#include<iostream>

conio.h cstdlib stdio  
using namespace std;  
int a,b,c;  
for (a=1;a<=20;a++)  
 for (b=1;b<=33;b++)  
 for (c=3;c<=99;c++)  
 if (5\*a+3\*b+c/3==100)  
 if (a+b+c==100)  
 if (c%3==0){   
 cout<<"公鸡数为："<<a<<"母鸡

数为："<<b<<"小鸡数为："<<c<<endl;  
}

template<class Type> //冒泡排序

void BubbleSort(Type\* array,int start,int end){

for(int i = 0;i < end - start;i++)

for(int j = 0;j < end - start;j++){

if(array[j] > array[j + 1])

swap(array[j],array[j + 1]);

}

}

template<class Type> //选择排序

void SelectionSort(Type\* array,int start,int end){

for(int i = 0;i < end - start;i++)

for(int j = i + 1;j < end - start + 1;j++){

if(array[i] > array[j])

swap(array[i],array[j]);

}

}

template<class Type> //插入排序

void InsertionSort(Type\* array,int start,int end){

for(int i = 1;i < end - start + 1;i++)

for(int j = 0;j < i;j++){

if(array[j] > array[i]){

int temp = array[i];

for(int k = i;k > j;k--)

array[k] = array[k - 1];

array[j] = temp; //此时array[j]是最大的了 break; } }

}

template<class Type> 快速排序

int Partition(Type \*a,int start,int end){

Type x = a[start];

int i = start;

for(int j = start + 1;j <= end;j++){

if(a[j] <= x){

i++;

swap(a[i],a[j]);

}

}

swap(a[i],a[start]); //把基准的值放在中间，则左边都小于他，右边都大于他 return i;

}

template<class Type>

void QuickSort(Type \*array,int start,int end){

if(start < end){

int q = Partition(array,start,end); //分割成两部分

QuickSort(array,start,q - 1);

QuickSort(array,q + 1,end); } }

template<class Type> //希尔排序

void ShellPass(Type\* array,int start,int end,int d){

for(int i = 0;i < d;i++){

for(int j = start + i + d;j < end - start + 1;j += d){

for(int k = start + i;k < j;k += d){

if(array[k] > array[j]){

Type temp = array[j];

for(int l = j;l > k;l -= d)

array[l] = array[l - d];

array[k] = temp;

}

}

}

}

}

template<class Type> 希尔排序

void ShellSort(Type\* array,int start,int end){

int d = 10;

while(d > 0){

d = (d + 1) / 2;

ShellPass(array,start,end,d);

if(d == 1)

break; } }

int main(){ //海盗

int pirate[30];

int i,j,survived;

for(i=0;i<30;i++)

pirate[i]=0;

i=0;j=0;

for(survived=30;survived>1;){

if(pirate[i]==0){

j++;

if(j%7==0){

pirate[i]=1;

cout<<"No."<<i+1<<" private jump"<<endl;

survived-=1;

} } i=(i+1)%30; }

for(i=0;i<30;i++)

if(pirate[i]==0)

cout<<"No."<<i+1<<" private survive"<<endl;

return 0;}

int f(int x) //兔子生兔子

{if (x == 1 || x == 2)

return 1;

else

return f(x - 1) + f(x - 2);

} int main(void)

{ int a; scanf("%d", &a);

a = f(a);

printf("\n%d", a);

return 0;}

int BinarySearch(int \*array, int aSize, int key)//不递归

{ if ( array == NULL || aSize == 0 ) //二分查找

return -1;

int low = 0;

int high = aSize - 1;

int mid = 0;

while ( low <= high )

{ mid = (low + high )/2;

if ( array[mid] < key)

low = mid + 1;

else if ( array[mid] > key )

high = mid - 1;

else

return mid;

}

return -1; }

int BinarySearchRecursive(int \*array, int low, int high, int key) //递归 //二分查找

{ if ( low > high )

return -1;

int mid = ( low + high )/2;

if ( array[mid] == key )

return mid;

else if ( array[mid] < key )

return BinarySearchRecursive(array, mid+1, high, key);

else

return BinarySearchRecursive(array, low, mid-1, key);

}

int main()

{ int array[10];

for (int i=0; i<10; i++)

array[i] = i;

cout<<"No recursive:"<<endl;

cout<<"position:"<<BinarySearch(array, 10, 6)<<endl;

cout<<"recursive:"<<endl; cout<<"position:"<<BinarySearchRecursive(array, 0, 9, 6)<<endl;

return 0; }

#include <iostreaminclude <iomanip>//二分法解方程

double func(double x)

{return (x\*x\*x - 6 \* x - 3);}

void root(double a, double b, double e, double \*pResult)

{ while (b - a >= e)

{ \*pResult = (a + b) / 2;

if (func(\*pResult) \* func(a) < 0)

{b = \*pResult;

}

else if (func(\*pResult) \* func(a) > 0)

{a = \*pResult;

}

else

{break;

} }}

int main()

{double e = (double)0.00000001;

double a = (double)2;

double b = (double)3;

double Result;

root(a, b, e, &Result);

cout << setiosflags(ios::fixed);

cout << "所求实根为: " << setprecision(9) << Result << endl; return 0;}

//复化梯形公式求定积分

#include<iostream> <cmath> std;

int main()

{ double up,down,a,b,c,n= 100; //积分区间被分为n份

cout << "依次输入积分上限up、积分下限down和函数参数a、b、c的值：" << endl;

cin >> up >> down >> a >> b >> c; cout << endl;

double h = (up -down) / n; //子区间长度,即迭代步长

double result, fx=0, fa, fb; //result积分最终结果

double x = 0;

for (int i = 1; i <= n - 1;i++)

{

fx += c / (1 + a\*pow(x, b)\*sqrt(x));

x += (up - down) / n;

}

fa = c / (1 + a\*pow(a, b)\*sqrt(a));

fb = c / (1 + a\*pow(b, b)\*sqrt(b));

result = (h / 2)\*(fa+2\*fx+fb);

cout << "步长h:" << h << endl;

cout << "f(a):" << fa << endl;

cout << "f(b):" << fb << endl;

cout <<"result:"<< result << endl;

return 0; }

#include <iostream> <cmath> std;

double f(double x)

{ return 1.0/(1+x\*x);

}

double SnSum(double a, double b, double n)

{ double sum = 0;

for(int i=1; i<=n; i++){

sum += 2\*f(1.0/n\*i)+4\*f((1.0/n\*(i-1)+1.0/n\*i)/2);

}

return sum;

}

double Sn(double a, double b, int n)

{ double h = (b-a)/n;

return h/6\*(f(a)+f(b)+SnSum(a,b,n));

}

int main()

{ double n;

cout<<"等分次数n:"; cin>>n;

cout<<"计算结果:"<<Sn(a,b, n)<<endl;

}

//牛顿迭代法求一元三次方程组根

#include <stdio.h>

#include<math.h>

double solut(double a,double b,double c,double d)

{ double x=1,x1=2,f,f1;

while(fabs(x1-x)>=0.00000001)

{ f=((a\*x+b)\*x+c)\*x+d;

f1=(3\*a\*x+2\*b)\*x+c;

x=x1;

x1=x-f/f1;

} return x1;}

int main()

{ double solut(double ,double ,double ,double );

double a,b,c,d;

scanf("%lf%lf%lf%lf",&a,&b,&c,&d);

printf("%.2f",solut(a,b,c,d));;

return 0; }

#include <iostream>

using namespace std;

class complex{ //复数类重载有关

public:

complex(double real = 0.0, double imag = 0.0): m\_real(real), m\_imag(imag){ };

public:

friend complex operator+(const complex & A, const complex & B);

friend complex operator-(const complex & A, const complex & B);

friend complex operator\*(const complex & A, const complex & B);

friend complex operator/(const complex & A, const complex & B);

friend istream & operator>>(istream & in, complex & A);

friend ostream & operator<<(ostream & out, complex & A);

private:

double m\_real; //实部

double m\_imag; //虚部

};

//重载加法运算符

complex operator+(const complex & A, const complex &B){

complex C;

C.m\_real = A.m\_real + B.m\_real;

C.m\_imag = A.m\_imag + B.m\_imag;

return C;

}

//重载减法运算符

complex operator-(const complex & A, const complex &B){

complex C;

C.m\_real = A.m\_real - B.m\_real;

C.m\_imag = A.m\_imag - B.m\_imag;

return C;

}

//重载乘法运算符

complex operator\*(const complex & A, const complex &B){

complex C;

C.m\_real = A.m\_real \* B.m\_real - A.m\_imag \* B.m\_imag;

C.m\_imag = A.m\_imag \* B.m\_real + A.m\_real \* B.m\_imag;

return C;

}

//重载除法运算符

complex operator/(const complex & A, const complex & B){

complex C;

double square = A.m\_real \* A.m\_real + A.m\_imag \* A.m\_imag;

C.m\_real = (A.m\_real \* B.m\_real + A.m\_imag \* B.m\_imag)/square;

C.m\_imag = (A.m\_imag \* B.m\_real - A.m\_real \* B.m\_imag)/square;

return C;

}

//重载输入运算符

istream & operator>>(istream & in, complex & A){

in >> A.m\_real >> A.m\_imag;

return in;

}

//重载输出运算符

ostream & operator<<(ostream & out, complex & A){

out << A.m\_real <<" + "<< A.m\_imag <<" i ";;

return out;

}

int main(){

complex c1, c2, c3;

cin>>c1>>c2;

c3 = c1 + c2;

cout<<"c1 + c2 = "<<c3<<endl;

c3 = c1 - c2;

cout<<"c1 - c2 = "<<c3<<endl;

c3 = c1 \* c2;

cout<<"c1 \* c2 = "<<c3<<endl;

c3 = c1 / c2;

cout<<"c1 / c2 = "<<c3<<endl;

return 0;

}

//C++类模板 函数模板看前面排序

#include <iostream>

using namespace std ;

template <class T>

class Base

{

public :

T a ;

Base(T b)

{

a = b ;

}

T getA(){ return a ;} //类内定义

void setA(T c);

};

template <class T> //模板在类外的定义

void Base<T>::setA(T c)

{

a = c ;

}

int main(void)

{

Base <int>b(4);

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

Base <double> bc(4);

bc.setA(4.3);

cout<<bc.getA()<<endl;

system("pause");

return 0 ;

}

#include<Cmath> //////重要

abs(X)

cos(x) //弧度

sin(x)

fabs(x)

ceil(x) 不小于x的最小整数

floor(x) 不大于x的最大帧数

log(x)

log10(x)

pow(x,y) 求x的y次方

sqrt(x) 出问题强制转换为(double)x

exp(x) e^x

//万扣灭灯

int t[6][8], w[6][8];

void fuck();

bool u();

int main(void)

{ //输入数组

for (int i = 1;i < 6;++i)

for (int j = 1;j < 7;++j)

cin >> t[i][j];

cout << endl;

fuck();

for (int i = 1;i < 6;++i)

{ for (int j = 1;j < 7;++j)

cout << w[i][j]<<" ";

cout << endl;

}

getch();

return 0;

}void fuck()

{ for (int a = 0;a < 2;++a)

for (int b = 0;b < 2;++b)

for (int c = 0;c < 2;++c)

for (int d = 0;d < 2;++d)

for (int e = 0;e < 2;++e)

for (int f = 0;f < 2;++f)

{

w[1][1] = a;

w[1][2] = b;

w[1][3] = c;

w[1][4] = d;

w[1][5] = e;

w[1][6] = f;

if (u())

return;

}

}

bool u()

{

for (int r = 1;r <= 4;++r)

for (int k = 1;k <= 6;++k)

w[r + 1][k] = (t[r][k] + w[r][k] + w[r - 1][k] + w[r][k - 1] + w[r][k + 1]) % 2;

for (int r = 1;r <= 6;++r)

if ((w[5][r] + w[5][r - 1] + w[5][r + 1] + w[4][r]) % 2 != t[5][r])

return false;

return true;

}

//链表

template<class T>

class slistNode

{

public:

slistNode(){next=NULL;}

T data;//值

slistNode\* next;//指向下一个节点的指针

};

template<class T>

class myslist

{

private:

unsigned int listlength;

slistNode<T>\* node;//临时节点

slistNode<T>\* lastnode;//头结点

slistNode<T>\* headnode;//尾节点

public:

myslist();//初始化

unsigned int length();//链表元素的个数

void add(T x);//表尾添加元素

void traversal();//遍历整个链表并打印

bool isEmpty();//判断链表是否为空

slistNode<T>\* find(T x);//查找值为x的节点,返回节点的地址,找不到返回NULL

void Delete(T x);//删除值为x的节点

void insert(T x,slistNode<T>\* p);//在p节点后插入值为x的节点

void insertHead(T x);//在链表的头部插入节点

};

template<class T>

myslist<T>::myslist()

{

node=NULL;

lastnode=NULL;

headnode=NULL;

listlength=0;

}

template<class T>

inline unsigned int myslist<T>::length(){return listlength;}

template<class T>

void myslist<T>::add(T x)

{

node=new slistNode<T>();//申请一个新的节点

node->data=x;//新节点赋值为x

if(lastnode==NULL)//如果没有尾节点则链表为空,node既为头结点,又是尾节点

{

headnode=node;

lastnode=node;

}

else//如果链表非空

{

lastnode->next=node;//node既为尾节点的下一个节点

lastnode=node;//node变成了尾节点,把尾节点赋值为node

}

++listlength;//元素个数+1

}

template<class T>

void myslist<T>::traversal()

{

node=headnode;//用临时节点指向头结点

while(node!=NULL)//遍历链表并输出

{

cout<<node->data<<ends;

node=node->next;

}

cout<<endl;

}

template<class T>

bool myslist<T>::isEmpty()

{

return listlength==0;

}

template<class T>

slistNode<T>\* myslist<T>::find(T x)

{

node=headnode;//用临时节点指向头结点

while(node!=NULL&&node->data!=x)//遍历链表,遇到值相同的节点跳出

{

node=node->next;

}

return node;//返回找到的节点的地址,如果没有找到则返回NULL

}

template<class T>

void myslist<T>::Delete(T x)

{

slistNode<T>\* temp=headnode;//申请一个临时节点指向头节点

if(temp==NULL) return;//如果头节点为空,则该链表无元素,直接返回

if(temp->data==x)//如果头节点的值为要删除的值,则删除投节点

{

headnode=temp->next;//把头节点指向头节点的下一个节点

if(temp->next==NULL) lastnode=NULL;//如果链表中只有一个节点,删除之后就没有节点了,把尾节点置为空

delete(temp);//删除头节点

return;

}

while(temp->next!=NULL&&temp->next->data!=x)//遍历链表找到第一个值与x相等的节点,temp表示这个节点的上一个节点

{

temp=temp->next;

}

if(temp->next==NULL) return;//如果没有找到则返回

if(temp->next==lastnode)//如果找到的时候尾节点

{

lastnode=temp;//把尾节点指向他的上一个节点

delete(temp->next);//删除尾节点

temp->next=NULL;

}

else//如果不是尾节点

{

node=temp->next;//用临时节点node指向要删除的节点

temp->next=node->next;//要删除的节点的上一个节点指向要删除节点的下一个节点

delete(node);//删除节点

node=NULL;

}

}

template<class T>

void myslist<T>::insert(T x,slistNode<T>\* p)

{

if(p==NULL) return;

node=new slistNode<T>();//申请一个新的空间

node->data=x;

node->next=p->next;

p->next=node;

if(node->next==NULL)//如果node为尾节点

lastnode=node;

}

template<class T>

void myslist<T>::insertHead(T x)

{

node=new slistNode<T>();

node->data=x;

node->next=headnode;

headnode=node;

}

//链表结束

string str="abc";

char \*p=str.data();

1、如果要将string转换为char\*，可以使用string提供的函数c\_str() ，或是函数data()，data除了返回字符串内容外，不附加结束符'\0'，而c\_str()返回一个以‘\0’结尾的字符数组。

2、const char \*c\_str();

c\_str()函数返回一个指向正规C字符串的指针,内容与本string串相同.

这是为了与c语言兼容,在c语言中没有string类型,故必须通过string类对象的成员函数c\_str()把string 对象转换成c中的字符串样式.

注意：一定要使用strcpy()函数 等来操作方法c\_str()返回的指针

比如：最好不要这样:

char\* c;

string s="1234";

c = s.c\_str(); //c最后指向的内容是垃圾,因为s对象被析构,其内容被处理

应该这样用：

char c[20];

string s="1234";

strcpy(c,s.c\_str());

这样才不会出错,c\_str()返回的是一个临时指针,不能对其进行操作

再举个例子

c\_str() 以 char\* 形式传回 string 内含字符串

如果一个函数要求char\*参数,可以使用c\_str()方法：

string s = "Hello World!";

printf("%s",s.c\_str()); //输出 "Hello World!"

char \*转换成string

可以直接赋值。

string s;

char \*p = "adghrtyh";

s = p;

a) string s; //生成一个空字符串s

b) string s(str) //拷贝构造函数 生成str的复制品

c) string s(str,stridx) //将字符串str内“始于位置stridx”的部分当作字符串的初值

d) string s(str,stridx,strlen) //将字符串str内“始于stridx且长度顶多strlen”的部分作为字符串的初值

e) string s(cstr) //将C字符串作为s的初值

f) string s(chars,chars\_len) //将C字符串前chars\_len个字符作为字符串s的初值。

g) string s(num,c) //生成一个字符串，包含num个c字符

h) string s(beg,end) //以区间beg;end(不包含end)内的字符作为字符串s的初值

i) s.~string() //销毁所有字符，释放内存

FILE \*fp = nullptr; //文件操作

if (fopen\_s(&fp, fileName, "r") != 0)

{ cout << " 文件打开失败 ";

}

while ((feof(fp)) == 0)

{ char str[1000];

char\* ad = nullptr;

char\* bk = nullptr;

fgets(str, 1000, fp);

str[strlen(str) - 1] = '\0';

ad = strtok\_s(str, "%[.][?][!][\n]", &bk); while (ad != NULL)

{

push(ad);

ad = strtok\_s(NULL, "%[.][?][!][\n]", &bk);

}

　　sscanf() - 从一个字符串中读进与指定格式相符的数据.

　　函数原型:

　　Int sscanf( string str, string fmt, mixed var1, mixed var2 ... );

　　int scanf( const char \*format [,argument]... );

　　说明：

　　sscanf与scanf类似，都是用于输入的，只是后者以屏幕(stdin)为输入源，前者以固定字符串为输入源。

　　其中的format可以是一个或多个 {%[\*] [width] [{h | l | I64 | L}]type | ' ' | '\t' | '\n' | 非%符号}

　　注：

　　1、 \* 亦可用于格式中, (即 %\*d 和 %\*s) 加了星号 (\*) 表示跳过此数据不读入. (也就是不把此数据读入参数中)

　　2、{a|b|c}表示a,b,c中选一，[d],表示可以有d也可以没有d。

　　3、width表示读取宽度。

　　4、{h | l | I64 | L}:参数的size,通常h表示单字节size，I表示2字节 size,L表示4字节size(double例外),l64表示8字节size。

　　5、type :这就很多了，就是%s,%d之类。

　　6、特别的：%\*[width] [{h | l | I64 | L}]type 表示满足该条件的被过滤掉，不会向目标参数中写入值

　　支持集合操作：

　　%[a-z] 表示匹配a到z中任意字符，贪婪性(尽可能多的匹配)

　　%[aB'] 匹配a、B、'中一员，贪婪性

%[^a] 匹配非a的任意字符，贪婪性

sscanf("iios/12DDWDFF@122", "%\*[^/]/%[^@]", buf);

　　printf("%s\n", buf);

　　结果为：12DDWDFF

深拷贝浅拷贝区别

简单的来说就是，在有指针的情况下，浅拷贝只是增加了一个指针指向已经存在的内存，而深拷贝就是增加一个指针并且申请一个新的内存，使这个增加的指针指向这个新的内存，采用深拷贝的情况下，释放内存的时候就不会出现在浅拷贝时重复释放同一内存的错误！

我列举一个例子来说吧：

你正在编写C++程序中有时用到，操作符的重载。最能体现深层拷贝与浅层拷贝的，就是‘=’的重载。

看下面一个简单的程序：

class string

{

char \*m\_str;

public:

string(char \*s）

{

m\_str=s;

}

string()

{};

String & operator=(const string s)

{

m\_str=s.m\_str;

return \*this

}

};

int main()

{

string s1("abc"),s2;

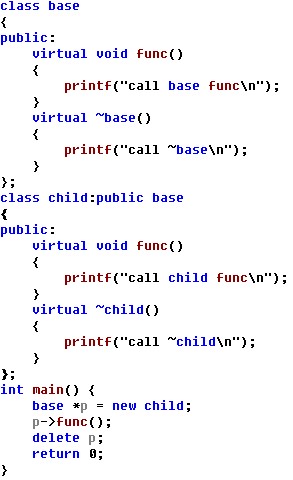
s2=s1;

cout<<s2.m\_str;

}

上面的 =重载其是就是实现了浅拷贝原因。是由于对象之中含有指针数据类型.s1,s2恰好指向同一各内存。所以是浅拷贝。而你如果修改一下原来的程序：

string&operator=(const string&s)

{

if(strlen(m\_str)!=strlen(s.m\_str))

m\_str=new char[strlen(s.m\_str)+1];

if(\*this!=s)

strcopy(m\_str,s.m\_str);

return \*this;

}

这样你就实现了深拷贝，原因是你为被赋值对象申请了一个新的内存所以就是深拷贝。

//类封装

#include <iostream> #include <string>

using namespace std;

/\*\*

\* 定义类：Student

\* 数据成员：m\_strName

\* 无参构造函数：Student()

\* 有参构造函数：Student(string \_name)

\* 拷贝构造函数：Student(const Student& stu)

\* 析构函数：~Student()

\* 数据成员函数：setName(string \_name)、getName()

\*/

class Student

{

public:

Student()

{

}

Student(string \_name)

{

}

Student(const Student& stu)

{

}

~Student()

{

}

void setName(string \_name)

{

m\_strName=\_name;

}

string getName()

{

return m\_strName;

}

private:

string m\_strName;

};

int main(void)

{

// 通过new方式实例化对象\*stu

Student \*stu = new Student();

// 更改对象的数据成员为“慕课网”

stu->setName("慕课网");

// 打印对象的数据成员

cout<<stu->getName()<<endl;

return 0;

}

1.类的一个特征就是封装，public和private作用就是实现这一目的。所以：

用户代码（类外）可以访问public成员而不能访问private成员；private成员只能由类成员（类内）和友元访问。

2.类的另一个特征就是继承，protected的作用就是实现这一目的。所以：

protected成员可以被派生类对象访问，不能被用户代码（类外）访问。

多态

什么是多态？

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

答案：call child func

call ~child

call ~base

多态的实现方式分析？

实现多态，有二种方式，覆盖，重载。覆盖：是指子类重新定义父类的虚函数的做法。重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。