通信的文件描述符

失败: -1

函数作用：创建一个套接字

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

sockfd: 监听的文件描述符，通过 socket () 调用得到的返回值

addr: 传入参数，要绑定的 IP 和端口信息需要初始化到这个结构体中，IP和端口要转换为网络字节序

addrlen: 参数 addr 指向的内存大小，sizeof (struct sockaddr)

返回值：成功返回 0，失败返回 - 1

函数作用：将文件描述符和本地的IP与端口进行绑定

int listen(int sockfd, int backlog);

参数:

sockfd: 文件描述符，可以通过调用 socket () 得到，在监听之前必须要绑定 bind ()

backlog: 同时能处理的最大连接要求，最大值为 128

返回值：函数调用成功返回 0，调用失败返回 -1

函数作用：给监听的套接字设置监听

int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

sockfd: 监听的文件描述符

addr: 传出参数，里边存储了建立连接的客户端的地址信息

addrlen: 传入传出参数，用于存储 addr 指向的内存大小

返回值：函数调用成功，得到一个文件描述符，用于和建立连接的这个客户端通信，调用失败返回 -1

函数作用： 等待并接受客户端的连接请求, 建立新的连接, 会得到一个新的文件描述符(通信的) 这个函数是一个阻塞函数，当没有新的客户端连接请求的时候，该函数阻塞；当检测到有新的客户端连接请求时，阻塞解除，新连接就建立了，得到的返回值也是一个文件描述符，基于这个文件描述符就可以和客户端通信了。

ssize\_t read(int sockfd, void \*buf, size\_t size);

ssize\_t recv(int sockfd, void \*buf, size\_t size, int flags);

参数:

sockfd: 用于通信的文件描述符，accept () 函数的返回值

buf: 指向一块有效内存，用于存储接收是数据

size: 参数 buf 指向的内存的容量

flags: 特殊的属性，一般不使用，指定为 0

返回值:

大于 0：实际接收的字节数

等于 0：对方断开了连接

-1：接收数据失败了

函数作用：接收数据，如果连接没有断开，接收端接收不到数据，接收数据的函数会阻塞等待数据到达，数据到达后函数解除阻塞，开始接收数据，当发送端断开连接，接收端无法接收到任何数据，但是这时候就不会阻塞了，函数直接返回0。

ssize\_t write(int fd, const void \*buf, size\_t len);

ssize\_t send(int fd, const void \*buf, size\_t len, int flags);

参数:

fd: 通信的文件描述符，accept () 函数的返回值

buf: 传入参数，要发送的字符串

len: 要发送的字符串的长度

flags: 特殊的属性，一般不使用，指定为 0

返回值：

大于 0：实际发送的字节数，和参数 len 是相等的

-1：发送数据失败了

函数作用：发送数据的函数，如果发送缓存已满也会阻塞在这个函数上

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

参数:

sockfd: 通信的文件描述符，通过调用 socket () 函数就得到了

addr: 存储了要连接的服务器端的地址信息: iP 和 端口，这个 IP 和端口也需要转换为大端然后再赋值

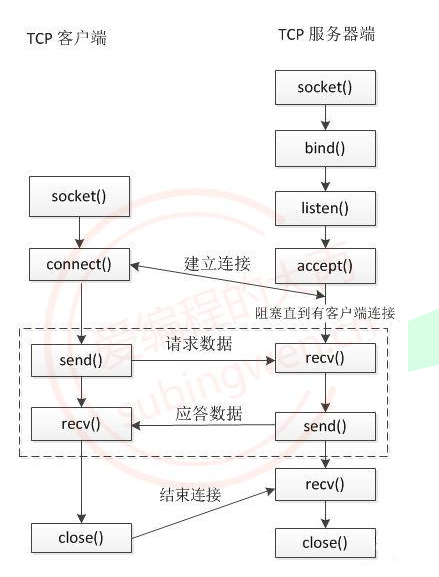
addrlen: addr 指针指向的内存的大小 sizeof (struct sockaddr)

返回值：连接成功返回 0，连接失败返回 - 1

函数作用：成功连接服务器之后, 客户端会自动随机绑定一个端口

服务器端调用accept()的函数, 第二个参数存储的就是客户端的IP和端口信息

TCP通信流程



ssize\_t recvfrom(int sockfd, void \*buf, size\_t nbytes,

int flags, struct sockaddr \*from, socklen\_t \*addrlen);

参数：

sockfd：用于通信的文件描述符

buf：接收数据的首地址

nbytes：可接收数据的最大长度

flags: 特殊的属性，一般不使用，指定为 0

from：存放发送方的IP地址和端口

addrlen：用于存储 from 指向的内存大小

返回值：若成功，均返回读或者写的字节数；失败则返回-1

ssize\_t sendto(int sockfd, const void \*buf, size\_t nsize,

int flags, const struct sockaddr \*to, const socklen\_t \*addrlen);

函数作用：用于UDP接收数据

sockfd：用于通信的文件描述符

buf：发送数据的首地址

nsize：发送数据的最大长度

flags: 特殊的属性，一般不使用，指定为 0

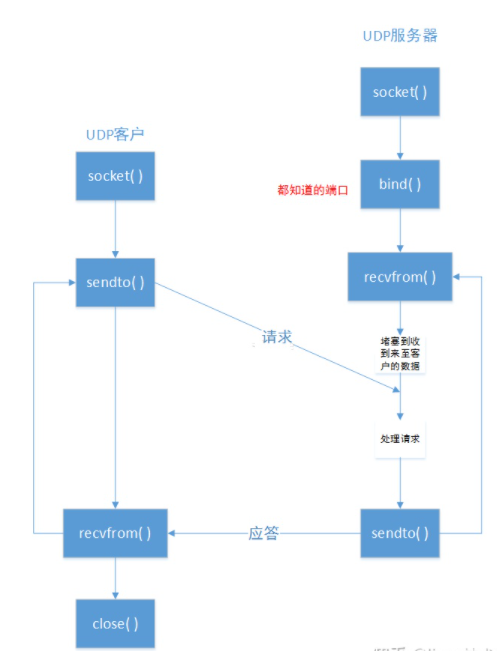
to：存放目的主机的IP地址和端口

addrlen：用于存储 to指向的内存大小

返回值：若成功，均返回读或者写的字节数；失败则返回-1

函数作用：用于UDP发送数据

UDP通信流程·



send 和 sendto 函数在 UDP 层没有输出缓冲区，在 TCP 层有输出缓冲区，recv 和recvfrom 无论在 UDP 层还是 TCP 层都有接收缓冲区。这样看来 sendto 应该是不会阻塞的。

27.字节序

小端

数据的低位字节存储到内存的低地址位 , 数据的高位字节存储到内存的高地址位

我们使用的 PC 机，数据的存储默认使用的是小端

大端

数据的低位字节存储到内存的高地址位 , 数据的高位字节存储到内存的低地址位

套接字通信过程中操作的数据都是大端存储的，包括：接收/发送的数据、IP地址、端口。

28.

memcpy(&(TempReidDetFromBody.astReidDet),&(\*iter),sizeof(TempReidDetFromBody.astReidDet));