1.数据类型

#include <iostream>

using namespace std;

//1.2.1#define定义宏常量后不能修改;在最上方,无需加上数据类型

#define nianlin 7

int main()//一个项目只能有一个main

{

cout << "YueYiFan" << endl;//是L不是一

/\*多

行

注

释\*/

//1.1变量创建的方法

int a = 9;//a是标识符，10是变量

cout << "a=" << a << endl;

cout << "你" << nianlin << "岁" << endl;

//1.2.2const修饰的变量,修饰后即为常量不能更改

const int month = 12;

cout << "现在已经" << month << "月了";

//1.4标识符；尽量让标识符能代表变量意义

int \_1d = 30;

int \_1D = 40;

//1.4不能int1D

cout << \_1d << \_1D << endl;

//2.1整型;

long b = 9;

short c = 9;

long long d = 9;

//2.2sizeof

cout << "short占用的内存空间为" << sizeof(short) << endl;

cout << "b占用的内存空间为" << sizeof(b) << endl;

//2.3实型(默认小数是双精度double;故要在数字后加上f以表示该数是float)

float f1 = 3.14159268f;

double d1 = 3.1415926;

//2.3科学计数法输出

float f2 = 4e-2;//4的-2次方，输出0.04

cout << f2 << endl;

//2.3最多显示六位有效数字

cout << f1 << endl;

cout << d1 << endl;

//2.3查看占用的内存空间

cout << sizeof(f1) << endl;//float4个

cout << sizeof(d1) << endl;//doule8个

//2.4字符型

char ch = 'Z';//注意创建字符型一定要是！单引号！；单引号内只能有一个字符

cout << ch << endl;

//查看字符型的ASCII编码

cout << (int)ch << endl;//a是97,z是122,A是65，Z是90

//2.5转义字符

cout << "\\" << endl;//要两个\\输出一个

cout << "aaa\tYYF" << endl;//\t可以使语句对齐

cout << "aaaaaa\tYYF" << endl;

//2.6字符串型

//①C风格字符串

char str[] = "YUEYIFAN";//等号左标识符名后有[]；等号后双引号（与字符型不同）

cout << str << endl;

//②C++风格字符串

string str2 = "YUEYIFAN";

cout << str2 << endl;

//2.6布尔数据类型bool

bool flag = true;//bool定义数据结构类型为bool

cout << flag << endl;//输出1表示true实质上是1

flag = false;

cout << flag << endl;//即使不用bool开头定义，0也代表假

cout << sizeof(bool) << endl;//bool所占内存空间为1

//2.8数据的输入

//2.8.1输入整型

int m = 0;

cout << "给m赋值" << endl;

cin >> m;

cout << "整型变量m=" << m << endl;

//2.8.2输入实型（浮点型）

float n = 3.14f;

cout << "给n赋值" << endl;

cin >> n;

cout << "浮点型变量n=" << n << endl;

//2.8.3输入字符型

char cha = 'a';

cout << "给字符型变量cha赋值" << endl;

cin >> cha;

cout << "字符型变量cha=" << cha << endl;

//2.8.4字符串型

string strin = "aha";

cout << "给字符串赋值" << endl;

cin >> strin;

cout << "字符串型变量strin=" << strin << endl;

//2.8.5布尔型

bool boo = true;

cout << "给bool赋值" << endl;

cin >> boo;

cout << "bool值为" << boo << endl;

system("pause");

return 0;

}

//1.1变量：用于快速读取内存；；不要在同一段函数中重复定义一个变量

//1.2常量：用于记录程序中不可更改的数据

//1.2.1#define宏常量，写在最上；1.2.2或者用const加在变量前使该变量变为常量

//1.3关键字，C++中预先保留的单词，不能用它定义常变量63个:int if return 等

//1.4标识符命名规则:1.不可以是关键字；2.仅能由①字母②数字③下划线组成；3.第一个字符必须为字母或者下划线（第一个字符不能是数字）4.区分大小写

// 2.数据类型：给变量分配一个合适的内存空间

// 2.1整型：①int4字节+-2^31②short短整型2字节+-2^15③long长整型4字节+-2^31④long long长整型8字节+-2^31

// 2.1当超出最大范围出错；编译器下方一小块会变红；short<int<=long<=long long

// 2.2sizeof关键字：统计数据类型所占内存大小

// 2.3实型：表示小数①单精度float(4个字节，7位有效数字)②双精度double（8字节，15-16位有效数字）

// float和numpy中不同

// 2.4字符型char balabala='a';(单引号内只有！！一！！个字符）；占用一个字节

// 2.4字符型变量是转化成ASCII编码放入存储单元

// 2.5转义字符：表示不能显示出来的ASCII字符；

// \n换行，\\反斜杠,\t水平制表符

// 2.6字符串型:①C风格②C++风格

// 2.7布尔类型：真1或者假0

// 2.8数据的输入cin

2.运算符

#include <iostream>

using namespace std;

int main()

{

//3.1.1加减乘除

int a1 = 10;

int b1 = 9;

cout << a1 + b1 << endl;

cout << a1 - b1 << endl;

cout << a1 \* b1 << endl;

cout << a1 / b1 << endl;//两个整数（int）相除结果依然是整数，会自动去除小数部分

//被除数为0时报错

//两小数相除结果可以是小数

double d1 = 0.5;

double d2 = 0.28;

cout << d1 / d2 << endl;

//3.1.2取模；被除数一样不能为0否则报错

cout << a1 % b1 << endl;//得到10%9=1

cout << b1 % a1 << endl;//得到9%10=9

//有小数不能做取模运算

//3.1.3递增递减

int a = 10;

++a;//前置递增，使a=a+1

int a2 = ++a \* 10;//前置运算是先使a=a+1=11+1=12,再进行其它运算

cout << "a2=" << a2 << endl;

int b = 10;

b++;//后置递增，使得b=b+1

int b2 = b++ \* 10;//后置运算是先运行其他表达式，再使b=b+1=11+1

cout << "b2=" << b2 << endl;

int c = 10;

--c;//前置递减

int c2 = --c \* 10;//先让c=c-1,再c\*10

cout << "c2=" << c2<<c << endl;

int e = 10;

e--;

int e2 = e-- \* 10;//先让e\*10，再让e=e-1

cout << "e2=" <<e2<<e<< endl;

//赋值运算符

int g = 10;

g = 100;//将100赋值给g

cout << "g=" << g << endl;

g -= 10;//g=g-10

cout << "g=" << g << endl;

g += 9;//g=g+9

cout << "g=" << g << endl;

g /= 9;//g=g/9

cout << "g=" << g << endl;

g \*= 2;//g=g\*2

cout << "g=" << g << endl;

g %= 3;//g=g%3

cout << "g=" << g << endl;

//3.3比较运算符

cout << (e == g) << endl;//表明e和g相等不成立，返回0

cout << (e != g) << endl;//e确实不等于g，返回1

cout << (e > g) << endl;//e比g大成立，返回1

cout << (e < g) << endl;//e比g小不成立，返回0

cout << (e >= g) << endl;

cout << (e <= g) << endl;

//3.4逻辑运算符g=1 e=8

cout << !e << endl;//c++中除了0都是真

cout << !!e << endl;//两次取反又变为真

cout << (e && g) << endl;//全是真1才是真1

cout << (!e && g) << endl;//有一个不是真1就是假0

cout << (!e || !g) << endl;//全不为真才是假

cout << (!e || g) << endl;//有一个为真就是真

system("pause");

return 0;

}

//3.运算符

//3.1算术运算符

//3.1.1+-\*/

//3.1.2取模运算%：求余数10%3=1

//3.1.3递增递减运算符：++前置(先加1再做别的运算)、后置++（先运算再加1）

//3.2赋值运算= += -= \*= /= %=

//3.3比较运算符：== != < > <= >=;输出结果为0（假）或1（真）

//3.4逻辑运算符：非! 与&& 或||

3.1if循环练习

#include <iostream>

using namespace std;

//练习

int main()

{

int pig1 = 0;

int pig2 = 0;

int pig3 = 0;

cout << "pig1's weight" << endl;

cin >> pig1;

cout << "pig2's weight" << endl;

cin >> pig2;

cout << "pig3's weight" << endl;

cin >> pig3;

if (pig1 >= pig2)

{

if (pig1 >= pig3)

{

cout << "pig1 is the fattest" << endl;

}

else

{

cout << "pig3 is the fattest" << endl;

}

}

else

{

if (pig3 >= pig2)

{

cout << "pig3 is the fattest" << endl;

}

else

{

cout << "pig2 is the fattest" << endl;

}

}

system("pause");

return 0;

}

3.if\_switch选择结构

#include <iostream>

using namespace std;

int main()

{

//4.1.1 if语句

int score = 0;

cout << "分数" << endl;

cin >> score;

cout << "您输入的分数" << score<<endl;

//if条件后不要加分号冒号，和python不一样

if (score > 60)

{

cout << "passed" << endl;

}

else

{

cout << "failed" << endl;

}

//多条件if语句

//一定不要区间性判断（0<=score<=60等），python可以，C++不行，分成几个else if来判断即可

int score2 = 0;

cout << "分数" << endl;

cin >> score2;

cout << "您的分数" <<score2<< endl;

if (score2>100)

{

cout << "wrong" << endl;

}

else if ( score2 <0)

{

cout << "wrong" << endl;

}

else if (score2>=60)

{

cout << "passed" << endl;

}

else

{

cout << "failed" << endl;

}

//4.1.2三目运算符

int a = 10;

int b = 1;

int c = 0;

c = (a > b ? a : b) ;

cout << "c="<< c << endl;//返回较大的a

(a > b ? a : b) = 100;//将100赋值给较大的a

cout << "a=" << a << endl;

//4.1.3switch语句（电影打分）//不能判断区间但是执行效率高

cout << "电影打分" << endl;

int fenshu = 0;

cin >> fenshu;

cout << "您的打分" << fenshu << endl;

switch (fenshu)

{

case 10://case后只能是整型或者字符型

cout << "经典电影" << endl;

break;//符合该条件就推出当前分支

case 9:

cout << "经典电影" << endl;

break;

case 8:

cout << "不错的电影" << endl;

break;

case 7:

cout << "不错的电影" << endl;

break;

case 6:

cout << "一般电影" << endl;

break;

case 5:

cout << "一般电影" << endl;

break;

default:

cout << "烂片" << endl;

break;

}

system("pause");

return 0;

}

//4.程序流程结构：①顺序结构②选择结构③循环结构

//4.1选择结构

//4.1.1 if语句：条件后不要加分号冒号

//可以只有if没有else；没有else就是单行if语句;多条件if 、else if、else if、else

//4.1.2三目运算符;可以带上括号继续赋值； a>b?a:b 如果a>b返回变量a，否则返回变量b

//4.1.3 switch语句:switch(case){case 整数/字符型: break;case: break; default: break;}

4.1do while循环

//4.程序流程结构

//4.2.2do while循环（与while的不同在于while先判断是否符合循环条件在执行，dowhile先执行再判断）

#include <iostream>

using namespace std;

int main()

{

int num = 0;//0在C++中是false

do

{

cout << num << endl;

num++;

} while (num < 10);

//练习：水仙花数：一个三位数，每个位数上的数字的3次幂之和等于它本身

int num1 = 100;

do

{

int a = 0;//很重要：要先初始化再赋值

int b = 0;//很重要：要先初始化再赋值

int c = 0;//很重要：要先初始化再赋值

a = num1 /100;

b = num1 / 10 % 10;

c = num1 % 10;

if(a\*a\*a+b\*b\*b+c\*c\*c==num1)

{

cout << num1 << endl;

}

num1++;

} while (num1 < 1000);

system("pause");

return 0;

}

4.while循环

//4.2循环结构

//4.2.1 while循环

#include <iostream>

using namespace std;

#include <ctime>//使用time需要加此头文件

int main()

{

int num = 0;

while (num < 10)//括号内写循环条件；避免死循环的出现

{

cout << num << endl;

num++;

}

//4.2.1练习猜数字

// 添加随机数种子，用系统时间生成随机数，防止每次随机数都一样

srand((unsigned int)time(NULL));//不加这行生成的是伪随机数

//生成随机数

int num1 = rand() % 10 + 1;//rand()%10生成0-9之间的随机数

//玩家输入数

int wan = 0;

while (1)

{

cin >> wan;

if (wan > num1)

{

cout << "猜大了" << endl;

}

else if (wan < num1)

{

cout << "猜小了" << endl;

}

else

{

cout << "猜对啦" << endl;

break;//可以利用break退出当前循环

}

}

system("pause");

return 0;

}

5.for循环

//4.程序流程结构

// 4.2.3for循环语句(起始表达式；条件表达式；末尾循环体){循环语句}

//

//

//

#include <iostream>

using namespace std;

int main()

{

//for循环分号分隔

for (int i = 0; i < 10; i++)//也可以括号内就写两个分号，把三个条件分开写

{

cout << i << endl;

}

//敲桌子练习

for (int i = 1; i < 101; i++)

{

if (i % 7 == 0)

{

cout << "敲桌子" << i <<endl;

}

else if (i % 10 == 7)

{

cout << "敲桌子" << i << endl;

}

else if (i / 10 % 10 == 7)

{

cout << "敲桌子" << i << endl;

}

else

{

cout << i << endl;

}

}

system("pause");

return 0;

}

6.嵌套循环

//4.2嵌套循环

//在循环中嵌套一层循环

#include <iostream>

using namespace std;

int main()

{

//打出星图10\*10

//外层循环一次，内层循环一周

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

{

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

{

cout << "\* ";//没有endl不换行

}

cout << endl;//endl实际是换行的意思

}

//乘法口诀表

for (int i = 1; i < 10; i++)

{

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

{

cout << i << "\*" << j << "=" << i \* j<<" ";

}

cout << endl;

}

system("pause");

return 0;

}

7.跳转语句

//4.3跳转语句

//4.3.1break语句：跳出选择结构或循环结构

//break语句的三个使用时机

//①出现在switch语句中，作用是终止case并跳出switch

//②出现在循环语句中，作用是跳出循环语句

//③出现在嵌套循环中，跳出最内层的循环语句

//4.3.2continue语句

//作用：在循环语句中，跳过本次循环中余下尚未执行的语句，继续执行下一次循环；break是退出，continue是跳过

//4.3.3goto语句：无条件跳转语句

#include <iostream>

using namespace std;

int main()

{

//①在switch语句中，作用是终止case并跳出switch

//副本难度选择

cout << "选择副本难度" << "1.hard" << "2.median" << "3.easy" << endl;

int select = 0;

cin >> select;

switch (select)

{

case 1:

cout << "困难" << endl;

break;

case 2:

cout << "中等" << endl;

break;

case 3:

cout << "简单" << endl;

break;

default:

break;

}

//②出现在循环语句中，作用是跳出循环语句

//打印十个数，打印到5就退出

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

{

if (i == 5)

{

break;

}

cout << i << endl;

}

//③出现在嵌套循环中，跳出最内层的循环语句

//输出一半的星星

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

{

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

{

if (j > i)

{

break;

}

cout << "\*" ;

}

cout << endl;

}

//4.3.2continue

for (int i = 0; i <= 100; i++)

{

if (i % 2 == 0)

{

continue;//i为2的倍数的时候，continue跳出此循环，进入下一次循环；break则到0循环就退出了

}

cout << i << endl;

}

//4.3.3goto:跳转到标记处；不推荐使用

cout << "1" << endl;

goto FLAG;

cout << "2" << endl;

cout << "3" << endl;

cout << "4" << endl;

FLAG:

cout << "5" << endl;

system("pause");

return 0;

}

8.1二维数组

//5.2二维数组:4种定义方式

// 二维数组的数组名：①查看二维数组所占内存空间；②获取二维数组首地址

#include <iostream>

using namespace std;

int main()

{

////①第一种定义方式

//int arr[2][3];//定义一个两行三列的数组

//arr[0][0] = 1;

//arr[0][1] = 2;

//arr[0][2] = 3;

//arr[1][0] = 4;

//arr[1][1] = 5;

//arr[1][2] = 6;

////遍历每个元素；嵌套循环打印矩阵和python不同

//for (int i = 0; i < 2; i++)

//{

// for (int j = 0; j < 3; j++)

// {

// cout << arr[i][j]<<" ";

// }

// cout << endl;

//}

////推荐②第二种定义方式;和python不同，这里都是大括号，没有方括号

//int arr2[2][3] =

//{

// {1,2,3},

// {4,5,6}

//};

////遍历每个元素；嵌套循环打印矩阵和python不同

//for (int i = 0; i < 2; i++)

//{

// for (int j = 0; j < 3; j++)

// {

// cout << arr2[i][j] << " ";

// }

// cout << endl;

//}

////③第三种定义方式（第二种方式的基础上省去括号）

//int arr3[2][3] = { 1,2,3,4,5,6 };//电脑可以自行分行

////遍历每个元素；嵌套循环打印矩阵和python不同

//for (int i = 0; i < 2; i++)

//{

// for (int j = 0; j < 3; j++)

// {

// cout << arr3[i][j] << " ";

// }

// cout << endl;

//}

////④第四种定义方式（省去行数或者列数）

//int arr4[][3] = { 1,2,3,4,5,6 };

//for (int i = 0; i < 2; i++)

//{

// for (int j = 0; j < 3; j++)

// {

// cout << arr3[i][j] << " ";

// }

// cout << endl;

//}

//cout << "二维数组占用的内存空间" << sizeof(arr) << endl;//每个int占4个字节

//cout << "二维数组第一行占用的内存空间" << sizeof(arr[0]) << endl;//第一行占用12个字节

//

////与python不同，不能用arr[][0]来查看第一列

//cout << "二维数组的行数" << sizeof(arr) / sizeof(arr[0]) << endl;

//cout << "二维数组列数" << sizeof(arr[0]) / sizeof(arr[0][0]) << endl;

//

////数组的内存首地址

//cout << "二维数组内存首地址" <<arr<< endl;

//cout << "二维数组第一个行地址" << arr[0] << endl;

//cout << "二维数组第一个元素的首地址" << &arr[0][0] << endl;//要打&arr[][]

//练习：考试成绩：输出三名同学语数英总成绩：列为语数英成绩，行为每个人的成绩

int arr5[3][3] =

{

{100,100,100},

{90,50,100},

{60,70,80}

};

//添加人名

string names[3] = { "张三","李四","王五" };

for (int i = 0; i < 3; i++)

{

int sum = 0;

for (int j = 0; j < 3; j ++ )

{

sum += arr5[i][j];

}

cout << names[i] << "总分" << sum << endl;

}

system("pause");

return 0;

}

8.数组

//5.数组：一个集合，里面存放了a.相同类型的数据元素；数组是由b.连续的内存位置组成的。

//5.1一维数组

//一维数组数据名的用途：1.可以统计整个数组在内存中的长度sizeof。2.可以获取数组在内存中的首个地址。

//数组名是常量不可以进行赋值

//5.1.2冒泡排序

#include <iostream>

using namespace std;

int main()

{

//第一种方法

int arr[5];//定义一个整型的长度为5的数组,先不写元素

arr[0] = 10;//定义数组中每个元素的值，下标从0开始

arr[1] = 20;

arr[2] = 30;

arr[3] = 40;

arr[4] = 50;

//第二种方法

int arr2[5] = { 10,20,30,40,50 };//如果赋值没有赋满，会用0来填补剩余数据

//利用循环输出数组中的元素

for (int i = 0; i < 5; i++)

{

cout << arr2[i] << endl;

}

//第三种方法

int arr3[] = { 90,80,70,60,50,40,30,20,10 };//不写数组长度，但写出所有的元素

//利用循环输出数组中的元素

for (int i = 0; i < 9; i++)

{

cout << arr3[i] << endl;

}

cout<<sizeof(arr)<<endl;//查看数组在内存中的空间

cout << sizeof(arr[0]) << endl;//查看单个元素占用空间

cout << sizeof(arr) / sizeof(arr[0]) << endl;//查看数组长度

cout << arr << endl;//查看数组在内存中的首地址（16进制）

cout << (int)arr << endl;//查看数组在内存中的首地址（10进制）

cout << (int)&arr[0] << endl;//查看第一个元素内存的首地址（和数组首地址是一样的）

cout << (int)&arr[1] << endl;//数组的每个元素之间内存相差4

//练习：5只小猪称体重,输出最终的体重

int arr4[5];

for (int i = 0; i < 5; i++)//一定不要忘记定义int i；

{

cin >> arr4[i];

}

int max = arr4[0];

for (int j = 0; j < 5; j++)

{

if (arr4[j] > max)

{

max = arr4[j];

}

}

cout << max << endl;

//练习2：数组逆置;要设置一个中间变量temp

int arr5[5];

for (int i = 0; i < 5; i++)

{

cin >> arr5[i];

}

int start = 0;

int end = sizeof(arr5) / sizeof(arr5[0])-1;

int sum = sizeof(arr5) / sizeof(arr5[0]) ;

while (start < end)

{

int temp = arr5[start];

arr5[start] = arr5[end];

arr5[end] = temp;

start++;

end--;

}

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

{

cout << arr5[i] << endl;

}

//5.1.2冒泡排序（最简单的排序算法）

//升序排列

//1.先找到最大的数：比较两个相邻的元素，如果第一个比第二个大，则交换它们；

//2.对数组所有的元素进行①中比较，此时最后一个数就是最大值。

//3.重复以上步骤，直到所有元素从小到大排序为止

int arr6[8];//对8个数的数组升序排列

//先输入8个数

for (int i = 0; i < 8; i++)

{

cin >> arr6[i];

}

//冒泡排序需要排8-1=7次

for (int m = 7; m > 0; m--)

{

for (int j = 0; j < m; j++)//从第一个元素开始和相邻元素比较，大的放在后面

{

if (arr6[j] > arr6[j + 1])

{

int temp = arr6[j];

arr6[j] = arr6[j + 1];

arr6[j + 1] = temp;

}

}

}

for (int k = 0; k < 8 ; k++)

{

cout << arr6[k] << endl;

}

system("pause");

return 0;

}

9.函数

头文件

#include <iostream>

using namespace std;

// 6.6 函数的分文件编写（4个步骤）

//①创建.h后缀名的头文件

//②创建.cpp后缀名的源文件

//③在头文件中写函数的声明void swap(int a, int b);

//④在原文件中写函数的定义

//⑤在.h中写上 #include <iostream> using namespace std;

//⑥在.cpp中写上#include "fenwenjian.h"；引号表示自定义的函数

//⑦在需要用到此函数的cpp文件中写上#include "fenwenjian.h"

void swap(int a, int b);

分文件cpp

#include "fenwenjian.h"//

// 6.6 函数的分文件编写（4个步骤）

//①创建.h后缀名的头文件

//②创建.cpp后缀名的源文件

//③在头文件中写函数的声明void swap(int a, int b);

//④在原文件中写函数的定义

//⑤在.h中写上 #include <iostream> using namespace std;

//⑥在.cpp中写上#include "fenwenjian.h"；引号表示自定义的函数

//⑦在需要用到此函数的cpp文件中写上#include "fenwenjian.h"

void swap(int a, int b)

{

int temp = a;

a = b;

b = temp;

cout << "a=" << a << endl;

cout << "b=" << b << endl;

}

Main文件cpp

//6.函数：将经常使用的代码封装起来，减少重复代码

//6.1函数定义的五个步骤：①返回值类型②函数名③参数列表④函数体语句⑤return表达式

//返回值类型 函数名（参数列表）

//{

// 函数体语句

// return表达式

// }

// 6.2调用函数

// 6.3值传递：值传递时，如果形参改变不影响实参

// 6.4函数的常见样式4种：有无参数、有无返回值

// 6.5函数的声明：可以多次，告诉编译器如何使用(当该函数写在main函数后面时需要用到)

// 6.6 函数的分文件编写（4个步骤）

//①创建.h后缀名的头文件

//②创建.cpp后缀名的源文件

//③在头文件中写函数的声明void swap(int a, int b);

//④在原文件中写函数的定义

//⑤在.h中写上 #include <iostream> using namespace std;

//⑥在.cpp中写上#include "fenwenjian.h"；引号表示自定义的函数

//⑦在需要用到此函数的cpp文件中写上#include "fenwenjian.h"

//6.练习：实现一个加法函数，功能是传入两个整型数据，计算数据相加结果，并且返回

#include <iostream>

using namespace std;

#include "fenwenjian.h"

//num1 num2是形参

int add(int num1, int num2)//①返回值类型是int②函数名是add③参数列表：传入两个int

{

int sum = num1 + num2;//④传入函数体语句

return sum;//⑤return表达式

}

//定义数值交换函数，如果函数不需要返回值，返回值类型写为void,不写return

void exchan(int numb1, int numb2)

{

cout << "numb1=" << numb1 << endl;

cout << "numb2=" << numb2 << endl;

int temp = numb1;

numb1 = numb2;

numb2 = temp;

cout << "numb1=" << numb1 << endl;

cout << "numb2=" << numb2 << endl;

}

//6.4函数常见样式

//①无参() 无返void

void test1()

{

cout << "①无参无返" << endl;

}

//②有参（。。。）无返void

void test2(int a)

{

cout << "②有参无返 " << a << endl;

}

//③无参() 有返return

int test3()

{

return 0;

}

//④有参（。。。）有返return

int test4(int num4)

{

return num4;

}

//6.5函数的声明:提前告诉编译器函数存在

int max(int a, int b);//函数的声明；没有大括号

int main()//不需要传入东西

{

//6.2在main函数中调用add函数

int a = 90;

int b = 30;

int c = add(a, b);//调用时实参的值会传递给形参

cout << c << endl;

//6.3值传递，交换后d和e并不会改变；实参和形参和temp占用不同的内存

int d = 30;

int e = 49;

exchan(d, e);

cout << "d" << d << endl;//d和e的值不会改变

cout << "e" << e << endl;

test1();//6.4①调用无参无返的函数

test2(2);//6.4②调用有参无返的函数

int q = test3();//6.4③调用无参有返的函数

cout << q << endl;

int p = test4(2000);//6.4④调用有参有返的函数

cout << p << endl;

//6.5函数的声明

int a1 = 20;

int b1 = 10;

cout << max(a1, b1) << endl;

//6.6分文件

swap(10, 20);

system("pause");

return 0;

}

//6.5函数的声明练习：比较两数大小返回大的

int max(int a, int b)

{

return a > b ? 1 : 2;//三目运算符比大小；a大返回1，否则返回2

}

10.指针

//7.指针：可以通过指针间接访问内存

//编号是从0开始记录的，一般用16进制数字表示

//可以利用 指针变量 保存地址；

//7.3指针占用的内存空间大小：在32位操作系统下指针占4个字节；64位占8个

//7.4.1空指针：指针变量指向内存编号为0的空间；用于初始化指针变量；空指针指向的内存不可访问（不能用\*p访问）

//7.4.2野指针：指针指向非法的内存空间

//空指针和野指针都不是我们申请的空间，因此不要访问。

//7.5const修饰指针：①修饰常量②修饰指针③修饰常量和指针

#include <iostream>

using namespace std;

int main()

{

int a = 10;

//1.①指针定义的语法

int\* p;

//1.②让指针记录变量a的地址：指针就是变量的地址

p = &a;

cout << "a的地址为" << &a << endl;

cout << "a的地址为" << p << endl;

//2.使用指针

\*p = 100;//使用\*p改变对应内存的变量

cout << "p指向内存中的变量\*p=" << \*p << endl;

cout << "变量a的值" << a << endl;

//3.查看指针占内存变量大小//因为占8个字节，电脑是64位操作系统；x86是32位

cout << "sizeof (int\*)=" << sizeof(int\*) << endl;//不管指针是int double float 还是char指针占的字节是一样的

cout << "sizeof (int\*)=" << sizeof(float\*) << endl;

cout << "sizeof (int\*)=" << sizeof(double\*) << endl;

cout << "sizeof (int\*)=" << sizeof(char\*) << endl;

//7.4.1空指针

int\* n = NULL;//定义空指针

//不能用\*n进行方位或修改指针对应的变量；0~255之间的内存编号是系统占用的，因此不可以访问

//7.4.2野指针:避免出现野指针

//int\* y = (int\*)0x1100;//这就属于野指针

//7.5.1常量修饰：指针指向的值不可以改（不能\*p=200）,指针的指向可以改（p=ab）

const int\* c = &a;//const在int前值不能改

//7.5.2指针修饰：（\*p=20可以，但不能p=ab）可以改变指针指向的值，但不能改变指针指向的位置

int\* const d = &a;//const在指针前指针方向不能改。

//7.5.3即修饰指针也修饰常量

const int\* const e = &a;

//7.5总结：const在哪里，后面的变量就不能变了

system("pause");

return 0;

}

10.1指针和数组

#include <iostream>

using namespace std;

//7.7指针和函数

//先创建一个交换函数；改变的是指针指向的值不是指针指向的方向

void swap02(int \*p1,int \*p2)

{

int temp = \*p1;

\*p1 = \*p2;

\*p2 = temp;

}

//练习：函数创建冒泡排序

void bubble(int\* arr, int len)//只能用指针的方式导入数组

{

for (int i = 0; i < len - 1; i++)

{

for (int j = 0; j < len - i - 1; j++)

{

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

{

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

void printArray(int \* arr, int len)//只能用指针的方式导入数组

{

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

{

cout << arr[i] << " ";

}

}

int main()

{

//7.6指针和数组

//利用指针访问数组中的元素

int arr[10] = { 9,8,5,3,5,6,7,4,2,1 };

cout << "第一个元素为：" << arr[0] << endl;

int\* p = arr;//指针p指向arr数组首地址

cout << "利用指针访问第一个元素： " << \*p << endl;

//利用循环访问指针指向的每个元素

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

{

cout << "利用指针访问第" << i + 1 << "个元素" << \*p << endl;

p++;

}

//7.7指针和函数

// 值传递改变形参不改变实参

//地址传递;地址传递可以修饰实参，但实参的地址没有改变

int a = 10;

int b = 20;

cout << "&a= " << &a << endl;

cout << "&b= " << &b << endl;

swap02(&a, &b);

cout << "a=" << a << endl;

cout << "b=" << b << endl;

cout << "&a= " << &a << endl;//a和b的内存地址没有变

cout << "&b= " << &b << endl;

//练习：对数组进行升序排列（冒泡）

int arr1[10] = { 10,9,7,5,8,4,3,6,1,2 };

bubble(arr1, 10);//只能用指针的方式导入数组到函数，但直接输入数组名就可以

printArray(arr1, 10);

system("pause");

return 0;

}

11.结构体

//8结构体：用户自定义的数据类型，允许用户存储不同的数据类型

//8.1结构体定义及使用

#include <iostream>

using namespace std;

//自定义数据类型：学生 包括（姓名、年龄和分数）

struct Student//定义数据类型struct不能省略

{

string name;

int age;

int score;

};

//通过学生类型创建具体学生信息

int main()

{

//根据自定义数据类型创建数据的三种方法

//①

struct Student s1;//创建数据，这里struct可以省略

s1.name = "张三";

s1.age = 18;

s1.score = 94;

cout << "姓名： " << s1.name << " 年龄： " << s1.age << " 分数：" << s1.score << endl;;

//②

struct Student s2 = { "李四",19,90 };

cout << "姓名： " << s2.name << " 年龄： " << s2.age << " 分数：" << s2.score << endl;;

//③在开头的函数大括号最后写上s3，①中省去第一行，其他一样

system("pause");

return 0;

}

11.1结构体数组

#include <iostream>

using namespace std;

//8.2结构体数组

//8.2.1.定义结构体

struct Student//定义数据类型struct不能省略

{

string name;

int age;

int score;

};

//8.3结构体指针

//通过学生类型创建具体学生信息

int main()

{

//8.2.2.创建结构体数组

struct Student stuArray[3] =

{

{"张三",18,100},

{"李四",28,99},

{"王五",38,66}

};//别忘了分号

//8.2.3.更改结构体数组中的元素；stuArray编号从0开始，2对应的其实是第三个

stuArray[2].name = "赵六";

stuArray[2].age = 80;

stuArray[2].score = 60;

//8.2.4.遍历结构体数组.name

for (int i = 0; i < 3; i++)

{

//太长的时候可以分行但输出结果可以分行

cout << "姓名： " << stuArray[i].name

<< "年龄： " <<stuArray[i].age

<< "分数： " <<stuArray[i].score << endl;

}

//8.3.1创建学生结构体变量

struct Student s = { "张三",18,100 };

//8.3.2通过指针指向结构体变量

Student \* p = &s;

//8.3.3通过指针访问结构体变量中的数据 !!! p->name一定要用->

cout << "姓名：" << p->name << endl;

system("pause");

return 0;

}

11.2嵌套结构体

#include <iostream>

using namespace std;

//8.4嵌套结构体：一个结构体中还有一个结构体

//先定义学生结构体

struct student

{

string name;//姓名

int age;//年龄

int score;//分数

};

//再定义老师结构体

struct teacher

{

int id;

string name;

int age;

struct student stu;//定义辅导的学生

};

int main()

{

//一起定义老师和学生

teacher t;//定义老师

t.id = 1001;

t.name = "wang";

t.age = 50;

t.stu.name = "smallwang";//定义该老师的学生

t.stu.age = 20;

t.stu.score = 90;

cout << "老师的姓名： " << t.name << "老师编号： " << t.id << "老师的学生信息" << t.stu.name << endl;

system("pause");

return 0;

}

11.3结构体做函数参数

//8.5结构体做函数参数 ：

//函数传递方式：①值传递（实参会变）；②地址传递（实参不变）

#include <iostream>

using namespace std;

struct Student

{

string name;

int age;

int score;

};

//建立打印函数

//1.值传递

void prints1(struct Student s1)

{

s1.age = 0;//形参变了但是实参没变

cout << "子函数1 年龄：" << s1.age << endl;

}

//2.地址传递 用->；可以改变实参

void prints2(struct Student \*p)

{

p->age = 200;//形参实参都变成了200

cout << "子函数2 年龄：" << p->age << endl;

}

int main()

{

//结构体做函数参数

//创建结构体变量

struct Student s1;

s1.name = "张三";

s1.age = 18;

s1.score = 94;

prints1(s1);

prints2(&s1);

cout << "年龄" << s1.age << endl;

system("pause");

return 0;

}

11.4结构体中的const

#include <iostream>

using namespace std;

//8.6结构体中const使用场景

struct student

{

string name;

int age;

int score;

};

//值传递打印：占内存大

void prints(student s)

{

cout << "子函数1 年龄：" << s.age << endl;

}

//地址传递占内存小，但会改变实参，为了防止改变实参，在指针前加上const；如果在函数中改变实参会报错

void prints2(const student \* s)

{

cout << "子函数1 年龄：" <<s->age << endl;

}

int main()

{

struct student s = { "san",15,70 };

prints(s);

prints2(&s);

system("pause");

return 0;

}

14 四区：代码区 全局区

//1.内存分区模型：C++执行程序时，将内存大方向划分为4个区域

//代码区；存放程序的二进制代码，由操作系统管理

//全局区;存放局部变量、静态变量以及常量

//栈区；由编译器自动分配释放，存放函数的参数值，局部变量等

//堆区；由程序员分配和释放，若程序员不释放，程序结束时操作系统回收

////内存四驱的意义：不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程

//程序运行前只有 代码区（共享、只读） 全局区（该区域的数据在程序结束后由操作系统释放）

#include <iostream>

using namespace std;

//全局变量

int g\_a = 10;

int g\_b = 10;

//const修饰全局常量

const int c\_g\_a = 10;

const int c\_g\_b = 10;

int main()

{

//不在全局区：局部变量、const修饰的局部变量

//在全局区：全局变量、静态变量、字符串常量、constant修饰的全局常量

//创建普通局部变量

int a = 10;

int b = 10;

cout << "局部变量a的地址为" << (int)&a << endl;

cout << "局部变量b的地址为" << (int)&b << endl;

//全局变量和局部变量不在一起

cout << "全局变量g\_a的地址为" << (int)&g\_a << endl;

cout << "全局变量g\_b的地址为" << (int)&g\_b << endl;

//全局变量 静态变量 字符串常量 constant修饰全局常量 在一个区域中

//静态变量 在普通便两千加static,属于静态变量

static int s\_a = 10;

static int s\_b = 10;

cout << "静态变量s\_a的地址为" << (int)&s\_a << endl;

cout << "静态变量s\_b的地址为" << (int)&s\_b << endl;

//常量

//字符串常量

cout << "字符串常量的地址为：" << (int)&"hello world" << endl;

//const修饰变量

//const修饰的全局常量 和局部变量

cout << "全局常量c\_g\_a的地址为：" << (int)&c\_g\_a << endl;

cout << "全局常量c\_g\_b的地址为：" << (int)&c\_g\_b << endl;

//const修饰局部变量

int c\_l\_a = 10;

int c\_l\_b = 10;

cout << "局部常量c\_l\_a的地址为：" << (int)&c\_l\_a << endl;

cout << "局部常量c\_l\_b的地址为：" << (int)&c\_l\_b << endl;

system("pause");

return 0;

}

15四区：栈区

#include <iostream>

using namespace std;

//栈区数据的注意事项---不要返回局部变量的地址

//栈区的数据由编译器管理开辟和释放

int \* func(int b)//形参数据也会存放在栈区

{

b = 100;

int a = 10;//局部变量存放在栈区，栈区数据在函数执行完后自动释放

return &a;//返回局部变量的地址,不要返回局部变量地址

}

int main()

{

//接收func函数的返回值

int\* p = func(1);

cout << \*p << endl;//打印正确是因为编译器做了保留，不要返回局部变量地址

system("pause");

return 0;

}

16.四区：堆区

#include <iostream>

using namespace std;

//堆区：由程序员分配释放，若程序员不释放，程序结束时由操作系统回收

//C++中主要利用new在堆区开辟内存，手动释放内存用delete

//1.new的基本语法

int\* func()

{

//利用new将数据开辟到堆区

//指针 的本质是局部变量，放在栈上，指针保存到数据放在堆区

//在堆区创建整型数据

//new返回的是该数据类型的指针

int\* p = new int(10);

return p;

}

void test1()

{

int\* p = func();

cout << \*p << endl;

//利用关键词delete释放内存区数据

delete p;

}

//2.在堆区用new开辟数组

void test2()

{

//在堆区创建10整型数据的数组

int\* arr = new int[10];//10代表数组有10个元素 方括号

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

{

arr[i] = i + 100;//给10个元素赋值100-109

}

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

{

cout << arr[i] << endl;

}

//释放堆区数组;释放数组要加[]才可以

delete[] arr;

}

int main()

{

test1();

test2();

int\* p = func();

cout << \*p << endl;

system("pause");

return 0;

}

17.引用

#include <iostream>

using namespace std;

//1.作用：给变量起别名

//数据类型（和原名一样）&别名=原名

//用原名，别名操作的是同一块内存，操纵别名原名也会变

//2.引用的注意事项

//①引用必须要初始化；错误示例 int &b;

//②引用一旦初始化后就不能更改了：不能 b是a的别名，将其改为c的别名；只能将c的值赋给b同时赋给了a

//3.引用做函数参数

//作用：函数传参时，可以利用引用的技术让形参修饰实参（简化指针修改实参）

//交换函数

//①值传递：形参变了实参没变

void exchange1(int c,int d)

{

int temp = c;

c = d;

d= temp;

cout << "exhange1 c=" << c << endl;

cout << "exchange1 d=" << d << endl;

}

//②地址传递：实参形参都改变了

void exchange2(int \*c, int \*d)

{

int temp = \*c;

\*c = \*d;

\*d = temp;

cout << "exhange2 c=" << \*c << endl;

cout << "exchange2 d=" << \*d << endl;

}

//③引用传递:代码和值传递一样，参数不一样；用别名修改，操控的同一块内存

void exchange3(int& c, int& d)

{

int temp = c;

c = d;

d = temp;

cout << "exhange3 c=" << c << endl;

cout << "exchange3 d=" << d << endl;

}

int main()

{

int a = 10;

int &b = a;

cout << "a=" << a << endl;

cout << "b=" << b << endl;

b = 100;//更改别名原名也会变

cout << "a=" << a << endl;

cout << "b=" << b << endl;

//值传递

int c = 10;

int d = 20;

cout << "初始c=" << c << endl;

cout << "初始d=" << d << endl;

exchange1(c, d);//值传递，形参不会修饰实参

cout << "值传递c=" << c << endl;

cout << "值传递d=" << d << endl;

//地址传递：实参会改变

exchange2(&c, &d);

cout << "地址传递c=" << c << endl;

cout << "地址传递d=" << d << endl;

//引用传递：实参会改变

exchange3(c, d);

cout << "引用传递c=" << c << endl;

cout << "引用传递d=" << d << endl;

system("pause");

return 0;

}

18.引用2

#include <iostream>

using namespace std;

//4.引用做函数返回值

//作用：引用是可以作为函数的返回值存在的

//1.注意：不要返回局部变量引用

int &test1()//假如返回局部变量引用

{

int a = 10;//局部变量存放在四区中的栈区

return a;

}

//2.（等号左边是左值）如果函数的返回值是引用，这个函数的调用可以作为左值

int& test2()

{

static int a = 10;//静态变量存放在全局区；程序结束后全局区数据释放

return a;//返回a的引用（变量）

}

//5.引用的本质：指针常量（指针指向不可以修改）

//int a = 10;

//int& ref = a;自动转换为 int\*const ref=&a;指针常量指向不能更改

//ref = 20;内部发现ref是引用，自动转换为\*ref=20;(指针指向的内容变为20)；就把ref当成a即可

//6.常量引用：主要用来修饰形参，防止误操作

//在形参列表中，加const修饰形参，防止形参改变实参

//打印数据的函数

void show(const int& val)//防止误操作改变val

{

//val = 1000;加上const后无法修改val

cout << "val= " << val << endl;

}

int main()

{

int& ref = test1();

cout << "ref=" << ref << endl;//乱码了因为返回了局部变量

int& ref2 = test2();

cout << "ref2=" << ref2 << endl;//静态变量存在全局区，程序结束才会被释放

test2() = 1000;//（使得a=1000,ref2是a的别名）

//如果函数的返回值是引用，这个函数调用可以作为左值

cout << "ref2=" << ref2 << endl;

//6.常量引用

//加上const后，编译器将代码修改 int temp=10;const int &ref3=temp;

const int& ref3 = 10;

//用引用的方式输出a

int a = 100;

show(a);

system("pause");

return 0;

}

19.函数提高

#include <iostream>

using namespace std;

//1.函数默认参数

int func1(int a, int b=20, int c=30)//这样可以只传一个参数；如果我们自己传入数据就用自己的数据，没传入用默认值

{

return a + b + c;

}

//int func2(int a, int b = 20, int c)

//注意1：如果某个位置已经有了默认参数，则此位置以后都必须有默认参数

//{

// return a + b + c;

//}

//注意2：如果函数的声明有默认参数，函数现实就不能有默认参数;否则会报错

// 声明函数和实现只能有一个默认参数

int func3(int a =20 , int b =20);

int func3(int a , int b)

{

return a + b;

}

//2.函数占位参数,这样要传两个参数进去才能正常调用此函数

void func4(int a,int)

{

cout << "this is func" << endl;

}

//3.1函数重载：函数名可以相同，提高复用性

//函数重载的满足条件：1.必须在同一个作用域（全局 局部）下2.函数名称相同3.函数的参数类型不同 或者个数不同 或者顺序不同

//注意函数返回值不可以作为重载的条件（比如void改int）

//3.2函数重载的注意事项：①引用②碰到默认参数

void func5()

{

cout << "func5调用" << endl;

}

void func5(int a)//参数不同,可以重载

{

cout << "func5(int a)的调用" << endl;

}

void func5(double a)//参数不同,可以重载

{

cout << "func5(double a)的调用" << endl;

}

void func5(double a, int b)//顺序不同，可以重载

{

cout << "func5(double a,int b)的调用" << endl;

}

void func5(int a , double b)//顺序不同，可以重载

{

cout << "func5(int a,double b)的调用" << endl;

}

//1.引用作为重载条件

void func6(int& a)//int &a=10;不合法，引用需要先初始化比如int b=10；int &a=b;

{

cout << "func6(int &a)调用" << endl;

}

void func6(const int& a)//const int 引用和int引用属于类型不同

//const int &a=10;合法；相当于先创建int temp=10；再使得const int &a=temp；

{

cout << "func6(const int &a)调用" << endl;

}

//2.函数重载碰到默认函数

void func7(int a ,int b = 10)

{

cout << "func7(int a，int b)的调用" << endl;

}

void func7(int a)

{

cout << "func7(int a)的调用" << endl;

}

int main()

{

cout << func1(10,30) << endl;//这样会把b=20改为b=30，而不是把c赋值

cout << func3() << endl;

func4(10,30);//含有占位参数；必须输入两个参数才能运行

func5();//函数重载调用

func5(3.14);//函数重载调用

func5(2, 9.8);//函数重载调用

func5(9.8, 2);//函数重载调用

//1.引用作为重载条件

int a = 10;

func6(a);//调用的是不加const的函数

func6(10);//调用的是带const的函数

//func7(10);不能这么调用了，编译器不知道该调用哪个;有歧义

func7(10, 20);

system("pause");

return 0;

}

20.类和对象\_封装1

#include <iostream>

using namespace std;

//封装

//访问权限有三种：

// 1.public公共权限(成员类内、类外均可访问)

// 2.protected保护权限（成员类内可以访问、类外不可以访问）儿子可以访问父亲保护的内容

// 3.private私有权限（同保护权限）儿子不可以访问父亲的私有内容

//

class Circle//class代表设计一个类，类后面紧跟着的就是类的名称

{

//公共权限

public:

//属性

int m\_r;

double calculateZC()

{

return 2 \* 3.14159 \* m\_r;

}

};

//设计学生类

class Student

{

public://公共权限

//类中的属性和行为统称为成员

//属性：成员属性、成员变量

string m\_name;

int m\_id;//学号

//行为：成员函数、成员方法

void showstudent()//展示学生姓名和id

{

cout << "姓名： " << m\_name << " 学号： " << m\_id << endl;

}

//给姓名赋值

void setname(string name)

{

m\_name = name;

}

//给学号赋值

void setid(int id)

{

m\_id = id;

}

};

//三种权限

class person

{

public:

//公共权限

string name;

protected:

//保护权限

string car;

private:

//私有权限

int password;

public:

void func()

{

name = "zhangsan";

car = "拖拉机";

password = 123456;

}

};

//struct 和 class 区别

//唯一区别在于默认访问权限不同

//struct默认权限为公共

//class默认权限为私有

class C1

{

int m\_A;//默认私有权限

};

struct C2

{

int m\_A;//默认权限是公共

};

int main()

{

//实例化（通过一个类创建一个对象的过程）

Circle c1;

//给圆的对象的属性赋值

c1.m\_r = 10;

cout << "圆的周长为：" << c1.calculateZC() << endl;

//创建一个具体的学生 实例化对象

Student s1;

s1.m\_name = "zhangsan";

s1.m\_id = 1;

//显示学生信息

s1.showstudent();

//创建第二个学生

Student s2;

s2.setname("lisi");//给姓名赋值的另一种方法

s2.setid(2);//给id赋值的另一种方法

s2.showstudent();

//实例化具体对象，分别尝试访问三个权限

person p1;

p1.name = "李四";

//只能修改name公共权限，不能修改protected、private权限：car和password

p1.func();//func在public权限下故可以类外访问

C1 c1;

//c1.m\_A = 100;class默认成员私有，无法访问

C2 c2;

c2.m\_A = 100;//struct默认公共权限，可以访问

system("pause");

return 0;

}

21.类和对象\_封装2

#include <iostream>

using namespace std;

//成员属性设置为私有：①可以自己控制读写权限②可以自己检测数据的有效性

class person

{

public:

//设置姓名:使得姓名可读可写

void setname(string name)

{

m\_name = name;

}

//获取姓名

string getname()

{

return m\_name;

}

//获取年龄:只读不写

int getage()

{

//初始化年龄为0岁

return m\_age;

}

//使得年龄可以设置

void setage(int age)

{

if (age < 0 || age>150)

{

cout << "你这个老妖精" << endl;

return;//设置错误直接返回

}

m\_age = age;

}

void setlover(string lover)

{

m\_lover = lover;

}

private:

//姓名 可读可写

string m\_name;

//年龄 只读

int m\_age;

//情人 只写

string m\_lover;

};

int main()

{

person p;

//读写姓名

p.setname("张三");

p.setage(1);

cout << p.getname() <<"的年龄为"<<p.getage() << endl;

p.setlover("李四");//情人是只写不读的不能访问

system("pause");

return 0;

}

22.类和对象\_封装例1

#include <iostream>

using namespace std;

//设计立方体类Cube

//求出立方体的面积和体积

//分别用全局函数和成员函数判断两个立方体是否相等

class cube

{

public:

//设置获取，长宽高

//长

void setL(int L)

{

m\_L = L;

}

int getL()

{

return m\_L;

}

//宽

void setW(int W)

{

m\_W = W;

}

int getW()

{

return m\_W;

}

//高

void setH(int H)

{

m\_H = H;

}

int getH()

{

return m\_H;

}

//获取表面积

int calculateA()

{

return 2 \* m\_L \* m\_H + 2 \* m\_L \* m\_W + 2 \* m\_H \* m\_W;

}

//获取体积

int calculateV()

{

return m\_L \* m\_H \* m\_W;

}

//利用成员函数判断两个立方体是否相等

bool issamebyclass(cube &c)

{

if (m\_L == c.getL() && m\_W == c.getW() && m\_H == c.getH())

{

return true;

}

return false;

}

private://私有属性

int m\_L;

int m\_H;

int m\_W;

};

//利用全局函数判断两个立方体是否相等

bool issame(cube &c1, cube &c2)

{

if (c1.getL() == c2.getL() && c1.getW() == c2.getW() && c1.getH() == c2.getH())

{

return true;

}

return false;

}

int main()

{

//设置一个立方体

cube c1;

c1.setL(10);

c1.setW(90);

c1.setH(10);

cout << "体积" << c1.calculateA() << endl;

cout << "表面积" << c1.calculateV() << endl;

//设置第二个立方体

cube c2;

c2.setH(10);

c2.setL(10);

c2.setW(10);

//全局函数判断两个立方体是否相等

bool ret = issame(c1, c2);

if (ret)

{

cout << "c1和c2相等" << endl;

}

else

{

cout << "c1和c2不相等" << endl;

}

//成员函数判断两个立方体是否相等

bool ret2 = c1.issamebyclass(c2);

if (ret2)

{

cout << "c1和c2相等" << endl;

}

else

{

cout << "c1和c2不相等" << endl;

}

system("pause");

return 0;

}

24.类和对象\_对象特性1

#include <iostream>

using namespace std;

//初始化对象的清理

//C++中面向对象，每个对象都会有初始化设置以及对象销毁前的清理数据的设置

//对象的初始化和清理是两个非常重要的安全问题

//C++利用构造函数和析构函数解决上述问题，两个函数会被编译器自动调用

//如果自己不提供 构造函数 析构函数，编译器会提供空实现

//构造函数 主要用于在创建对象时为对象的成员属性赋值，构造函数由编译器自动调用无需手动

//程序调用对象时会自动调用 构造函数，无需手动调用且只会调用一次

//析构函数~：主要作用在对象销毁前系统自动调用，执行一些清理工作

// 程序在销毁前会自动调用析构，无需手动调用而且只会调用一次

//构造函数 与 析构函数都没有返回值且不写void

class person

{

public:

//1.构造函数 初始化

//函数名与类的名称相同

//构造函数可以有参数，可发生重载

person()

{

cout << "person构造函数的调用" << endl;//如果不写这个构造函数编译器会自己写一个：大括号内什么都没有

}

//2.析构函数 进行清理操作

//函数名和类名相同加在名称前

//析构函数不可以由参数，不可以发生重载

//对象在销毁前会调用析构函数且只调用一次

~person()

{

cout << "person的析构函数调用" << endl;

}

};

//构造和析构 必须有的，如果自己不写，编译器提供一个空的

void test1()

{

person p;//在栈上的数据，test1执行完毕后自动释放此对象

}

int main()

{

//创建函数的时候构造函数会被自动调用且只调用一次

test1();

//析构函数调用且只调用一次

person p;//这样只有构造没有析构函数了 按任意键后才会出现析构函数

return 0;

}

25.构造函数的分类及调用

#include <iostream>

using namespace std;

//构造函数的分类及调用

//1.分类：参数分：有参构造 无参构造

// 类型分：普通构造 拷贝构造

class person

{

public:

person()//默认无参构造

{

cout << "person的构造函数调用" << endl;

}

person(int a)//有参构造

{

age = a;

cout << "person的构造函数调用" << endl;

}

//拷贝构造函数(拷贝一个人)

person(const person &p)//拷贝在函数前要加上const 函数后要加&引用

{

//将 传入 的人身上的所有属性拷贝到 此人 身上

age = p.age;

cout << "person的拷贝构造函数调用" << endl;

}

~person()

{

cout << "person的析构函数调用" << endl;

}

int age;//一定要给age 定义类型不然报错

};

//构造函数的调用

void test1()

{

//1.括号法

person p1;//默认构造函数调用,不要加括号；否则编译器会认为是函数的声明

person p2(10);//有参构造函数调用

person p3(p2);//拷贝构造函数

cout << "p2的年龄为" << p2.age << endl;

cout << "p3的年龄为" << p3.age << endl;

//2.显示法

person p4;//无参

person p5 = person(10);//有参；person（10）是匿名构造函数，当前执行完后就被释放

person p6 = person(p5);//拷贝

//3.隐式转换法

person p7 = 10;//相当于调用有参构造，写了person p4=person(10)

person p8 = p4;//拷贝构造

}

int main()

{

test1();

system("pause");

return 0;

}

26.拷贝构造函数调用时机

//拷贝构造函数调用时机的三种情况

//①使用一个已经创建完毕的对象来初始化一个新对象

//②值传递方式给函数参数传值

//③以值方式返回局部对象

//构造函数的调用规则

//\*\*\*默认情况下，C++编译器至少给一个类添加三个函数：1构造函数2析构函数3拷贝构造函数

#include <iostream>

using namespace std;

class Person

{

public:

//Person()

//{

// cout << "Person的默认构造函数调用" << endl;

//}

~Person()

{

cout << "Person的析构函数调用" << endl;

}

Person(int age)

{

cout << "Person的有参构造函数调用" << endl;

m\_Age = age;

}

Person(const Person& p)//不写编译器也会写

{

m\_Age = p.m\_Age;

cout << "Person的拷贝构造函数调用" << endl;

}

int m\_Age;

};

//如果我们写了有参构造函数，编译器就不再提供默认构造函数，依然提供拷贝函数

//如果写了拷贝构造函数，编译器就不再提供其它普通构造函数了

//void test01()

//{

// Person p;

// p.m\_Age = 18;

// Person p2(p);

// cout << "p2的年龄为：" << p2.m\_Age <<endl;

//}

void test02()

{

Person p(28);

Person p2(p);

}

int main()

{

//test01();

test02();

system("pause");

return 0;

}

27.深拷贝与浅拷贝

#include <iostream>

using namespace std;

//浅拷贝：简单赋值操作（编译器提供的拷贝构造函数）

//深拷贝：在堆区重新申请空间，进行拷贝操作

class Person

{

public:

Person()

{

cout << "Person的默认构造函数调用" << endl;

}

Person(int age,int height)

{

m\_Age = age;

new int(height);

m\_Height = new int(height);//使用指针接受堆区数据

cout << "Person的有参构造函数调用" << endl;

}

//自己创建拷贝构造函数，避免浅拷贝带来的问题

Person(const Person& p)

{

cout << "Person拷贝函数调用" << endl;

m\_Age = p.m\_Age;

//编译器默认实现m\_Height = p.m\_Height;

//\*深拷贝操作如下

m\_Height = new int(\*p.m\_Height);

}

~Person()

{

////析构代码，将堆区开辟数据做释放操作

if (m\_Height != NULL)

{

delete m\_Height;

m\_Height = NULL;

}

//浅拷贝时以上操作崩溃

cout << "Person的析构函数调用" << endl;

}

int m\_Age;

int \*m\_Height;//指针

};

void test01()

{

Person p1(18,169);

cout << "p1的年龄" << p1.m\_Age<<"身高为"<< \*p1.m\_Height<< endl;

//p1.m\_Height身高会输出指针，故要加\*解引用

Person p2(p1);

cout << "p2的年龄为" << p2.m\_Age << "身高为" << \*p2.m\_Height << endl;

//运行结果p2的年龄也为18

//编译器提供了拷贝构造函数（浅拷贝）

//指针指定堆区的一块内存，浅拷贝会导致堆区内存重复释放，需要用深拷贝

//深拷贝重新指定一块内存，两块内存中存放着相同的数据

}

int main()

{

test01();

system("pause");

return 0;

}

28.初始化列表

#include <iostream>

using namespace std;

//C++提供了初始化列表语法，用来初始化属性

class Person

{

public:

////传统初始化操作

//Person(int a, int b, int c)

//{

// m\_A = a;

// m\_B = b;

// m\_C = c;

//}

////初始化列表初始化属性

//Person() :m\_A(10), m\_B(20), m\_C(30)

//{

//}

//更灵活得初始化列表初始化属性，注意冒号位置

Person(int a,int b,int c) :m\_A(a), m\_B(b), m\_C(c)

{

}

int m\_A;

int m\_B;

int m\_C;

};

void test01()

{

//Person p(10, 20, 30);//传统初始化操作

//Person p;//初始化列表初始化属性

//cout << "m\_A= " << p.m\_A << endl;

//cout << "m\_B= " << p.m\_B << endl;

//cout << "m\_C= " << p.m\_C << endl;

Person p(30,20,10);//灵活的初始化列表初始化属性

cout << "m\_A= " << p.m\_A << endl;

cout << "m\_B= " << p.m\_B << endl;

cout << "m\_C= " << p.m\_C << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

29.类对象作为类成员

#include <iostream>

using namespace std;

//C++中的成员可以是另一类的对象，我们称该成员为对象成员

class Phone//设计一个手机类

{

public:

Phone(string pName)

{

m\_PName = pName;

cout << "Phone" << endl;

}

string m\_PName;//手机品牌名称

};

class Person//人类

{

public:

Person(string name, string pName) : m\_Name(name),m\_Phone(pName)

{

cout << "Person" << endl;

}

string m\_Name;

Phone m\_Phone;

};

void test01()//构造时会先构造Phone再构造Person；先释放Person再释放Phone

//析构（释放）顺序与构造顺序相反

{

Person p("张三", "iPhone Pro");

cout << p.m\_Name << "拿着" << p.m\_Phone.m\_PName << endl;

//注意p.m\_Phone.m\_PName

}

int main()

{

test01();

system("pause");

return 0;

}

30.静态成员

#include <iostream>

using namespace std;

//静态成员函数

//1.所有对象共享同一个函数

//2.静态成员函数只能访问静态成员变量

class Person

{

public:

//静态成员函数

static void func()

{

m\_A = 100;//静态成员函数可以访问静态成员变量（在全局区）

//m\_B = 200;//静态成员函数 不可以访问 非静态成员变量（在堆区）

cout << "static void func 调用" << endl;

}

static int m\_A;//静态成员变量

int m\_B;//非静态成员变量

//静态成员函数也有访问权限，私有权限在类外访问不到

private:

static void func2()

{

cout << "static void func2调用" << endl;

}

};

int Person::m\_A = 0;

//两种访问方式

void test01()

{

//1.通过对象访问

Person p;

p.func();

//2.通过类名访问

Person::func();

//Person::func2();//类外访问不到私有的成员函数

}

int main()

{

test01();

system("pause");

return 0;

}

31.C++对象模型和this指针

#include <iostream>

using namespace std;

//成员变量 和 成员函数 分开存储

class Person

{

int m\_A;//非静态成员变量，属于类的对象上的数据

static int m\_B;//静态成员变量，不属于类的对象上的数据

//没写static都是非静态的

void func(){}//非静态成员函数，不属于类的对象上

static void func2() {}//静态成员函数，也不属于类的对象上

};

void test01()

{

Person p;

cout << "size of p=" << sizeof(p) << endl;//查看p占多少内存空间

//空对象占用内存1个字节：C++编译器会给每个空对象分配一个字节为了区分空对象在内存上的位置

//每个空对象也应该有一个独一无二的内存地址

}

void test02()

{

Person p;

cout << "size of p=" << sizeof(p) << endl;

//对象内有int，分配4个字节的内存

//加上static int m\_B;仍然占4个字节内存

//加上void func(){}，仍占四个字节，表明非静态成员函数和非静态成员变量分开存储

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

32.this指针

#include <iostream>

using namespace std;

//this指针

class Person

{

public:

Person(int age)

{

this->age = age;//加this，否则下方int age 会乱码

}

Person& PersonAddAge(Person& p)//要返回本体使用Person&代替void作为返回

{

this->age += p.age;

//this是指向p2的指针，而\*this是指向p2这个对象本体

return \*this;//返回对象本身用this

}

int age;//直接这样会输出乱码，故要在上方加上this

};

//1解决名称冲突

void test01()

{

Person p1(18);

cout << "p1的年龄为： " << p1.age << endl;

Person p2(10);

//2返回对象本身用this

p2.PersonAddAge(p1);

cout << "p2的年龄为： " << p2.age << endl;

p2.PersonAddAge(p1).PersonAddAge(p1).PersonAddAge(p1);

//要一直追加年龄需要return\*this

cout << "p2的年龄为： " << p2.age << endl;

}

//2返回对象本身用this

int main()

{

test01();

system("pause");

return 0;

}

33.空指针访问成员函数

#include <iostream>

using namespace std;

//空指针调用成员函数

class Person

{

public:

void showClassName()

{

cout << "this is Person class" << endl;

}

void showPersonAge()

{

//报错的原因是传入的指针为NULL

if (this == NULL)

{

return;

}

cout << "age=" << this->m\_Age << endl;

}

int m\_Age;

};

void test01()

{

Person\* p = NULL;//空指针

p->showClassName();

p->showPersonAge();

}

int main()

{

test01();

system("pause");

return 0;

}

34.const修饰成员函数

#include <iostream>

using namespace std;

//常函数：成员函数后加const

class Person

{

public:

//this指针的本质是指针常量，指针的\*指向是不可以修改的

void showPerson() const//const使得m\_A的 值无法修改

//const（不可修改指针指向值） Person\*const(不可修改指针指向) this

//\*成员函数后加const使得指针指向值也不能修改

{

/\*this->m\_A = 100;\*/

//this = NULL;//THIS指针不可以修改指针的指向

}

void func()//不是常函数

{

}

int m\_A;

mutable int m\_B;//mutable特殊变量，常函数中也可以修改这个值

};

void test01()

{

Person p;

p.showPerson();

}

//常对象：声明对象前加const

void test02()

{

const Person p;//对象前加const，变为常对象

//p.m\_A = 100;//不能改

p.m\_B = 100;//有mutable的特殊变量可以改

//常对象只能调用常函数

p.showPerson();

//p.func();//常对象 不能调用非常函数，因为普通成员函数可以修改属性

}

int main()

{

system("pause");

return 0;

}

36.全局函数做友元

#include <iostream>

using namespace std;

//友元friend

//友元的三种实现

//1全局函数做友元

//2类做友元

//3成员函数做友元

//建筑物的类

class Building

{

//\*\*\*使得goodgay函数作为友元可以访问building中的私有成员

friend void goodgay(Building\* building);

public:

Building()

{

m\_sittingroom = "客厅";

m\_bedroom = "卧室";

}

public:

string m\_sittingroom;//客厅

private:

string m\_bedroom;//卧室

};

//全局函数 访问私有成员

void goodgay(Building \*building)

{

cout << "好基友的全局函数访问" << building->m\_sittingroom << endl;

//私有成员的访问；需要将全局函数前加friend放入类class的最上方

cout << "好基友的全局函数访问" << building->m\_bedroom << endl;

}

void test01()

{

Building building;

goodgay(&building);

}

int main()

{

test01();

system("pause");

return 0;

}

37.类做友元

#include <iostream>

using namespace std;

//类做友元

class Building;//告诉编译器一会我会写这个类，先不要报错

class goodgay

{

public:

goodgay();

void visit();//参观函数访问building中的属性

private:

Building \* building;

};

class Building

{

//\*\*goodgay这类是本类的好朋友可以访问私有成员,有了下面这行才可以访问bedroom

friend class goodgay;

public:

Building();

public:

string m\_sittingroom;

private:

string m\_bedroom;

};

//类外写成员函数

Building::Building()

{

m\_sittingroom = "客厅";

m\_bedroom = "卧室";

}

goodgay::goodgay()

{

//创建建筑物的对象

building = new Building;

}

//参观函数的类外是此案

void goodgay::visit()

{

cout << "好基友类正在访问" << building->m\_sittingroom << endl;

cout << "好基友类正在访问" << building->m\_bedroom << endl;

}

void test01()

{

goodgay gg;

gg.visit();

}

int main()

{

test01();

system("pause");

return 0;

}

38.成员函数做友元

#include <iostream>

using namespace std;

//成员函数做友元

class Building;//先声明避免编译器报错

class goodgay

{

public:

goodgay();

void visit();//visit可以访问building中私有成员

void visit2();//visit2不可以访问building中私有成员

private:

Building \* building;

};

class Building

{

//\*\*告诉编译器，goodgay类下的visit成员函数作为本类的好朋友，可以访问私有成员

friend void goodgay::visit();

public:

Building();

public:

string m\_sittingroom;

private:

string m\_bedroom;

};

//类外实现成员函数

Building::Building()

{

m\_sittingroom = "客厅";

m\_bedroom = "卧室";

}

goodgay::goodgay()

{

building = new Building;//为building创建堆区且用指针维护该对象

}

void goodgay::visit()//visit可以访问building中私有成员

{

cout << "visit在访问" << building->m\_sittingroom << endl;

cout << "visit在访问" << building->m\_bedroom << endl;

}

void goodgay::visit2()//visit2不可以访问building中私有成员

{

cout << "visit2在访问" << building->m\_sittingroom << endl;

}

void test01()

{

goodgay gg;

gg.visit();

gg.visit2();

}

int main()

{

test01();

system("pause");

return 0;

}

39.加号运算符重载

#include <iostream>

using namespace std;

//运算符重载：对已有的运算符进行重新定义，以适应不同的数据类型

//加号运算符重载：实现两个自定义数据类型相加的运算

//通过自己写成员函数，实现两个对象相加属性后得到虚拟的对象

//比如创建两个人，把他们的属性相加

class Person

{

public:

////1、成员函数重载+

//Person operator+(Person& p)

//{

// Person temp;

// temp.m\_A = this->m\_A + p.m\_A;

// temp.m\_B = this->m\_B + p.m\_B;

// return temp;

//}

int m\_A;

int m\_B;

};

//2、全局函数重载+

Person operator+(Person& p1, Person& p2)

{

Person temp;

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

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

return temp;

}

void test01()

{

Person p1;

p1.m\_A = 10;

p1.m\_B = 10;

Person p2;

p2.m\_A = 10;

p2.m\_B = 10;

//成员函数的本质调用

//Person p3 = p1.operator+(p2);

//全局函数重载的本质调用

//Person p3 = operator+(p1, p2);

Person p3 = p1 + p2;

cout << "p3A" << p3.m\_A << endl;

cout << "p3B" << p3.m\_B << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

40.左移运算符重载

#include <iostream>

using namespace std;

//左移运算符重载：输出自定义的数据类型

//输出对象，而不输出其属性

//重载的左移运算符可以实现输出自定义的数据类型

class Person

{

//友元 全局函数 访问私有内容

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

public:

Person(int a, int b)

{

m\_A = a;

m\_B = b;

}

private:

//利用成员函数 左移运算符

//不会利用成员函数重载<<,因为这样p在cout左侧

int m\_A;

int m\_B;

};

//只能利用全局函数重载左移运算符,ostream输出流是cout的类型

ostream & operator<<(ostream &cout,Person &p)//本质cout<<p;ostream是cout的类型

{

cout << "m\_A=" << p.m\_A << "m\_B=" << p.m\_B ;

return cout;

}

void test01()

{

Person p(10,10);

//p.m\_A = 10;

//p.m\_B = 10;

cout << p << endl;//试图输出p（自定义的数据类型）

}

int main()

{

test01();

system("pause");

return 0;

}

41.递增运算符重载

#include <iostream>

using namespace std;

//重载递增++运算符

//自定义的整型

class myinteger

{

friend ostream& operator << (ostream& cout, myinteger myint);

public:

myinteger()

{

m\_Num = 0;

}

//重载++运算符

//前置++:返回引用

myinteger& operator++()

{

//先进行++运算

m\_Num++;

//再将自身做返回

return \*this ;//返回引用，不返回值；为了一直对一个数据操作

}

//重载后置++运算符:在括号内加int（必须是int 不能是double float） int代表占位参数，用于区分前置和后置

myinteger& operator++(int)

{

//先 记录当时结果

myinteger temp = \*this;

//后 递增

m\_Num++;

//最后将记录结果返回

return temp;//后置递增返回值

}

private:

int m\_Num;

};

//重载左移<<运算符

ostream& operator<<(ostream & cout, myinteger myint)

{

cout << myint.m\_Num;

return cout;

}

void test01()

{

myinteger myint;

cout << ++myint << endl;

cout << myint << endl;

}

void test02()

{

myinteger myint;

cout << myint++ << endl;

cout << myint << endl;

}

int main()

{

test01();

test02();

int c = 10;

cout << c++ << endl;//c++输出的是起始的c，但是c+1；10

cout << c << endl;//11

int d = 10;

cout << ++d << endl;//使得d+1且输出d+1

cout << d << endl;//11

system("pause");

return 0;

}

45.赋值运算符重载

#include <iostream>

using namespace std;

//赋值运算符重载

//编译器给一个类添加四个函数：构造、析构、拷贝函数和④赋值运算符operator=，对属性值拷贝

class Person

{

public:

Person(int age)

{

m\_Age = new int(age);

}

~Person()//析构

{

if (m\_Age != NULL)

{

delete m\_Age;

m\_Age = NULL;

}

}

//\*\*重载 赋值运算符

Person& operator=(Person& p)

{

//编译器提供浅拷贝如下，使得p2的\*m\_Age也指向p2的堆区内存；使得堆区内存重复释放，崩溃

//m\_Age=p.m\_Age;

//应该先判断是否有属性在堆区，如果有先释放干净再做深拷贝

if (m\_Age != NULL)

{

delete m\_Age;

m\_Age = NULL;

}

//深拷贝

m\_Age = new int(\*p.m\_Age);

//返回对象本身(引用)，不是值

return \*this;

}

int \*m\_Age;//堆区数据

};

void test01()

{

Person p1(18);

Person p2(20);

Person p3(30);

//赋值操作

p3 = p2 = p1;

//深拷贝赋值

cout << "p1年龄" << \*p1.m\_Age << endl;

cout << "p2年龄" << \*p2.m\_Age << endl;

cout << "p3年龄" << \*p3.m\_Age << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

46.关系运算符重载

#include <iostream>

using namespace std;

//重载关系运算符><==！=；对比自定义的数据

class Person

{

public:

Person(string name, int age)

{

m\_Name = name;

m\_Age = age;

}

//重载关系运算符==

bool operator==(Person &p)//bool返回对错

{

if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)

{

return true;

}

return false;

}

//重载！=

bool operator!=(Person& p)//bool返回对错

{

if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)

{

return false;

}

return true;

}

string m\_Name;

int m\_Age;

};

void test01()

{

Person p1("Tom", 18);

Person p2("Tom", 10);

if (p1 == p2)

{

cout << "p1和p2相等" << endl;

}

else

{

cout << "p1和p2不相等" << endl;

}

if (p1 != p2)

{

cout << "p1和p2不相等" << endl;

}

else

{

cout << "p1和p2相等" << endl;

}

}

int main()

{

test01();

system("pause");

return 0;

}

47.函数调用运算符重载

#include <iostream>

using namespace std;

//函数调用（）运算符重载 仿函数 没有固定写法，非常灵活

//打印输出类

class myprint

{

public:

//\*重载的 函数调用运算符 反函数

void operator()(string test)

{

cout << test << endl;

}

};

class myadd

{

public:

int operator()(int num1, int num2)

{

return num1 + num2;

}

};

//真正的函数

void myprint2(string test)

{

cout << test << endl;

}

void test01()

{

myprint myprint;

myprint("hello world");//非常像真的函数故叫仿函数

myprint2("hello world");

myadd add;//先写函数名再写函数

int ret = add(100, 100);

cout << ret << endl;

//匿名函数对象

cout << myadd()(100, 100) << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

48.继承-基本语法

#include <iostream>

using namespace std;

//继承是面向对象的三大特性之一（封装、继承、多态）

//共有性质，也有特性;用于减少重复代码

//语法class 子类（派生类）：继承类 父类（基类）

//普通实现页面

//Java页面

//class Java

//{

//public:

// void header()

// {

// cout << "首页、公开课、登录、注册(公共头部)" << endl;

// }

// void footer()

// {

// cout << "帮助中心、交流合作、站内地图（公共底部）" << endl;

// }

// void left()

// {

// cout << "JAVA\PYTHON\C++(公共列表）" << endl;

// }

// void content()

// {

// cout << "JAVA视频" << endl;

// }

//};

////python页面

//class python

//{

//public:

// void header()

// {

// cout << "首页、公开课、登录、注册(公共头部)" << endl;

// }

// void footer()

// {

// cout << "帮助中心、交流合作、站内地图（公共底部）" << endl;

// }

// void left()

// {

// cout << "JAVA\PYTHON\C++(公共列表）" << endl;

// }

// void content()

// {

// cout << "python视频" << endl;

// }

//};

//继承实现页面

//公共页面

class Basepage

{

public:

void header()

{

cout << "首页、公开课、登录、注册(公共头部)" << endl;

}

void footer()

{

cout << "帮助中心、交流合作、站内地图（公共底部）" << endl;

}

void left()

{

cout << "JAVA\PYTHON\C++(公共列表）" << endl;

}

};

//JAVA页面

class Java:public Basepage

{

public:

void content()

{

cout << "JAVA学科视频" << endl;

}

};

class python :public Basepage

{

public:

void content()

{

cout << "python学科视频" << endl;

}

};

void test01()

{

cout << "JAVA下载视频页面如下： " << endl;

Java ja;

ja.header();

ja.footer();

ja.left();

ja.content();

cout << "----------------------------" << endl;

cout << "python下载视频页面如下： " << endl;

python py;

py.header();

py.footer();

py.left();

py.content();

}

int main()

{

test01();

system("pause");

return 0;

}

49.继承方式

#include <iostream>

using namespace std;

//3种继承方式：1公共继承public2保护继承protected3私有继承private

//私有权限永远是私有权、子类无法访问

//保护继承使得公共权限变为保护权限

//私有继承使得公共和保护权限均变为私有权限

class Base1

{

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class son1 :public Base1

{

public:

void func()

{

m\_A = 10;//仍是公共权限

m\_B = 10;//仍是保护权限

//m\_C=10;//父类中私有权限不能访问

}

};

class son2 :protected Base1

{

public:

void func()

{

m\_A = 10;//变为保护权限

m\_B = 10;//仍是保护权限

//m\_C=10;//父类中私有权限不能访问

}

};

class son3 :private Base1

{

public:

void func()

{

m\_A = 10;//变为私有权限

m\_B = 10;//变为私有权限

//m\_C=10;//父类中私有权限不能访问

}

};

class grandson :public son3

{

void func()

{

//m\_A = 1000;//是父类son3的私有成员

}

};

void test01()

{

son1 s1;

s1.m\_A = 100;

//s1.m\_B = 100;//保护权限类外访问不到

son2 s2;

//s2.m\_A = 100;//变为保护权限，类外访问不到

}

int main()

{

test01();

system("pause");

return 0;

}

50.继承中的对象类型

#include <iostream>

using namespace std;

//从父类继承过来的成员，哪些属于子类对象中

class Base

{

public:

int m\_A;

protected:

int m\_B;

private:

int m\_C;

};

class Son :public Base//共4个int 16个字节

{

public:

int m\_D;

};

//利用Developer command prompt for vs2022查看

//cl /d1 reportSingleClassLayoutSon "继承中的对象类型.cpp"

//cL d①

void test01()

{

//父类中所有非静态的属性都会被子类继承下去

//编译器把private隐藏了，但是继承下去了

cout << "size of Son=" << sizeof(Son) << endl;//占16个字节

}

int main()

{

test01();

system("pause");

return 0;

}

51.继承中的构造和析构顺序

#include <iostream>

using namespace std;

//继承中的构造和析构顺序

class Base

{

public:

Base()

{

cout << "Base的构造函数" << endl;

}

~Base()

{

cout << "Base的析构函数" << endl;

}

};

class Son :public Base

{

public:

Son()

{

cout << "son的构造函数" << endl;

}

~Son()

{

cout << "son的析构函数" << endl;

}

};

void test01()

{

Son s;

}

//继承中的构造和析构顺序如下

//先构造父类再构造儿子，先析构儿子再析构父亲

int main()

{

test01();

system("pause");

return 0;

}

52.继承\_同名成员处理

#include <iostream>

using namespace std;

//访问子类 同名成员，直接访问即可

//访问父类 同名成员需要加作用域

class Base

{

public:

Base()

{

m\_A = 100;

}

void func(int a)

{

cout << "Base下func" << endl;

}

int m\_A;

};

class Son :public Base

{

public:

Son()

{

m\_A = 200;

}

void func()

{

cout << "Son下func" << endl;

}

int m\_A;

};

void test01()

{

Son s;

cout << "直接访问m\_A=" << s.m\_A << endl;

cout << "父类中的m\_A=" << s.Base::m\_A << endl;//需要加作用域

s.func();//直接调用调用的是子类的成员

//加作用域调用父类的同名函数

s.Base::func(100);

//如果子类有和父类的 同名成员函数 ，会自动隐藏掉父类所有同名成员函数

//要访问父类的同名成员函数要加参数

}

int main()

{

test01();//直接访问到的 同名成员 是 子 类的

system("pause");

return 0;

}

53.继承\_同名静态static成员

#include <iostream>

using namespace std;

//继承中同名静态成员的处理方式

class Base

{

public:

static int m\_A;

static void func()

{

cout << "Base func" << endl;

}

};

int Base::m\_A = 100;

class Son :public Base

{

public:

static int m\_A;

static void func()//子类出现 静态同名函数 父类中 所有同名成员函数 都被隐藏

{

cout << "Son func" << endl;

}

};

int Son::m\_A = 200;

//同名静态成员属性

void test01()

{

Son s;

//1.通过对象访问

cout << "通过对象访问" << endl;

s.func();

s.Base::func();//加作用域

cout << "Son下m\_A" << s.m\_A << endl;

cout << "Base下m\_A" << s.Base::m\_A << endl;

//2.通过类名访问

cout << "通过类名访问" << endl;

Son::func();

Son::Base::func();

cout << "Son下m\_A" << Son::m\_A << endl;

cout << "Base下m\_A" << Son::Base::m\_A << endl;

//类名的方式 访问 父类作用域下的 m\_A

}

int main()

{

test01();

system("pause");

return 0;

}

54.多继承语法

#include <iostream>

using namespace std;

//多继承的语法

class Base1

{

public:

Base1()

{

m\_A = 100;

}

int m\_A;

};

class Base2

{

public:

Base2()

{

m\_A = 200;

}

int m\_A;

};

//子类继承两个父类

class Son:public Base1, public Base2

{

public:

Son()

{

m\_C = 300;

m\_D = 400;

}

int m\_C;

int m\_D;

};

void test01()

{

Son s;

cout << "sizeof Son=" << sizeof(s) << endl;//son占16个字节

//cl /d1 reportSingleClassLayoutSon "C++允许一类类继承多个类.cpp"

//查看字节数cLdone

//当不同父类出现同名成员，需要加作用域区分

cout << "Base1::m\_A" << s.Base1::m\_A << endl;

cout << "Base2::m\_A" << s.Base2::m\_A << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

55.菱形继承

#include <iostream>

using namespace std;

//菱形继承：两个子类共有一个父类，该两个子类又有一个共同的子类

//少写菱形继承

//动物类

class Animal//虚基类

{

public:

int m\_Age;

};

//利用虚继承，解决菱形继承的问题

//关键字virtual

//羊类

class sheep:virtual public Animal

{

};

//驼类

class tuo:virtual public Animal

{

};

//羊驼类

class yangtuo:public sheep,public tuo

{

};

void test01()

{

yangtuo st;

st.sheep::m\_Age = 18;

st.tuo::m\_Age = 28;

//菱形继承，当两个父类拥有相同数据，需要加以作用域区分

cout << "sheep"<< st.sheep::m\_Age << endl;

cout << "tuo" << st.tuo::m\_Age << endl;

//虚继承继承的是指针，数据age仅一份。

cout << "st.m\_Age" << st.m\_Age << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

56.多态的基本语法

#include <iostream>

using namespace std;

//多态分为：

//1静态多态：函数地址早绑定，编译阶段确定

//2动态多态：函数地址晚绑定，运行阶段确定 virtual;条件①要有继承关系②子类 重写 父类 中的虚virtual函数

//（虚函数）重写：1函数返回值类型 2函数名 3参数列表 完全一致

//重载函数名可以相同 参数不能完全相同

//动态多态使用：父类的指针或者引用指向子类对象

//动物类

class Animal

{

public:

virtual void speak()//地址晚绑定

{

cout << "动物在说话" << endl;

}

};

//猫类

class Cat :public Animal

{

public:

void speak()

{

cout << "猫在说话" << endl;

}

};

class Dog :public Animal

{

public:

void speak()

{

cout << "狗在说话" << endl;

}

};

//执行说话的函数

//函数的地址早绑定：静态多态 动物 说话

//想要猫说话，要地址晚绑定 动态多态 Animal speak加virtual

void dospeak(Animal &animal)//Animal&animal=cat;(允许父子类型的转换)

{

animal.speak();

}

void test01()//会输出 动物 在说话 不是 猫 在说话

{

Cat cat;

dospeak(cat);

Dog dog;

dospeak(dog);

//Animal ani;

//dospeak(ani);

}

void test02()

{

cout << sizeof(Animal) << endl;//指针都占8个字节 64位；4个字节 32位

}

int main()

{

//test01();

test02();

system("pause");

return 0;

}

//子类 重写了 父类的虚函数

//父类的虚函数内容 替换为 子类的虚函数地址

//当 父类指针 指向 子类对象，发生多态

57.多态的原理

#include <iostream>

using namespace std;

//案例：计算器

//多态好处

//1组织结构清晰2可读性强3对于前期和后期扩展以及维护性高

//普通写法

class calculator

{

public:

int getResult(string oper)

{

if (oper == "+")

{

return m\_Num1 + m\_Num2;

}

else if (oper=="-")

{

return m\_Num1 - m\_Num2;

}

else if (oper == "\*")

{

return m\_Num1 \* m\_Num2;

}

else if (oper == "/")

{

return m\_Num1 / m\_Num2;

}

//扩展新功能需要修改原码

}

int m\_Num1;

int m\_Num2;

};

void test01()

{

calculator c;

c.m\_Num1 = 10;

c.m\_Num2 = 10;

cout << c.getResult("+") << endl;

cout << c.getResult("-") << endl;

cout << c.getResult("\*") << endl;\

cout << c.getResult("/") << endl;

}

//多态实现计算器

//实现计算器抽象类

class acalculator

{

public:

virtual int getresult()

{

return 0;

}

int m\_Num1;

int m\_Num2;

};

//加法计算器

class add :public acalculator

{

public:

int getresult()

{

return m\_Num1 + m\_Num2;

}

};

//减法

class minus :public acalculator

{

public:

int getresult()

{

return m\_Num1 - m\_Num2;

}

};

//乘法

class multi :public acalculator

{

public:

int getresult()

{

return m\_Num1 \* m\_Num2;

}

};

//除法

class divide :public acalculator

{

public:

int getresult()

{

return m\_Num1 / m\_Num2;

}

};

void test02()

{

acalculator \* abc = new add;

abc->m\_Num1 = 100;

abc->m\_Num2 = 100;

cout << abc->getresult() << endl;

delete abc;//创建在堆区，用完后销毁

abc = new multi;

abc->m\_Num1 = 100;

abc->m\_Num2 = 100;

cout << abc->getresult() << endl;

delete abc;//创建在堆区，用完后销毁

}

int main()

{

/\*test01();\*/

test02();

system("pause");

return 0;

}

58.纯虚函数和抽象类

#include <iostream>

using namespace std;

//return 0;改为纯虚函数=》抽象类：1无法实例化元对象2子类必须重写，否则也是抽象类

class Base

{

public:

//纯虚函数

virtual void func() = 0;

};

class Son:public Base

{

public:

virtual void func() //virtual可有可无

{

cout << "func调用" << endl;

};

};

void test01()

{

//Base b;//纯虚函数无法实例化对象

//new Base;

Son s;//子类必须重写父类中的纯虚函数，否则无法实例化对象

Base\* base = new Son;

base->func();

}

int main()

{

test01();

system("pause");

return 0;

}

#include <iostream>

using namespace std;

//子类中没有堆区数据可以不写（纯）虚析构；纯虚析构属于抽象类

class Animal

{

public:

Animal()

{

cout << "ani构造函数调用" << endl;

}

////利用虚析构解决 父类指针释放子类时释放不干净

//virtual ~Animal()//

//{

// cout << "ani虚析构函数调用" << endl;

//}

//纯虚析构

//有了纯虚析构后，这个类也属于抽象类，无法实例化对象

virtual ~Animal() = 0;//类内声明，类外实现

virtual void speak() = 0;//纯虚函数

};

Animal::~Animal()//纯虚析构的实现

{

cout << "ani纯虚析构函数调用" << endl;

}

class cat :public Animal

{

public:

cat(string name)

{

cout << "cat构造函数调用" << endl;

m\_Name = new string(name);

}

virtual void speak()

{

cout << \*m\_Name<<"猫说话" << endl;

}

virtual ~cat()//析构函数释放堆区属性，没有释放；需要改为虚析构

//利用虚析构解决 父类指针释放子类时释放不干净

{

if (m\_Name != NULL)

{

cout << "cat析构函数调用" << endl;

delete m\_Name;

m\_Name = NULL;

}

}

string\* m\_Name;//猫名在堆区

};

void test01()

{

Animal\* animal = new cat("Tom");

animal->speak();

//父类指针在析构是，不会调用子类中的析构函数，导致子类如果有堆区属性会内存泄漏

delete animal;

}

int main()

{

test01();

system("pause");

return 0;

}

62.文件操作

#include <iostream>

//1包含头文件

#include <fstream>

using namespace std;

//写文件 包含头文件<fstream>

//1文本文件ASCII码2二进制文件0101

//ofstream:写

//ifstream:读

//fstream:读写

void test01()

{

//2创建流对象

ofstream ofs;

//3指定打开方式

ofs.open("test.txt", ios::out);

//4写内容

ofs << "姓名：张三" << endl;

ofs << "姓名：张三" << endl;

//5关闭文件

ofs.close();

}

int main()

{

test01();

system("pause");

return 0;

}

63.写文件

#include <iostream>

using namespace std;

//1包含头文件

#include<fstream>

#include<string>

//文本文件 读文件 ifstream 或fstream

void test01()

{

//2创建流对象

ifstream ifs;

//3打开文件

ifs.open("D:\\C++\\基本操作学习\\62.文件操作\\test.txt", ios::in);//和python一样都是\\

//且判断是否打开成功

if (!ifs.is\_open())

{

cout << "文件打开失败" << endl;

return;

}

//4读数据

////第一种

//char buf[1024] = { 0 };

//while (ifs >> buf)

//{

// cout << buf << endl;

//}

////第二种

//char buf[1024] = { 0 };

//while (ifs.getline(buf, sizeof(buf)))

//{

// cout << buf << endl;

//}

////第三种

//string buf;

//while ( getline (ifs,buf))//要直接使用getline需要string头文件

//{

// cout << buf << endl;

//}

//第四种:效率低下

char c;

while ((c=ifs.get())!=EOF)//每次读一个字符，读到的放入c，判断是否读到文件尾end of file

{

cout << c;

}

//5.关闭文件

ifs.close();

}

int main()

{

test01();

system("pause");

return 0;

}

64.二进制文件

#include <iostream>

using namespace std;

//1包含头文件

#include<fstream>

//二进制文件的打开方式为 ios::binary

#include<string>

class Person

{

public:

char m\_Name[64];

int m\_Age;

};

void test01()

{

//2创建输出流对象

ofstream ofs("person.txt", ios::out | ios::binary);

//3打开文件

/\*ofs.open("person.txt", ios::out | ios::binary);\*/

Person p = { "张三",18 };

//4写文件:通过write函数以二进制方式写数据

ofs.write((const char\*)&p, sizeof(p));

//5关闭文件

ofs.close();

}

//读取二进制文件 :istream read(char \*buffer,int len)字符指针只想内存中一段存储空间，len是读写的字节数

void test02()

{

//2创建流对象

ifstream ifs;

//3打开文件并判断是否打开成功

ifs.open("person.txt", ios::in | ios::binary);

if (!ifs.is\_open())

{

cout << "文件打开失败" << endl;

return;

}

//4读文件

Person p;

ifs.read((char\*)&p, sizeof(Person));

cout << p.m\_Age << p.m\_Name << endl;

//5关闭文件

ifs.close();

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

66.1函数模板的注意事项

#include <iostream>

using namespace std;

//函数模板的注意事项

template<typename T>//typename 可以替换成class

void myswap(T& a, T& b)

{

T temp = a;

a = b;

b = temp;

}

//1.自动类型推导（必须推导出一直的数据类型T才可以使用）

void test01()

{

int a = 10;

int b = 20;

char c = 'c';

myswap(a, b);

//myswap(b, c);推导不出一致的数据类型

cout << "a=" << a << endl;

}

//2.模板必须要确定出T的数据类型才可以使用

template<class T>//class也可以是typename

void func()

{

cout << "func调用" << endl;

}

void test02()

{

func<int>();//模板需要确定 T 的数据类型

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

66.模板\_泛型编程

#include <iostream>

using namespace std;

//模板不可以直接使用，它只是一个框架

//模板的通用并不是万能的

//函数模板：

//建立一个通用函数，其函数 返回值 和 形参 类型可以不具体指定，用一个虚拟类型来代表

//两个整型交换函数

void swapInt(int& a, int& b)

{

int temp = a;

a = b;

b = temp;

}

//交换两个浮点型函数

void swapdouble(double& a, double& b)

{

double temp = a;

a = b;

b = temp;

}

//将交换int和交换double放入一个模板中，使得各种数据类型都能交换

template<typename T>//声明一个模板，告诉编译器后面代码中紧跟着的T不要报错，T是一个通用数据类型

void myswap(T& a, T& b)

{

T temp = a;

a = b;

b = temp;

}

void test01()

{

int a = 10;

int b = 20;

//swapInt(a, b);

//利用函数模板实现交换:两种方法

//1、自动类型推到导

myswap(a, b);

//2.显示指定类型

myswap<int>(a, b);

cout << "a=" << a << endl;

double c = 1.1;

double d = 2.2;

//swapdouble(c, d);

cout << "c=" << c<<endl;

}

int main()

{

test01();

system("pause");

return 0;

}

66.2通用数组排序模板案例

#include <iostream>

using namespace std;

//实现通用对数组进行排序的函数

//从大到小

//算法 选择排序

//测试 char数组,int数组

//交换函数模板

template <class T>

void myswap(T& a, T& b)

{

T temp = a;

a = b;

b = temp;

}

//排序算法

template<class T>

void mysort(T arr[], int len)

{

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

{

int max = i;

for (int j = i + 1; j < len; j++)

{

if (arr[max] < arr[j])

{

max = j;//更新最大值

}

}

if (max != i)

{

//交换max和i元素

myswap(arr[max], arr[i]);

}

}

}

//打印数组的模板

template <class T>

void printA(T arr[], int len)

{

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

{

cout << arr[i] << " ";

}

cout << endl;

}

void test01()

{

//测试char数组

char charArr[] = "badcfe";

int num = sizeof(charArr) / sizeof(char);//计算字符串长度

mysort(charArr, num);

printA(charArr, num);

}

void test02()

{

int intArr[] = {999,99,0,1};

int num = sizeof(intArr) / sizeof(int);

mysort(intArr, num);

printA(intArr, num);

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

66.3普通函数和函数模板的区别

#include <iostream>

using namespace std;

//①普通函数调用时可以发生 自动类型转换（隐式类型转换）

//②函数模板调用时，如果利用 自动类型推到 ，不会发生隐式类型转换

//③如果利用显示 指定类型的方式 ，可以发生隐式类型转换

//普通函数:两数相加

int myadd01(int a,int b)

{

return a + b;

}

void test01()

{

int a = 10;

int b = 20;

char c = 'c';//a->97,c->99

cout << myadd01(a, c) << endl;//自动将字符型变量转化成了asc码，

}

//函数模板：两数相加

template <class T>

T myadd02(T a, T b)

{

return a + b;

}

void test02()

{

int a = 10;

int b = 20;

char c = 'c';//a->97,c->99

//自动类型推导无法发生隐式类型转换

cout << myadd02(a, b) << endl;

//显式指定类型 可以 发生隐式类型转换，转为int

cout << myadd02<int>(a, c) << endl;

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

67.普通函数和函数模板的调用规则

#include <iostream>

using namespace std;

//1如果 普通函数 函数模板 均可调用，有限调用普通函数

//2通过 空模板参数列表 强制 调用函数模板

//3函数模板 也可以 重载

//4如果函数模板 更好的匹配 ，优先调用 函数模板

//最好不要又有函数模板，又有普通函数

//普通函数 (只留下声明 还是会调用 普通函数)

void myprint(int a, int b)

{

cout << "调用的普通函数" << endl;

}

//函数模板(与普通函数名相同)

template<class T>

void myprint(T a, T b)

{

cout << "调用的模板" << endl;

}

//void myprint(T a, T b,T c)

//{

// cout << "调用的模板" << endl;

//}

void test01()

{

int a = 10;

int b = 20;

myprint(a, b);//会调用普通函数

//通过空模板参数列表，强制调用函数模板

myprint<>(a, b);

//myprint(a, b, 100);//发生重载

//更好地匹配，调用模板

char c1 = 'a';

char c2 = 'b';

myprint(c1, c2);

}

int main()

{

test01();

system("pause");

return 0;

}

68.模板的局限性

#include <iostream>

using namespace std;

#include<string.>

//特定数据类型要具体化实现,比如类，数组

class Person

{

public:

Person(string name, int age)

{

this->m\_name = name;

this->m\_age = age;

}

string m\_name;

int m\_age;

};

//对比两个数据是否相等的函数

template<class T>

bool mycompare(T& a, T& b)//引用传入

{

if (a == b)

{

return true;

}

else

{

return false;

}

}

//利用具体化的Person的版本实现代码，优化调用

template<>bool mycompare(Person& a, Person& b)//引用传入

{

if (a.m\_age == b.m\_age && a.m\_name==b.m\_name)

{

return true;

}

else

{

return false;

}

}

void test01()

{

int a = 10;

int b = 20;

bool ret = mycompare(a, b);

if (ret)

{

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

}

else

{

cout << "a!=b" << endl;

}

}

void test02()

{

Person p1("Tom", 10);

Person p2("Jerry", 12);

bool ret = mycompare(p1, p2);

if (ret)

{

cout << "p1=p2" << endl;

}

else

{

cout << "p1!=p2" << endl;

}

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

69.类模板

#include <iostream>

using namespace std;

#include<string>

//类模板, 又有字符串型又有数值型

template<class Nametype,class Agetype>

class Person

{

public:

Person(Nametype name, Agetype age)

{

this->m\_name = name;

this->m\_age = age;

}

void showperon()

{

cout << this->m\_name << endl;

cout << this->m\_age << endl;

}

Nametype m\_name;

Agetype m\_age;

};

void test01()

{

//在声明模板template后面加类，此类成为类模板

Person<string, int> p1("Tom", 20);//类模板的传入方法

p1.showperon();

}

int main()

{

test01();

system("pause");

return 0;

}

70.类模板与函数模板的区别

#include <iostream>

using namespace std;

//1.类模板无 自动类型推导的使用方式

//2.类模板 在模板的参数列表 可以有默认参数//函数模板无默认参数

//类模板

template<class Nametype,class Agetype=int>//int 是默认参数;函数模板无默认参数

class Person

{

public:

Person(Nametype name, Agetype age)

{

this->m\_age = age;

this->m\_name = name;

}

void showperson()

{

cout << this->m\_name << this->m\_age << endl;

}

Nametype m\_name;

Agetype m\_age;

};

//类模板无自动类型推导方式

void test01()

{

Person<string, int>p("Tom", 100);//不能省略《》内的东西，无 自动类型推导

p.showperson();

Person<string>p1("Jerry", 10);

p1.showperson();

}

int main()

{

test01();

system("pause");

return 0;

}

71.类模板中成员函数的创建时机

#include <iostream>

using namespace std;

//类模板中 成员函数 在调用时才去创建

class Person1

{

public:

void showperson1()

{

cout << "Person1show" << endl;

}

};

class Person2

{

public:

void showperson2()

{

cout << "Person2show" << endl;

}

};

template<class T>

class myclass

{

public:

T obj;

//类模板的成员函数

void func1()

{

obj.showperson1();

}

void func2()//没调用所以没创建，不会报错

{

obj.showperson2();

}

};

void test01()

{

myclass<Person1>m;

m.func1();//可以运行

//m.func2();//不能运行

}

int main()

{

system("pause");

return 0;

}

72.类模板对象做函数参数

#include <iostream>

#include<string>

using namespace std;

//类模板对象做函数参数

template<class T1,class T2>

class Person

{

public:

Person(T1 name, T2 age)

{

this->m\_name = name;

this->m\_age = age;

}

void showperson()

{

cout << "姓名" << this->m\_name << "年龄" << this->m\_age << endl;

}

T1 m\_name;

T2 m\_age;

};

//一共有三种传入方式：

//1指定传入类型（最常用）

void printperson1(Person<string,int>&p)

{

p.showperson();

}

void test01()

{

Person<string, int>p("Tom", 20);

printperson1(p);

}

//2参数模板化

template<class T1,class T2>

void printperson2(Person<T1, T2>& p)//参数模板化

{

p.showperson();

//查看类型

cout << "T1的类型为：" << typeid(T1).name() << endl;//typrid(T).name()查看T的数据类型

cout << "T2的类型为：" << typeid(T2).name() << endl;

}

void test02()

{

Person<string, int>p("Jerry", 20);

printperson2(p);

}

//3整个类模板化

template<class T>

void printperson3(T& p)//直接将Person作为一个类

{

p.showperson();

cout << "T的数据类型" << typeid(T).name() << endl;//T是person class类型

}

void test03()

{

Person<string, int>p("唐僧", 90);

printperson3(p);

}

int main()

{

test01();

test02();

test03();

system("pause");

return 0;

}

73.类模板与继承

#include <iostream>

using namespace std;

//类模板与继承

//当类模板碰到继承时，需要注意以下几点

//1当子类继承的父类是一个类模板时，子类在声明的时候，要制定出父类中T的类型

//2如果不指定，编译器无法给子类分配内存

//3如果想灵活指定出父类中T的类型，子类也需变为类模板

template<class T>

class Base

{

T m;

};

class Son: public Base<int> //1不能public Base而不加base的数据类型

{

};

void test01()

{

Son s1;

}

//3如果想灵活指定出父类中T的类型，子类也需变为类模板

template<class T1,class T2>

class Son2 :public Base<T2>

{

public:

Son2()

{

cout << "T1的类型为：" << typeid(T1).name() << endl;

}

T1 obj;

};

void test02()

{

Son2<int, char>S2;//父类是char，子类是int

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

74.类模板成员函数的类外实现

#include <iostream>

using namespace std;

#include<string>

//类模板成员函数的类外实现

//template+作用域+模板参数列表

template<class T1,class T2>

class Person

{

public:

Person(T1 name, T2 age);

//{

// this->m\_name = name;

// this->m\_age = age;

//}

void showperson();

//{

// cout << "姓名" << this->m\_name << "年龄" << this->m\_age << endl;

//}

T1 m\_name;

T2 m\_age;

};

//构造函数的类外实现

template<class T1,class T2>

Person<T1,T2>::Person(T1 name, T2 age)//要写作用域(后面加模板参数列表)，template

{

this->m\_name = name;

this->m\_age = age;

}

template<class T1, class T2>

void Person<T1, T2>::showperson()//先void再作用域再写模板参数列表

{

cout << "姓名" << this->m\_name << "年龄" << this->m\_age << endl;

}

void test01()

{

Person<string, int>p("Jeryy", 20);

p.showperson();

}

int main()

{

test01();

system("pause");

return 0;

}

75.类模板分文件编写

Main.Cpp

#include <iostream>

using namespace std;

#include<string>

#include"Person.hpp"//手动将.h改为.cpp

//#include"Person.cpp"//类模板文件分写要包含cpp而不是.h

//类模板分文件编写

//问题：

// 类模板中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1 包含.cpp而不是.h

//解决2 常用：将.h .cpp写道一起，后缀名改为.hpp

//template<class T1,class T2>

//class Person

//{

//public:

// Person(T1 name, T2 age);

// void showperson();

//

// T1 m\_name;

// T2 m\_age;

//

//};

////模板函数的类外实现

//template<class T1, class T2>

//Person<T1, T2>::Person(T1 name, T2 age)

//{

// this->m\_name = name;

// this->m\_age = age;

//}

//template<class T1, class T2>

//void Person<T1, T2>::showperson()

//{

// cout << "name" << this->m\_name << "age" << this->m\_age << endl;

//}

void test01()

{

Person<string, int>p("Jerry", 18);

p.showperson();

}

int main()

{

test01();

system("pause");

return 0;

}

Hpp文件

#pragma once

#include<iostream>

using namespace std;

template<class T1, class T2>

class Person

{

public:

Person(T1 name, T2 age);

void showperson();

T1 m\_name;

T2 m\_age;

};

//模板函数的类外实现

template<class T1, class T2>

Person<T1, T2>::Person(T1 name, T2 age)

{

this->m\_name = name;

this->m\_age = age;

}

template<class T1, class T2>

void Person<T1, T2>::showperson()

{

cout << "name" << this->m\_name << "age" << this->m\_age << endl;

}

76.类模板与友元

#include <iostream>

using namespace std;

#include<string>

//类模板配合友元的类内和类外实现

//全局函数类内实现：直接在类内声明友元即可

//全局函数类外实现：需要让编译器提前知道全局函数的存在《-麻烦

//类外实现

template<class T1, class T2>

class Person;//也要提前让编译器知道有Person这个类

template<class T1, class T2>

void printperson2(Person<T1, T2>p)

{

cout << "姓名：" << p.m\_name << "年龄：" << p.m\_age << endl;

}

template<class T1,class T2>

class Person

{

//全局函数 类内实现友元关键字friend

friend void printperson(Person<T1, T2>p)

{

cout << "姓名：" << p.m\_name << "年龄：" << p.m\_age << endl;

}

//全局函数类外实现

//加空模板的参数列表，类外实现要让编译器提前知道这段代码，剪切到最上方

friend void printperson2<>(Person<T1, T2>p);

public:

Person(T1 name, T2 age)

{

this->m\_name = name;

this->m\_age = age;

}

private:

T1 m\_name;

T2 m\_age;

};

//1.全局函数在类内实现

void test01()

{

Person<string, int>p("Tom", 20);

printperson(p);

}

void test02()

{

Person<string, int>p("Tom", 20);

printperson2(p);

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

78. STL

#include <iostream>

using namespace std;

//可重复利用

//C++的面向对象和泛型编程思想，目的是复用性的提升

// 大多数情况下，数据结合 和 算法 都未能有一套标准，导致被迫从事大量重复工作

//为建立数据结构和算法的一套标准，就诞生了STL

//stndard template library--STL

//STL广义上分为容器（container）算法(algorithm)迭代器（iterator）

//容器 算法 通过 迭代器 进行无缝链接

//STL几乎所有代码都采用了模板类或模板函数

//STL的六大组件

//1容器： 各种 数据结构vector、list、deque、set、map等，用来存放数据

//2算法：sort find copy for\_each（遍历）

//3迭代器：扮演了 容器 和 算法 之间的胶合剂

//4仿函数：行为类似函数，可作为 算法 的某种策略

//5适配器：一种用来修饰 容器 或 仿函数 或 迭代器接口 的东西

//6空间适配器：负责 空间的配置与管理

//STL中容器、算法、迭代器

//STL就是将运用最广泛的一些数据结构实现出来

//数组、链表、栈、树、队列、集合、映射表

//序列式容器 关联式容器

//序列式容器：强调值的排序，每个 元素 有 固定 的位置

//关联式容器：二叉树结构，各元素之间没有严格的物理上的顺序关系

//算法：解决逻辑或数学上的问题algorithm

//质变算法:运算期间会 更改 元素内容：拷贝、替换、删除

//非质变算法：不会更改区间内的元素内容：查找、计数、遍历、寻找极值

//迭代器：容器 和 算法 之间的 粘合剂

//算法 要通过 迭代器 才能访问 容器 中的的元素

//每个 容器 都有自己专属的 迭代器

//迭代器 类似于 指针

//常用迭代器：双向迭代器 和 随机访问迭代器

int main()

{

system("pause");

return 0;

}

79.vector存放内置数据类型

#include <iostream>

#include<vector>

#include<algorithm>

using namespace std;

//vector理解为数组，存放内置数据类型

//容器vector

//算法 for\_each遍历

//迭代器：vector<int>::iterator

void myPrint(int val)

{

cout << val << endl;

}

void test01()

{

//创建一个vector容器，数组

vector<int>v;

//向容器中插入数据

v.push\_back(10);

v.push\_back(20);

v.push\_back(30);

v.push\_back(40);

////通过迭代器访问容器中的数据

//vector<int>::iterator itBegin = v.begin();//v.begin()起始迭代器，指向容器中第一个元素

//

//vector<int>::iterator itEnd = v.end();//v.begin()结束迭代器，指向容器中最后一个元素的下一个位置

// //第一种遍历方式

//while (itBegin != itEnd)

//{

// cout << \*itBegin << endl;

// itBegin++;

//}

////第二种遍历方式

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

//{

// cout << \*it << endl;//\*it相当于对v进行取值操作

//}

//第三种遍历方式，利用STL提供遍历算法

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

}

int main()

{

test01();

system("pause");

return 0;

}

80.vector存放自定义数据类型

#include <iostream>

using namespace std;

#include<string>

#include<vector>

#include<algorithm>

//vector容器存放自定义数据类型

class Person

{

public:

Person(string name, int age)

{

this->m\_name = name;

this->m\_age = age;

}

string m\_name;

int m\_age;

};

void test01()

{

vector<Person>v;

Person p1("aaa", 10);

Person p2("abb", 10);

Person p3("aab", 10);

Person p4("aba", 10);

Person p5("baa", 10);

//向容器中添加数据

v.push\_back(p1);

v.push\_back(p2);

v.push\_back(p3);

v.push\_back(p4);

v.push\_back(p5);

//遍历容器中的数据

//it解引用后是person的数据类型\*it;<>内是什么数据类型，\*it解出来就是什么数据类型

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

{

cout << "Name: " << (\*it).m\_name << "Age" << (\*it).m\_age << endl;

cout << it->m\_name << it->m\_age << endl;//it本身是指针，可以通过->的方式拿到name，age

}

}

//存放自定义的数据类型的 指针

void test02()

{

vector<Person\*>v;

Person p1("aaa", 10);

Person p2("abb", 10);

Person p3("aab", 10);

Person p4("aba", 10);

Person p5("baa", 10);

//向容器中添加数据

v.push\_back(&p1);

v.push\_back(&p2);

v.push\_back(&p3);

v.push\_back(&p4);

v.push\_back(&p5);

//遍历容器中的数据

//it解引用后是person\*的数据类型\*it;<>内是什么数据类型，\*it解出来就是什么数据类型

for (vector<Person\*>::iterator it = v.begin(); it != v.end(); it++)//\*it是person的指针

{

cout << "\tName" << (\*it)->m\_name << " Age: " << (\*it)->m\_name << endl;

}

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

81.vector容器嵌套容器

#include<vector>

#include <iostream>

#include<algorithm>

using namespace std;

//容器嵌套容器=二维数组

void test01()

{

vector<vector<int>>v;//嵌套

//创建小容器

vector<int>v1;

vector<int>v2;

vector<int>v3;

vector<int>v4;

vector<int>v5;

//向小容器中添加数据

for (int i = 0; i < 4; i++)

{

v1.push\_back(i + 1);

v2.push\_back(i + 2);

v3.push\_back(i + 3);

v4.push\_back(i + 4);

v5.push\_back(i + 5);

}

//将小容器插入到大容器中

v.push\_back(v1);

v.push\_back(v2);

v.push\_back(v3);

v.push\_back(v4);

v.push\_back(v5);

//通过大容器遍历所有数据

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

{

//(\*it)就是一个小容器--vector<int>

for (vector<int>::iterator vit = (\*it).begin(); vit != (\*it).end(); vit++)

{

cout << \*vit << " ";

}

cout << endl;

}

}

int main()

{

test01();

system("pause");

return 0;

}

82.string容器-构造函数

#include <iostream>

using namespace std;

#include<string>

//string容器构造函数

//string是C++风格字符串，string容器 是一个类

//string容器是一个类，内部封装了char\*（字符串），管理此字符串，是一个char\*型的容器

//string类内部封装了很多成员方法：find copy delete replace insert

//string管理char\*所分配的内存，不用担心复制越界和取值越界等，由类内部进行负责

//string的构造函数

//string()创建一个空字符串

// string(const char\*s)使用字符串s初始化

// string(const string& str) 使用一个string对象初始化另一个string对象

// string(int n,char c)使用n个字符c初始化字符串

void test01()

{

string s1;//默认构造

const char\* str = "Hi";

string s2(str);//使用字符串s初始化

cout << "s2=" << s2 << endl;

string s3(s2);//使用一个string对象初始化另一个string对象

cout << "s3=" << s3 << endl;

string s4(10, 'b');//使用n个字符c初始化字符串,单引号注意

cout << "s4=" << s4 << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

83.string的赋值操作

#include <iostream>

using namespace std;

#include<string>

//string 字符串赋值

//等号的赋值方法assign实用

//

//

//

void test01()

{

string str1;

str1 = "hi!";//1char\*型字符串赋值给当前字符串

cout << "str1=" << str1 << endl;

string str2;

str2 = str1;//2拷贝赋值

cout << "str2=" << str2 << endl;

string str3;

str3 = 'a';//3单个字符赋值，单引号

cout << "str3=" << str3 << endl;

string str4;

str4.assign("Good luck");//4.assign赋值

cout << "str4=" << str4 << endl;

string str5;

str5.assign("Good luck", 4);//5.将字符串的前4个赋值给str5

cout << "str5=" << str5 << endl;

string str6;

str6.assign(str5);//6.assign的拷贝构造函数

cout << "str6=" << str6 << endl;

string str7;//7.10个w组成的字符串

str7.assign(10, 'w');

cout << "str7=" << str7 << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

84.string字符串拼接

#include <iostream>

using namespace std;

#include<string>

//string字符串拼接：实现在字符串 末尾 拼接 字符串

//重载版本很多

void test01()

{

string str1 = "我";

//1.

str1 += "爱玩游戏";

cout << "str1=" <<str1<< endl;

//2.一个字符用单引号

str1 += ':';

cout << "str1=" << str1 << endl;

//3.

string str2 = "LOL DNF";

str1 += str2;

cout << "str1" << str1 << endl;

//4.

string str3 = "I ";

str3.append("love playing");

cout << "str3=" << str3 << endl;

//5.前n个字符拼接到字符串

str3.append(" gamejaij", 5);

cout << "str3=" << str3 << endl;

//6.

str3.append(str2);

cout << "str3=" << str3 << endl;

//7.只截取一个字符串的0-3个字符添加

str3.append(str2, 0, 3);

cout << "str3=" << str3 << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

85.string的查找和替换

#include <iostream>

using namespace std;

//查找：查找指定的字符串是否存在

//替换：在指定的位置替换字符串

//1.查找find rfind

void test01()

{

//1.find 从左往右找

string str1 = "abcdegfgf";

int pos = str1.find("gf");//d在第四个位置0开始计数；如果不存在返回-1

cout << "pos=" << pos << endl;

if (pos == -1)

{

cout << "未找到字符串" << endl;

}

else

{

cout << "找到字符串" << endl;

}

//2.rfind从右往左查找 找到的是靠右的gf

int pos2 = str1.rfind("gf");

cout << "pos2=" << pos2 << endl;

}

//2.替换 replace

void test02()

{

string str1 = "abcdefg";

str1.replace(1, 3, "1111");//是将1-3号位的东西变为1111

cout << "str1=" << str1 << endl;

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

86.string的字符串比较

#include <iostream>

using namespace std;

//string字符串比较.compare

//比较方式：按字符的ASCll码比较;主要是比较是否相等

//=返回0

//>返回1

//<返回-1

void test01()

{

string str1 = "Hello";

string str2 = "Hell";

if (str1.compare(str2) == 0)

{

cout << "str1=str2" << endl;

}

else if (str1.compare(str2) == 1)

{

cout << "str1>str2" << endl;

}

else if (str1.compare(str2) == -1)

{

cout << "str1<str2" << endl;

}

}

int main()

{

test01();

system("pause");

return 0;

}

87.string容器的字符存取

#include <iostream>

using namespace std;

#include<string>

//string字符存取 [] 或者 at()

void test01()

{

//读取

string str = "hello";

cout << "str=" << str << endl;

//1.通过[]访问单个字符

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

{

cout << str[i] << endl;

}

//2.通过at方式访问单个字符

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

{

cout << str.at(i) << " ";

}

cout << endl;

//修改

str[0] = 'x';//将第0个字符改为x

cout << "str=" << str << endl;

str.at(1) = 'x';

cout << "str=" << str << endl;

}

int main()

{

test01();

system("pause");

return 0;

}

88.string的插入和删除

#include <iostream>

using namespace std;

#include<string>

//对string字符串进行插入和删除字符操作

void test01()

{

string str = "hi";

//插入

str.insert(1, "1111");//在第一个位置前插入1111,

cout << str << endl;//输出h1111i

//删除

str.erase(1, 4);//从第一个位置起删除4个

cout << str << endl;//变回hi

}

int main()

{

test01();

system("pause");

return 0;

}

89.string子串

#include <iostream>

using namespace std;

#include<string>

//string求子串

void test01()

{

string str = "abcdefg";

string substr = str.substr(1, 3);//求子串,从①号位开始，截取3个，不是截到第三个

cout << substr << endl;//输出bcd

}

void test02()

{

string email = "hello@sina.com";

//从邮件地址中获取用户名

int pos = email.find("@");

string username = email.substr(0, pos);

cout << username << endl;

}

int main()

{

test01();

test02();

system("pause");

return 0;

}

90.vector容器

#include <iostream>

using namespace std;

#include<vector>

//vector像数组，单端数组

//数组是静态空间，无法扩展空间，vector可以动态扩展

//动态扩展：并不是在原空间之后持续接新空间，而是找更大的内存空间，而是找更大的内存空间，然后将元数据拷贝新空间，释放原空间

//.rend()指向第一个元素的前一个位置

//.end()指向最后一个元素的最后一个

void printvector(vector<int>&v)

{

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

{

cout << \*it << " ";

}

cout << endl;

}

void test01()

{

vector<int>v1;//1默认构造 无参构造

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

{

v1.push\_back(i);

}

printvector(v1);

//2通过区间的方式构造

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

printvector(v2);

//3 n个elem方式构造

vector<int>v3(10, 100);//10个100组成的vector；第一个是个数，第二个才是赋值

printvector(v3);

//4拷贝构造

vector<int>v4(v3);

printvector(v4);

}

int main()

{

test01();

system("pause");

return 0;

}