程序设计基本概念

**1 一个小程序，输出结果**

int x = 2, y, z;

x \*= (y = z = 5);//等价于x = x \* y

cout << x << endl;//输出为10

z = 3;

int t = (x == (y = z));

cout << x << endl;//输出为10

cout << t << endl;//输出为0

x = (y == z);

cout << x << endl;//输出为1

x = y & z;

cout << x << endl;//输出为3

x = y && z;

cout << x << endl;//输出为1

y = 4;

x = (y | z);

cout << x << endl;//输出为7

x = (y || z);

cout << x << endl;//输出为1

**2** int i = 1, j = 2;

int k = i+++j;//等价于(i++) + j

cout << k << endl;//输出为3

微软公司的最新C++编译器2005解读C++源程序时如果需要“断句”，则规则是每次都会先找到包含尽量多的字符的一个语素后再在其后断开。比如对于欠揍人写的欠揍语句

                                   i+++j  （本身可能解释成 i++  +j和 i+  ++j两种语句 ）

但是按照以上原则：则前面语素必须拥有最多的字符：于是只能解释成 i+++j 等效为 i++     + j

**3 x = x + 1, x +=1, x++哪个效率最高？为什么？**

X++ > x +=1 >x = x+1

x = x + 1: (1)读取右x的地址（2）x+1（3）读取左x的地址（编译器病不认为左右x的地址相同）（4）讲右边的只给左边的x

x+=1：（1）读取右边x的地址（2）x+1（3）讲得到的值给x（因为x的地址已经读出）

x++：（1）读取右x的地址（2）x自增1

**4 输出**

#define product(x) (x \* x)

int i = 3, j, k;

j = product(i++);

k = product(++i);

cout << j << " " << k << endl;//输出为9和49

即使定义为#define product(x) ((x) \* (x))得到的结果还是一样

**5 类型转换**

char foo(void)

{

unsigned int a = 6;

int b = -20;

char c;

(a + b > 6) ? (c = 1) : (c = 0);

return c;// 此时c=1

}

Unsigned int类型的数据与int类型的数据相运算后，自动转化为unsigned int类型，因此a+b

的结果不是-14，而是一个unsigned int类型的数4294967382，当表达式中存在有符号类型和

无符号类型时，所有的操作数都自动转换为无符号类型

1 在混合类型的算数表达式中

在这种情况下最宽的数据类型称为目标转换类型，这也被称为算数转换

int ival = 3;

double dval = 3.141592;

cout << ival + dval << endl;//输出3.14159，这里int被提升为了double类型

2 用一种类型的表达式赋值为另外一种类型的对象

这种情况下目标转换类型是被赋值对象的类型

int \*p = 0;

int t = dval;

**6 a，b中较大的值，不用if，?:switch语句实现**

int a, b;

cin >> a >> b;

int max = (a + b + abs(a - b)) / 2;

cout << max << endl;

**7 a，b进行交换**

方案一：

a = a + b;

b = a - b;

a = a - b;

方案二：

a = a ^ b;

b = a ^ b;

a = a ^ b;（已经证明是对的）

方案一对大数据无能为力，因为a + b会超界

一点解释：

a = a ^ b;

b = a ^ b = a ^ b ^ b = a ^ 0 = a;

a = a ^ b= a ^ b ^ a = 0 ^ b = b;

**8 在c++程序中调用被c编译器编译后的函数，为什么要加上extern “C”？**

C++支持函数重载，c语言不支持函数重载，函数被c++编译后在库中的名字与c语言不

同，假设某个函数的原型为：void foo（int x，int y）。该函数被c编译器编译后在库中的

名字为foo\_，而c++编译器则会产生像\_foo\_int\_int之类的名字

C++提供了c链接交换指定符号extern “C”解决名字匹配问题

**9 #include <filename.h>与#include “filename.h”有什么区别？**

对于尖括号来说，编译器从标准库路径开始搜索filename.h

对于圆括号来说，编译器先从用户的定义的文件开始查找，找不到再在标准库中进行查找

**10 如何判断一段程序是由c编译器还是由c++编译器编译的？**

C++编译时定义了\_cplusplus

C编译时定义了\_STDC\_

#ifdef \_cplusplus

cout << "hello,cpp!";

#endif

#ifdef \_STDC\_

printf("hello,c!\n");

#endif

这两者实际上是可以共存在，很多编译器上都是两者共存

**11 main主函数执行完毕后，是否可能会再只执行一段代码？给出说明**

如果需要加入一段在main退出后执行的代码，可以使用atexit()函数注册一个函数

atexit(fn1);//输出next

atexit(fn2); //输出executed

atexit(fn3); //输出is

atexit(fn4); //输出This

cout << "This is executed first" << endl;

最终结果为：This is executed first This is executed next

注意：atexit（）注册的函数类型应为不接受任何参数的void函数,exit调用这些注

册函数的顺序与它们 登记时候的顺序相反

预处理，Const与sizeof

**1 代码输出**

#define SQR(x) (x \* x)

int a, b = 3;

a = SQR(b + 2);

cout << a << endl;//输出为11

如果把定义为#define SQR(x) ((x) \* (x))输出结果变为25

**2 宏定义**

写一个标准的“宏”MIN，这个宏输入两个参数并返回较小的一个

#define MIN(x, y) ((x) <= (y) ? (x) : (y))

注意：#define MIN(x, y) (x) <= (y) ? (x) : (y)这样就是错误的

**3 const与#define相比有什么不同？**

1. const常量有数据类型，而宏常量没有数据类型，编译器可以对前者进行类型安全检测，而对后者只进行字符替换，没有类型男权检查，并且在字符替换中可能uixiang产生意料不到的错误（边际效应）
2. 有些集成化的调试工具可以对const常量进行调试，但是不能对宏常量进行调试，在c++程序中只使用const常量而不使用宏常量，即const常量完全取代宏常量

**4 数据对齐原则**

struct Node

{

long a1;

short a2;

};

class person

{

private:

long a1;

short a2;

};

cout << sizeof(long) << endl;//输出为4

cout << sizeof(int) << endl;//输出为4

cout << sizeof(short) << endl;//输出为2

Node t;

cout << sizeof(t) << endl;//输出为8

person per;

cout << sizeof(per) << endl;//输出为8

在默认情况下，为了方便对结构体内元素的访问和管理，当结构体内的元素的长度都小于处

理器的位数的时候，便以结构体里面最长的数据元素为对齐单元，也就是说，结构体的长度

一定是最长的数据元素的整数倍，如果结构体内存在长度大于处理器位数的元素，那么就以

处理器的位数为对齐单位，但是结构体内类型相同的联素元素将在连续的空间内，和数组一

样

class A

{

};

class A2

{

char d, e;

};

cout << sizeof(A) << endl;//输出为1

cout << sizeof(A2) << endl;//输出为2

注意：对于一个类而言，即便是它为空类，编译器仍然要给它分配一个空间，即使它什么也

没有

class A3

{

static int ch;

};

cout << sizeof(A3) << endl;//输出为1

因为静态变量是存放在全局数据区的，而sizeof计算栈中分配的大小，是不会计算在内的

class T1

{

};

class T2: public virtual T1

{

};

cout << sizeof(T1) << endl;//输出为1

cout << sizeof(T2) << endl;//输出为4

空类所占空间为1，单一继承的空类也为1，多重继承的空类空间还是1，但是虚继承涉及

到虚表（虚指针)，所以sizeof（T2）的大小为4

对于一个空类而言，事实上它有一个隐晦的一字节，那是被编译器安插进去的一个char，这

使得这个class的两个对象得以在内存中配置独一无二的地址

**5 内联函数和宏的主要差别是什么？**

内联函数和普通函数相比可以加快程序运行速度，因为不需要中断调用，在编译的时候内

联函数可以直接被镶嵌到目标代母中，而宏只是一个简单的替换，内联函数要做参数类型

检查，这是内联函数跟宏相比的优势

指针与引用

**1 关于this指针**

1. 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，成员函数默认的第一个参数是T \* const this，由此可见，this在成员函数的开始前构造，在成员的结束后清除，这个声明周期同任何一个函数的参数是一样的，没有任何区别，我们会发现在调用的形式上与静态调用没有什么区别，但区别还是有的，iqi通常会对this指针做一些优化，因此，this指针的传递效率比较高---如VC通常是通过ecx寄存器传递this参数的

1. this指针存放在何处？

This指针会因编译器不同而有不同的存放位置，可能是栈，也可能是寄存器，甚至是全局变量，在汇编级别里面，一个值只会以3种形式出现：立即数，寄存器值和内存变量值，不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的

1. 类在实例化时只分配类中的变量空间，并没有为函数分配空间
2. 每个类编译后，是否创建一个类中函数表保存函数指针以便用来调用函数？

普通的类函数（不论是长远函数还是静态函数）都不会创建一个函数表来保存函数指针，只有虚函数才会被放到函数表中

但是，即使是虚函数，如果编译器能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用么人是会直接调用该函数

**2 一个小程序**

void swapxy(char \*a, char \*b)

{

int x = \*a;

int y = \*b;

x = x + y;

y = x - y;

x = x - y;

\*a = x;

\*b = y;

return;

}

char a = 'a', b = 'b';

char &x = a, &y = b;

cout << a << " " << b << endl;

swapxy(x, y);

cout << a << " " << b << endl;

出现的错误：



注意：尽管是引用，但是编译器还是把它当作了char型，其实本来就是char型

**3 错误的代码**

int \*ptr;

ptr = (int \*)0x8000;

\*ptr = 7;

这样做会导致运行时错误，因为这种做法会给一个指针分配一个随意的地址，这是非常危险

的，不管这个指针有没有被使用过，这样做都是不允许的，会报写入冲突

**4 空指针和迷途指针的区别是什么？**

当delete一个指针的时，实际上仅仅是让编译器释放内存，当时指针本身依然存在，这时

它就是一个迷途指针

当使用下列语句时，可以把迷途指针改为一个空指针

MyPtr = 0；

通常在删除一个指针后又把它删除一次，程序会变得非常不稳定，任何情况都可能发生，

但是如果只是删除了一个空指针，则什么事都不会发生，这样做非常安全

使用迷途指针或空指针是非法的，而且有可能造成程序崩溃，如果指针是空指针们尽管同

样是崩溃吗但是它同迷途指针造成的崩溃相比是一种可以预料的崩溃，这样调试起来会方

便很多

**5 c++中有了malloc/free，为什么还要new/delete？**

Malloc和free是c++/c语言的标准库函数，new与delete是c++的运算符，他们都可以用

于申请动态内存和释放内存

对于非内部数据类型的对象而言，光用malloc与free无法满足动态对象的要求，对象在创

建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数，由于malloc和free

是库函数而不是运算符，不再编译器控制权限之内，不能够把执行狗喊函数和析构函数的任

务强加于malloc与free

因此，c++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成

清理与释放内存工作的运算符delete，new与delete不是库函数，而是运算符

**添加一点：**

**malloc/free 和new/delete的区别：**

1. new/delete是保留字,不需要头文件支持. malloc/free需要头文件库函数支持. 使用

malloc/free需要包含 #include<cstdlib> 或<stdlib>.

2) new 建立的是一个对象,new会根据对象计算大小，直接返回对象的指针，当使用完毕后

调用delete来释放，但malloc分配的是一块内存，需要用户制定所要分配内存的大小，

而且返回的均为void的指针，使用时需要相应的强制类型转化，使用结束后调用free来

释放内存.

1. new/delete的使用除了分配内存和释放，还调用了类型的构造函数和析构函数，而

malloc/free只是简单的分配和释放内存。

**6 句柄和指针的区别和联系是什么？**

句柄和指针其实是两个截然不同的概念，Windows系统用句柄标记系统资源，用句柄隐藏

系统的信息，只要知道有这个东西存在然后去调研那个就行了，它是一个32bit的uint，

只恨则标记某个物理内存地址，两者是不同的概念

STL模版与容器

**1 STL是跨平台的，一个类的模版叫做泛型类，一个函数的模版也自然叫做泛型函数**

游标（iterator）

**2 介绍一下STL和包容器**

C++的一个新特性就是采用了标准模版库，所有主要编程器销售商都把裱糊在内模版库作

为编译器的一部分进行提供，标准模版库是一个基于模版的容器类库，保罗链表，列表，队

列和堆栈。标准模版库还包含许多常用的算法，包括排序与查找

标准模版库的目的是提供对常用需求重新开发的一种替代方法，标准模版库已经经过测试

和调试，具有很搞的性能并且是免费的，最重要的是，标准模范库是可重用的，当你知道如

何使用一个标准模版库的容器以后，就可以在所有的程序中使用它而不需要重新开发了

面向对象

**1 面向对象技术的基本概念是什么?**

对象，类和继承

**2 下面的程序如果把静态成员设为私有，该如何访问？**

struct Test

{

Test(int){}

Test(){}

void fun(){}

static int GetHowMany()

{

return HowManyCats;

}

private:

static int HowManyCats;

};

int Test::HowManyCats = 20;//这里进行初始化

可以通过共有的静态成员函数进行访问

如果静态成员数据为public类型则既可以通过类也可以通过对象进行访问

**3 析构函数可以是内联函数么？**

析构函数可以为内联函数

inline ~T(void);

**4 构造函数能为虚函数么？**

不行，虚调用是一种可以在只有部分信息的情况下工作的机制，特别允许我们调用一个只

知道接口而不知道其准确对象类型的函数，但是如果要创建一个对象，势必要知道对象的准

确类型，因此构造函数不能为虚，构造函数是由系统在对象分配空间之前都调用的,而虚构

函数需要是创建了对象之后才能够调用~`所以是不行的~但是,析构函数就可以~而且析

构函数经常是用虚构函数

**5 如果虚函数是非常有效的，我们是否可以把每个函数都声明为虚函数？**

不行，这是因为虚函数是有代价的，；由于每个虚函数的对象都必须维护一个v表，因此在

使用虚函数的时候都会产生一个系统开销，如果仅是很小的类，而且不像培生其它的类，那

么就没有必要使用虚函数

cout << Test::GetHowMany() << endl;

cout << b.GetHowMany() << endl;

cout << p->GetHowMany() << endl;

cout << b.HowManyCats << endl;

cout << b.GetHowMany << endl;

cout << Test::GetHowMany << endl;

cout << p->GetHowMany << endl;



TT t;

cout << t.GetHowMany() << endl;

cout << t.GetHowMany << endl;

这里的TT是继承Test的



这里我们可以看出static函数也是继承了的，但是始终只是保持一个副本

**6 编写String的构造函数，析构函数和赋值函数**

loadstring::~loadstring(void)

{

std::cout << str << std::endl;

std::cout << "析构了" << std::endl;

delete[] str;

}

loadstring::loadstring(const char \*loadstr)

{

if(loadstr == NULL)//这里特别重要，因为在vc中前M是段保护的，访问非法，即strlen出错

{

str = new char[1];

\*str = '\0';

}

else

{

int length = strlen(loadstr);

str = new char[length + 1];

strcpy(str, loadstr);

}

}

loadstring::loadstring(const loadstring &pramstring)

{

int length = strlen(pramstring.str);

str = new char[length + 1];

strcpy(str, pramstring.str);

}

loadstring& loadstring::operator =(const loadstring &pramstring)

{

if(this == &pramstring)

return \*this;

delete[] str;//这里很重要，要先释放已经存在的内存

int length = strlen(pramstring.str);

str = new char[length + 1];

strcpy(str, pramstring.str);

std::cout << (void \*)str << std::endl;

return \*this;

}

多态的作用是什么呢？我们知道，封装可以隐藏实现细节，使得代码模块化，集成可以扩展

已经存在的代码模块（类），它们的目的都是为了代码重用，而多台则是为了实现另外一个

目的---接口重用

继承与接口

**1 一个输出问题**

class A

{

public:

virtual void f()

{

cout << "A";

}

};

class B: public A

{

public:

virtual void f()

{

cout << "B";

}

};

A \*pa = new A();

pa->f();

B \*pb = (B\*)pa;

pb->f();

delete pa, pb;

//delete pb;

pa = new B();

pa->f();

pb = (B\*)pa;

pb->f();



B \*pb = (B\*)pa;

该语句的意思是转换pa为B类型并新建一个指针pb，讲pa复制到pb中，但是这里有一点

请注意，就是pa的指针始终灭有发生变化，所以pb也指向pb的f函数，这里并不存在覆

盖问题

三种继承方式的总结：

**1 公有继承方式**

基类成员对其对象的可见性与一般类及其对象的可见性相同，共有成员可见，其他成员不

可见，这里保护成员与私有成员相同

基类成员对派生类的可见性对派生类来说，基类的公有成员和保护成员可见：基类的共有

成员和保护成员作为派生类的成员时，它们都保持原有的状态；基类的私有成员不可见：基

类的私有成员仍然是私有的，派生类不可访问基类中的私有成员

基类成员对派生类对象的可见性对派生类对象来说，基类的共有成员是可见的，其他成员

是不可见的

所以在共有继承时，派生类的对象可以访问基类中的公有成员，派生类的成员函数可以访

问基类中的公有成员和保护成员

**2 私有继承方式**

基类成员对其对象的可见性与一般类及其对象的可见性相同，公有成员可见，其他成员不

可见

基类成员对派生类的可见性对派生类来说，基类的公有成员和保护成员是可见的：基类的

公有成员和保护成员都作为派生类的私有成员，并且不能被这个派生类的子类所访问；基类

的私有成员是不可见的：派生类不可访问基类中的私有成员

基类成员对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的

所以，在私有继承时，基类的成员只能有直接派生类访问

**3 保护继承方式**

这种继承方式与私有继承方式的情况相同。两者的区别仅仅在于对派生类的成员而言，基

类成员对其对象的可见性与一般类及其对象的可见性相同，公有成员可见，其他成员不可见

基类成员对派生类的可见性对于派生类来说，基类的共有成员和保护成员是可见的：基类

的公有成员和保护成员都作为派生类的保护成员，并且不能被这个派生类的子类所访问；基

类的私有成员是不可见的：派生类不可访问基类中的私有成员

基类成员对派生类对象的可见性对派生类对象来说，基类的所有成员都是不可见的

所以在保护继承时，基类的成员也只能由直接派生类访问，而无法再往性爱继承

**一个小例子：**

class Base

{

private://一般来说应该变为protected，要不对派生类就没有用了

int t;

public:

Base(int x){t = x;};

};

class Derive: public Base

{

private:

int i;

public:

//Derive(int x, int y){i = x;};//这里出错，要在子类中设定初始成员变量要用下面的方式

Derive(int x, int y): Base(x){i = y;};//这是正确的

//void printTotal(){int tioal = i + Base::t;};//不能访问基类的私有成员

};

一个关于纯虚函数的例子：

class Shape

{

public:

Shape() = 0{};

~Shape(){};

virtual void Draw() = 0{};

virtual void DDD() = 0;

};

class Circle: public Shape

{

void Draw(){};

void DDD(){};

};

Circle circle;

1. 如果抽象类的构造函数没有定义则会出现链接错误
2. 如果在抽象类中定义了纯虚函数，则必须在派生类中进行重写，否则会出错

**虚函数的入口地址和普通函数有什么不同？**

每个虚函数都在vtable中占了一个表项，保存着一条跳转到它的入口地址的指令（实际上就

是保存了它的入口地址）。当一个包含虚函数的对象（注意，不是对象的指针）被创建的时

候，它的头部附加 个指针，指向btable中相应的位置，调用虚函数的时候，不管你是用什

么指针调用的，它先根据vtable找到入口地址再执行，从而实现了“动态联编”，而不像普

通函数那样简单地跳转到一个固定地址

**C+如何阻止一个类被实例化？**

是用抽象类或者将构造函数声明为private

**什么时候编译器会生成默认的的复制构造函数？**

只要自己没写，而程序需要就会生成

字符串

**题目：编程实现字符串转化为整型，不用atoi**

string str;

int sum = 0;

cin >> str;

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

{

sum = sum \* 10 + str[i] - '0';

}

cout << sum << endl;

**题目：重新 编写strcpy函数**

**Main函数中：**

char strsrc[100] = "zhonghuarenmingongheguo";

char \*str = new char[strlen(strsrc) + 1];

char \*p = strcpy1(str, strsrc);

cout << str << endl;

cout << strlen(p) << endl;

delete[] str;

函数实现：

char\* strcpy1(char \*strDest, const char\* strSrc)

{

int i;

char \*address = strDest;

for(i = 0; strSrc[i] != '\0'; i++)

strDest[i] = strSrc[i];

strDest[i] = '\0';

return address;

}

**为什么要返回一个char\*呢？**

为了实现链式表达式，返回具体指

int length = strlen(strcpy(strDest, “hello world”));

**串拷贝（strcpy）和内存拷贝（memcpy）有什么不同？它们适合于哪种情况下使用？**

Strcpy函数只能拷贝字符串，strcpy函数将源字符串的每个字节拷贝到目的字符串中，当遇

到字符串末尾的NULL字符时，它会删去该字符，并结束拷贝

Memcpy函数可以拷贝任意类型的数据，因为并不是所有的数据都是以NULL结束的，所以

要为memcpy函数制定要拷贝的字节数，

extern void \*memcpy(void \*dest, void \*src, unsigned int count);

在拷贝字符串时冗长使用strcpy函数，在拷贝其它数据（如结构）时，通常使用memcpy

**例子：**

char a = 256;

int d = a;

cout << d + 1 << endl;//输出为1

char数组溢出了，char类型的变量赋值范围为0～255，当把256赋值为a以后，超出了a

的有效取值范围，此时a的实际值为0

**例子：经原始串中指定子串删除（例如原始串为askdaskaskdaskg，指定删除串为ask，最**

**后结果为ddg）**

void DeleteStr(string &Srcstr, const string &Substr)

{

string Results("");

int i = 0;

while(i < Srcstr.size())

{

int j = i;

int k = 0;

while(k < Substr.size())

{

if(Srcstr[j] == Substr[k])//如果当前的匹配则查找两个字符串的下一个字符

{

j++;

k++;

}

else//如果当前不相等则退出

break;

}

if(k == Substr.size())//得到了一个匹配的子串，就跳跃子串个字符后寻找下一个字符

i += Substr.size();

else if(k == 0)//此时一个字符都没有匹配到

{

Results += Srcstr[i];

i++;

}

else//没有匹配子串则当前已经匹配完了k个字符，从下一个字符开始寻找

{

for(int t = i; t < i + k; t++)

Results += Srcstr[t];

i += k;//从匹配的k个字符之后进行匹配

}

}

cout << "删除子串以后的结果为：" << Results << endl;

}

**例子：请写出一个函数来模拟c++中的strstr函数：该函数的返回值是主传中字符子串的位**

**置以后的所有字符，请不要使用任何c程序已有的函数**

string LeftSting(const string &Srcstr, const string &Substr)

{

string Results("");

int i = 0;

while(i < Srcstr.size())

{

int j = i;

int k = 0;

while(k < Substr.size())

{

if(Srcstr[j] == Substr[k])

{

j++;

k++;

}

else

break;

}

if(k == Substr.size())//找到了子串

{

for(int t = i; t < Srcstr.size(); t++)

Results += Srcstr[t];

break;

}

else if(k == 0)//此时第一个不是匹配的

i++;

else//此时已经匹配了k个字符

i += k;

}

return Results;

}

**例子：存在一个数组。找出连续数之和最大的一段**

void Find(int number[], int length)

{

if(length == 0)//如果数组为空直接返回

return;

int Max = number[0], Acclulate = number[0];//初始化最大值为数组的第一个值

Node \*Position = new Node[length];//用于存取到当前位置可能的最大值

Node MaxPosition = {0, 0};//初始化最大的起始位置和终止位置

Position[0].begin = 0;

Position[0].end = 0;

for(int i = 1; i < length; i++)

{

if(Acclulate > 0)//前面的累积和大于

{

Position[i].begin = Position[i - 1].begin;

Position[i].end = Position[i - 1].end + 1;

Acclulate += number[i];

if(Acclulate > Max)

{

Max = Acclulate;

MaxPosition = Position[i];

}

}

else//前面的累加和小于等于则重新设定起始位置和终止位置

{

Position[i].begin = i;

Position[i].end = i;

Acclulate = number[i];

if(number[i] > Max)

{

Max = number[i];

MaxPosition = Position[i];

}

}

}

cout << "最大的值为：" << Max << endl;

cout << "他们分别为：";

for(int i = MaxPosition.begin; i <= MaxPosition.end; i++)

cout << number[i] << " ";

cout << endl;

}

数据结构基础

**例子：编程实现单链表的删除节点**

Node\* Find(Node\* Head, int num)

{

Node \*p = Head;

Node \*q = p;

while(p)

{

if(num != p->data)

{

q = p;//q用于保存要删除的p结点的前一个结点

p = p->next;

}

else//此时找到了要删除的结点

break;

}

if(!p)//此时没有找到该数

return NULL;

else

{

if(p == Head)//删除的是头结点

{

Head = p->next;

delete p;

return Head;

}

else

{

q->next = p->next;

return Head;

delete p;

}

}

}

**例子：编程实现单链表的逆置**

Node\* Reserve(Node \*Head)

{

Node \*p = Head;

if(!p)//空表

return NULL;

else if(!p->next)//单结点表

return NULL;

else

{

Node \*q = p->next;

p->next = NULL;//首先处理头结点,设置为尾结点

while(q)//处理中间结点

{

Node \*r = q->next;

q->next = p;

p = q;

q = r;

}

return p;

}

}

**例子：编程实现单链表的插入**

Node\* InsertNode(Node \*Head, int num)

{

Node \*newNode = new Node;

newNode->data = num;

if(!Head)//此时为空链表

{

newNode->next = NULL;

return newNode;

}

Node \*p = Head;

Node \*q = NULL;//q指向p结点之前的结点

while(p)//此时寻找位置

{

if(p->data < num)

{

q = p;

p = p->next;

}

else//此时找到了位置

break;

}

if(p == Head)//插入到头结点之前

{

newNode->next = Head;

Head = newNode;

}

else if(!p)//插入到尾结点之后,此时q指向尾结点

{

q->next = newNode;

newNode->next = NULL;

}

else//插入到p结点和q结点之间

{

newNode->next = q->next;

q->next = newNode;

}

return Head;

}

**例子：编程实现双链表删除结点（注意它和单链表删除结点的情况有所不同）**

Node\* DoubleLink\_DelNode(Node \*Head, int num)

{

Node \*p = Head;

if(!p)

return NULL;

while(p)

{

if(num != p->data)

p = p->next;

else

break;

}

if(!p)//没有找到要删除的结点

return NULL;

else

{

if(p == Head)//此时删除的是头结点

{

Head = Head->next;

delete p;

}

else if(p->next)//此时删除的是中间结点

{

p->prev->next = p->next;

p->next->prev = p->prev;

delete p;

}

else//删除的是尾结点

{

p->prev->next = NULL;

delete p;

}

}

return Head;

}

**例子：编程实现双链表的插入**

Node\* DoubleLink\_InsertNode(Node \*Head, int num)

{

Node \*newNode = new Node;//初始化产生一个新结点

newNode->data = num;

newNode->prev = NULL;

newNode->next = NULL;

Node \*p = Head;

Node \*q = NULL;

while(p)

{

if(p->data < num)

{

q = p;

p = p->next;

}

else

break;

}

if(p == Head)//此时是在头结点之前进行插入

{

newNode->next = p;

p->prev = newNode;

Head = newNode;

}

else if(!p)//在尾结点之后进行插入

{

q->next = newNode;

newNode->prev = q;

}

else//在中间进行插入

{

p->prev->next = newNode;

newNode->prev = p->prev;

newNode->next = p;

p->prev = newNode;

}

return Head;

}

**例子：如何证明一个表是循环链表**

link \* p,\*q;

p=head;

q=p->next;

while(q&&q->next&&p!=q)  //q or q->next ==NULL时无环， q==q时有环

{

p=p->next;

 q=q->next->next;

}

 if(p==q)   
     cout<<"have ring";   
else   
   cout<<"no ring";

**例子：朴素字符串匹配算法的一点改进（子串必须各个字符不相等），时间复杂度O(n)**

int Match(const string &str1, const string &str2)

{

int i = 0;

while(i < str1.size())

{

int j = i;

int t = 0;

while(t < str2.size())

{

if(str1[j] == str2[t])

{

j++;

t++;

}

else

break;

}

if(t == str2.size())//此时找到了子串

return i;

if(t == 0)//当第一个字符都不匹配的时候就取下一个

i++;

else//前面有部分字符匹配成功的情况

i += t;

}

return -1;

}

另外一种方法：

int i = 0, q = -1;//q表示在当前字符之前已经正确匹配到第几个字符了

while(i < str1.size())

{

if(str1[i] == str2[q + 1])

q += 1;

else if(q != -1)//当前已经匹配了个以上的字符

{

q = -1;

i--;;//保持源字符串的当前字符不变

}

if(q == str2.size() - 1)//得到一组解

return i - q;

i++;

}

return -1;

**例子：实现队列的出队与入队**

//数据入队列

Node \*EnQueue(Node \*head, Node \*\*tail, int data)

{

//创建一个新结点

Node \*p = new Node;

p->data = data;

p->next = NULL;

if(head == NULL)//此时为空队列

{

head = p;

\*tail = p;

}

else

{

(\*tail)->next = p;

\*tail = p;

}

return head;

}

//删除头结点

Node\* DeQueue(Node \*head)

{

if(!head)//头结点为空

return NULL;

else

{

Node \*p = head;

head = head->next;

delete p;

}

return head;

}