# C++介绍

## C++和C的区别

1. c++完全支持面向对象

2. 比C更加安全

3. 支持过程化编程, 面向对象编程和泛型编程

## C++代码规范

## 代码编译

# 变量

# 基础数据类型

## 数值

### int

查看不同int的最大值，最小值以及每个占用的内存大小

#include <iostream>

using namespace std;

int main()

{

// 数值

// 整型

int16\_t int\_num = 100;

// int8

cout << "int8最大值: " << INT8\_MIN << endl;

cout << "int8最大值: " << INT8\_MAX << endl;

// int16

cout << "int16最小值: " <<INT16\_MIN << endl;

cout << "int16最大值: " << INT16\_MAX << endl;

// int32

cout << "int32最小值: " <<INT32\_MIN << endl;

cout << "int32最大值: " << INT32\_MAX << endl;

// int 64

cout << "int64最大值: " << INT64\_MIN << endl;

cout << "int64最大值: " << INT64\_MAX << endl;

//

cout << "int8: " << sizeof(int8\_t) << endl;

cout << "int: "<< sizeof(int) << endl;

cout << "int16: "<< sizeof(int16\_t) << endl;

cout << "int32: "<< sizeof(int32\_t) << endl;

cout << "int64: "<< sizeof(int64\_t) << endl;

}

**一般都是直接使用int， int就代指int32**

|  |  |  |  |
| --- | --- | --- | --- |
| 整型 | 最大值 | 最小值 | 字节数 |
| Int8 | 127 | -128 | 1 |
| Int16 | 32767 | -32768 | 2 |
| Int32 | 2147483647 | -2147483648 | 4 |
| Int64 | 9223372036854775807 | -9223372036854775808 | 8 |

### char

char使用单引号表示， 单引号表示字符

#include <iostream>

using namespace std;

int main()

{

char char\_num = -128;

cout << "char类型占用的字节大小: " << sizeof(char) << endl;

}

因为刚学不知道char有什么作用， 也不知道怎么查看他的最大最小值，通过sizeof查看了char类型占用的字节数为1， 由此可以看出char类型能表示的范围为-128-127

### bool

#include <iostream>

using namespace std;

int main()

{

// bool

cout << "bool占用的字节数: " << sizeof(bool) << endl;

cout << "bool(true): " << true << endl;

cout << "bool(false): " << false << endl;

/\*\*

bool占用的字节数: 1

bool(true): 1

bool(false): 0

\*\*/

}

**在c++中true就是1，false就是0**

### 浮点型

#### float

#### double

#### long double

## 非数值

### string

c++标准模板库中提供了string的数据类型，专门用来处理字符串，string是一个类型，这个类型的变量称为“string”对象，一个string对象可以用来保存一个具体的字符串。

string对象与基本数据类型有不同之处

例如：一个整数可以直接保存在一个整型变量中，而string对象所代表的的字符串保存在内存中，这段内存的首地址保存在string对象中，也就是说并不是字符串本身保存在string对象中，简而言之，string对象中只存了字符串在内存中的首地址，这样一来每个string对象的大小就是固定的，也就是首地址

### 声明字符串

要想使用string，必须包含头文件string

#include **<string>**

string 变量名;

在声明string变量的同时也可以进行初始化，既可以使用字符串常量进行初始化，也可以使用另一个字符串变量进行初始化，没有进行初始化的字符串变量的值为空串，即“”

注意，字符串常量使用双引号括起来

**int** main() {  
 string name;  
 string city = **"北京"**;  
 name = **"张三"**;  
  
 **return** 0;  
}

还可以使用字符串数组对string对象进行初始化

**int** main() {  
 string name;  
 **char** alias[] = **"C++程序"**;  
 name = alias;  
 **return** 0;  
}

还可以声明一个string数组， 即数组中每一个元素都是字符串

**int** main() {  
 string names[] = {**"张三"**, **"李四"**, **"王五"**};  
 cout << names[2] << endl;  
 **return** 0;  
}

### 字符串对象的操作

比较运算符

string对象之间可以使用<,<=,==,!=,>=,>运算符进行比较，大小的判断标准是根据ASCII字典排序的，而且大小写相关的

还可以使用“+”号进行连接

**int** main() {  
 string name = **"张三"**;  
 string address = **"北京"**;  
 cout << name + **"的家在"** + address << endl;  
 **return** 0;  
}

### 字符串对象用法（成员函数）

## 其他数据类型

### size\_t

### 枚举类型

### 自定义类型

### 指针类型

### 空类型

# 数据类型转换

无论是自动类型转换还是强制类型转换，都只是为了本次运算而进行的临时性转换，转换的结果也会保存到临时的内存空间，不会改变数据本来的类型或者值

## 自动数据类型转换

在赋值运算中，赋值号两边的数据类型不同时，需要把右边表达式的类型转换为左边变量的类型，这可能会导致数据失真，或者精度降低；所以说，自动类型转换并不一定是安全的。对于不安全的类型转换，编译器一般会给出警告。

#include <iostream>

using namespace std;

int main()

{

//数据类型转换

// 声明一个浮点型的价格

double double\_price = 65.9;

// 将浮点型转换成整型

cout << static\_cast<int>(double\_price) << endl;

// 将浮点型转成char

cout << static\_cast<char>(double\_price) << endl;

// 小数转成整数

cout << static\_cast<int>(10.0 / 3.0) << endl;

/\*\*

65

A

3

\*\*/

return 0;

}

## 强制数据类型转换

**float** cc = 4399.1;  
**int** ff;  
ff = (**int**) cc;

# 逻辑语句

## if判断

#include <iostream>

using namespace std;

int main()

{

double price = 99.99;

double account = 100;

if (account > price){

cout << "购买成功, 还能剩点钱" << endl;

}

else if (account == price){

cout << "刚刚够, 买完就没钱了." << endl;

}

else {

cout << "钱不够!!!" << endl;

}

return 0;

}

## switch判断

#include <iostream>

using namespace std;

int main()

{

int account = 1264;

// switch的条件里只能放整型或者枚举

switch (account)

{

case 1264:

cout << "可以买个AirPods" << endl;

break;

case 1999:

cout << "可以买个AirPods Pro" << endl;

break;

case 12000:

cout << "可以买个2020款的MacBook Pro 13.3" << endl;

break;

default:

cout << "买个冰棍凉快凉快吧!" << endl;

}

return 0;

}

**switch的执行逻辑是：**

当都不匹配时执行default(没有default就什么都不执行)，当遇到一个条件匹配的case后，如果没有break会继续往下执行(并且不会再和case进行比较，而是直接执行)，只有遇到break或者全部case执行完才会停止。

坑：

default可以在任意位置

case不能重复

## for循环

#include <iostream>

using namespace std;

int main()

{

/\*\*

for (循环变量; 循环条件; 变量更新){

循环体

}

\*\*/

// 循环变量可以直接声明在括号里

for(int a = 1;a <= 10; a++) {

cout << a << endl;

}

return 0;

for (;;){

cout << "死循环" << endl;

}

}

## while循环

#include <iostream>

using namespace std;

int main()

{

int start = 10;

while (start > 0)

{

cout << start << endl;

start -= 1;

}

}

## do-while循环

**先执行循环体，在执行循环条件**

#include <iostream>

using namespace std;

int main()

{

int start = 10;

do

{

cout << start << endl;

start--;

} while (start > 0);

}

# 命名空间

## 作用

为了避免名称定义冲突, 特别引入名称空间的定义, 即namespace的作用是为了消除同名引起的歧义

## 声明名称空间

namespace 名称空间名

{

命名空间内的各种声明(函数声明, 类声明.......)

}

## 引用(使用)命名空间

c++标准程序库中的所有标识符都定义在一个名为std的命名空间中, 假如没有using namespace std;则cin和cout, 都不能使用, 所以你要将代码卸载std命名空间中才能够使用这些标准的程序库

方式一：

#include <iostream>

using namespace std;

方式二：

在使用某个关键字时直接指定其所在的名称空间

#include <iostream>

int main()

{

std::cout << "第二种应用变量空间的方式" << std::endl;

}

这种两种方式可以和python的导包方式做一个类比，可以看出来是一样的

**import** operator  
**from** operator **import** add  
  
operator.***add*(**10, 20**)  
  
*add*(**10, 20**)**

# 数组

## 概念

数据也是一个变量，由**数据类型相同**的一组元素组成

从内存上来看，数据就是**一块连续的空间**

## 数组的结构和组成

标识符：数组的名称

数组元素：数组中存放的数据

元素下标：对数组中的元素进行编号

元素类型：数据中元素的类型

## 一维数组

### 声明语法

#include <iostream>

using namespace std;

int main()

{

dataType arrayName[size] = {元素1, 元素2};

}

如果size不能是变量， 可以是常量，在不同的c++版本中可能不一样

#### 声明数组的常见方式

#include <iostream>

using namespace std;

int main()

{

// size可大不可小

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

int months[12] = {1, 2, 3, 4, 5, 6};

// size可以不写, 动态根据元素的个数创建

int days[] = {1, 13, 25, 30};

int sceond[] = {};

}

**如果在声明的时候没有往数组中添加元素，那么之后是不可以像下面这样赋值的，只能够通过索引一个一个的赋值**

#include <iostream>

using namespace std;

int main()

{

int nums[5];

nums = {1,2,3,4,6}

}

#### 使用循环创建数组

#include <iostream>

using namespace std;

int main()

{

const int SIZE = 5;

int nums[SIZE];

for (int n = 0; n < SIZE; n++){

cout << "请输入第" << n + 1 << "个元素: ";

// 从命令行获取输入, 通过下标将元素放到数组中

cin >> nums[n];

}

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

cout << nums[i] << endl;

}

}

### 计算数组的长度

只适用于数组内元素数据类型一致的情况

sizeof(nums) / sizeof(int)

因为创建内存时已经声明了数组中存放的数据类型以及元素个数，就会根据这些信息会内存中开辟空间

此时，总大小 / 每个元素的大小 = 元素的个数

### 练习题

1. 求出数组的总和以及平均值

int main()

{

int nums[] = {8, 4, 2, 1, 23, 344, 12};

int sum = 0;

int length = sizeof(nums) / sizeof(int);

for (int n = 0; n < length; n++)

{

sum += nums[n];

}

cout << "和: " << sum << endl;

cout << "平均值: " << sum / length << endl;

}

1. 求数组中的最大值, 最小值

#include <iostream>

using namespace std;

int main()

{

int nums[] = {8, 4, 2, 1, 23, 344, 12};

int length = sizeof(nums) / sizeof(int);

int max\_assume = 0;

int min\_assume = 0;

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

{

if (nums[n] > nums[max\_assume])

{

max\_assume = n;

}

if (nums[n] < nums[min\_assume])

{

min\_assume = n;

}

}

cout << "最大值: " << nums[max\_assume] << endl;

cout << "最小值: " << nums[min\_assume] << endl;

}

1. 求数组中的奇数和偶数的个数

#include <iostream>

using namespace std;

int main()

{

int nums[] = {8, 4, 2, 1, 23, 344, 12};

int length = sizeof(nums) / sizeof(int);

// 奇数

int odd\_count = 0;

// 偶数

int even\_count = 0;

for (int n = 0; n < length; n++)

{

int quotient = nums[n] % 2;

if (quotient == 0)

{

even\_count++;

}

else

{

odd\_count++;

}

}

cout << "奇数个数: " << odd\_count << endl;

cout << "偶数个数: " << even\_count << endl;

}

1. 根据输出的元素，找该元素在数组中的下标，不存在时返回-1

#include <iostream>

using namespace std;

int main()

{

int nums[] = {8, 4, 2, 1, 23, 344, 12};

int length = sizeof(nums) / sizeof(int);

int target\_index = -1;

int find;

cout << "请输出要找的元素: ";

cin >> find;

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

{

if (nums[i] == find)

{

target\_index = i;

break;

}

}

cout << "要找的元素下标为: " << target\_index << endl;

}

1. 冒泡排序

#include <iostream>

using namespace std;

int main()

{

int nums[] = {8, 4, 2, 1, 23, 344, 12};

int length = sizeof(nums) / sizeof(int);

for (int n = 0; n < length; n++)

{

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

{

if (nums[i] > nums[i + 1])

{

int tmp = nums[i];

nums[i] = nums[i + 1];

nums[i + 1] = tmp;

}

}

}

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

{

cout << nums[i] << "\t";

}

}

## 二维数组

### 什么是二位数组

元素有两个下标的数组成为二维数组， 二维数组用于存储逻辑上按行，列组织在一起的书

#include <iostream>

using namespace std;

int main()

{

string names[3] = {"刘备", "关羽", "张飞"};

string courses[3] = {"语文", "数学", "英语"};

const int ROW = 3;

const int COL = 3;

double scores[ROW][COL];

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

{

for (int n = 0; n < COL; n++)

{

cout << "请输入" << names[i] << "的" << courses[n] << "成绩: ";

cin >> scores[i][n];

}

}

cout << "\t";

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

{

cout << names[i] << "\t";

}

cout << endl;

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

{

cout << courses[i] << "\t";

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

{

cout << scores[j][i] << "\t";

}

cout << endl;

}

}

## 数组的替代品

### 什么是vector

1. 动态数组， 可以在运行阶段设置长度
2. 具有数组的快速索引方式
3. 可以插入和删除元素

### 定义和初始化

# 指针

## 什么是指针

指针是一种数据类型，指针是存放数据的内存单元地址（指针是一个值为内存地址的变量）

## 指针的作用

1. 使程序简介，紧凑，高效
2. 有效的表示复杂的数据结构
3. 动态分配内存
4. 获得多于一个的函数返回值

## 指针的定义初始化

指针变量与其他变量一样，遵循先定义后使用的原则，在定义指针变量的同时也可以对其进行初始化。

**格式**

数据类型符 \*指针变量名[=初始地址值]

**功能**

定义指向“数据类型”的变量或数组的指针变量，同时为其赋值。

**说明**

1. “\*”表示定义的是一个指针变量，指针变量的前面必须有“\*”

例如**int \*p, i;**表示定义了一个整型指针变量p，其中存放着一个整型变量的地址，同时还定义了一个整型的变量i，“\*”只对p起作用。

1. 在定义指针变量的同时也可以定义普通变量或数组。
2. “数据类型符”是指针变量所指向变量的数据类型，可以是任何基本数据类型，也可以是其他数据类型。**这一数据类型符不是指针变量中存放的数据类型，而是指针变量所指向的变量或数据的数据类型**

例如

float \*p1; // p1是浮点型指针变量, 即指向浮点型数据

char \*p2; // p2是字符型指针变量, 即指向字符型数据

1. “初始地址值”通常是“&变量名”“&数组元素”或“一维数组名”这里的变量或数组必须是已定义的
2. 在定义指针变量是，可以只给部分指针变量赋初值。
3. 指针变量的初始化，除了可以是已定义变量的地址，也可以是已初始化的同类型的指针变量， 也可以是NULL（空指针）

例如

int x, y, \*px = &x; // 在定义指针变量px的同时进行初初始化

int \*py = &y, \*pz = px; // 变量y已定义, 指针变量px已经保存的事变量x的地址

char \*pc = nullptr; // 指针变量pc初始书为NULL

1. 指针变量初始化时，指针变量的“数据类型符”必须与其“初始地址”中保存的数据类型相同

## 指针变量定义和初始化的例子

char x;

char \*pointer = &x;

float a, b[5], \*p1 = &a, \*p2 = b;

## 指针和指针变量的区别

指针和指针变量的区别，就是变量值和变量的区别，为了表示指针变量和其他指向变量之间的关系，用指针运算符“\*”表示

也就是说在变量前加“\*”的都是指针变量

## 取地址符(&)

int main()

{

// 取地址符

int num = 10; // 开一块内存, 存放整型10

int\* ptr\_num; // 再开一块内存(\*表示指针类型)

ptr\_num = &num; // 取num的地址赋给ptr\_num, 所以ptr\_num中存储的是num的内存地址

}

## 间接运算符(\*)

int main()

{

// 取地址符

int num = 10; // 开一块内存, 存放整型10

int\* ptr\_num; // 再开一块内存(\*表示指针类型)

ptr\_num = &num; // 取num的地址赋给ptr\_num, 所以ptr\_num中存储的事num的内存地址

\*ptr\_num = 222; // 此时的\*代表根据地址找到内存空间, 此时直接修改了ptr\_num这个指针中存放的内存地址的值

cout << num << endl;

return 0;

}

## 空指针

当不给指针赋值时

#include <iostream>

using namespace std;

int main()

{

double \* ptr\_double;

cout << ptr\_double << endl;

return 0;

}

输出的结果是0x7ffeef75c568，还是指向了一个内存地址，江湖人称**野指针**

这时使用nullptr可以解决这个危险的操作

#include <iostream>

using namespace std;

int main()

{

double \* ptr\_double = nullptr;

cout << ptr\_double << endl;

return 0;

}

## const与指针共同使用

与c语言一样， c++也能使用const限定访问的权限，告诉编译器，他所修饰的变量是不能修改的， 也就是说不能作为左值使用。

程序中使用指针时，涉及的量有两个：

一个是之指针本身

另一个是指针所指向的内容

当const与指针共同使用时，其确切的含义是什么？

是说指针本身不能改变？，即不能当其指向其他地址？，还是指针所执行的内容不能改变，抑或两个含义都有？

当const与指针共同使用时，其书写的位置不同，语句的含义不同

const修饰指针变量时，基本含义如下

1. 如果唯一的const位于符号\*的左侧，表示指针所指向的数据是常量，数据不能通过本指针改变，但可以通过其他方式进行修改；指针本身是常量，可以指向其他的内存单元。
2. 如果唯一的const位于符号\*的右侧，表示指针是常量，不能让该指针执行其他的内存单元，指针所指向的数据可以通过该指针进行修改
3. 在符号\*的左右有一个const时，表示指针和指针所指向的数据都是常量，既不能让指针指向其他的地址，也不能通过该指针修改其所指向的内容。

示例

#include **<iostream>  
  
using namespace** std;  
  
**int** main() {  
 **int** a1 = 3; *// 普通变量* **const int** a2 = a1; *// 数据是常量，不能修改* **int** \*a3 = &a1; *// 普通指针指向普通变量* **const int** \*a4 = &a1; *// 普通指针指向常量数据* **int** \***const** a5 = &a1; *// 常量指针指向普通数据* **int const** \***const** a6 = &a1; *// 常量指针指向常量数据* **const int** \***const** a7 = &a1; *// 常量指针指向常量数据* **return** 0;  
}

## 指针和数组

## 指针运算

一般将指针的运算称为指针平移

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

int \*ptr\_score;

ptr\_score = &socre[4];

cout << \*(ptr\_score - 2)<< endl; // 对指针-2

## 动态分配内存

这种内存分配是在程序运行期间进行的，故称为“动态内存分配”，相对的在编译时确定内存的大小可以成为“静态内存分配”

**静态内存分配**

在编译时就指定好了的

**动态内存分配**

在运行时，执行的代码会使用new关键字申请内存

1. 指针的真正用武之地： 在运行阶段分配未命名的内存以存储值
2. 在此情况下，只能通过指针在访问内存

使用delete释放内存

1. 与new配对使用
2. 不要释放已经释放的内存
3. 不能释放声明变量分配的内存

在c++中使用new运算符实现动态内存分配，如下

p = **new** T;

其中T为任意的类型名，p是类型为T\*的指针，这样语句会动态分配出一片大小为sizeof(T)字节的内存空间，并且将该内存空间的起始地址赋值给指针p，如下

**int** main() {  
 **int** \*p; *// 声明一个int型的指针 p* p = **new int**; *// 申请一个int大小的内存， 并赋值给指针p* \*p = 10; *// 使用指针p往内存空间中填值* **return** 0;  
}

使用new运算符还可以动态分配一个任意大小的数组

**int** main() {  
 **int** \*pArray; *// 声明指向数字的指针* **int** i = 5;  
 pArray = **new int**[i \* 20]; *// 申请100个int可用的内存地址* pArray[0] = 20; *// 使用指针往数组中赋值* pArray[99] = 20;  
 **return** 0;  
}

**使用下标操作时如果出现超出范围，在编译时不会提示，而显示编译成功，但是在运行时会有意料之外的结果**

## 释放动态分配的内存

使用new关键字动态动态申请的内存，需要再使用完之后进行释放，c++提供了delete运算符，用来释放动态分配的内存

delete运算符后面的指针必须指向动态分配的内存空间，否则很可能出错

**int** main() {  
 **int** oneInt = 6;  
 **int** \*p = &oneInt;  
 cout << \*p << endl;  
 **delete** p; *// 此时释放p会报错，因为p不是动态分配的* **int** \*q = **new int**;  
 \*q = 100;  
 cout << \*q << endl;  
 **delete** q; *// 可以正常的释放* **return** 0;  
}

释放动态分配的数组

**int** main() {  
 **int** \*p = **new int**[5];  
 p[0] = 100;  
 **delete**[]p;  
  
 **return** 0;  
}

使用new关键字动态分配的内存空间，一定要用delete运算符进行释放，否则，即使程序运行结束，这部分内存空间认证不会被操作系统回收，从而程序被白白浪费到的内存垃圾，这种现象称为“内存泄露”， 当一个程序不停的进行内存分配但总是不释放，那么可用的内存就会越来越少，导致操作系统运行速度变慢，甚至影响其他程序的正常运行，严重时必须重启计算机才能解决.

new和delete的使用规则

1. 不要使用delete释放不是new分配的内存
2. 不要使用delete释放同一个内存两次
3. 如果使用new[]为数组分配内存，则对应使用delete[]释放内存
4. 对空指针使用的delete是安全的

## 程序的内存分配

### 栈区（stack）

由编译器分配释放，一般存放函数的参数值，剧本变量的值等

操作方式类似于数据结构中的栈(先进后出)

### 堆区（heap）

一般由程序员分配释放，若程序不释放，程序结束时可能由操作系统回收

注意：与数据结构中的堆是两回事，分配方式类似链表

### 全局区（静态区-static）

全局变量和静态变量是存储在一起的

程序结束后由系统释放

### 文字常量区

常量字符串就放在这里，程序结束右系统释放

### 程序代码区

存放函数体的二进制代码

## 使用指针实现数组逆序

#include <iostream>

using namespace std;

int main()

{

int score[] = {70, 60, 30, 100, 5, 222};

int len = sizeof(score) / sizeof(int);

int \*ptr\_start = score; // 前指针

int \*ptr\_end = score + len - 1; // 后指针

int temp;

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

{

temp = \*ptr\_start; // 将前指针的指向值放在临时空间中

\*ptr\_start = \*ptr\_end; // 前指针指向的内存中放后指针的值

\*ptr\_end = temp; // 后指针指向的内存放临时空间中的值

ptr\_start++; // 前指针向后移

ptr\_end--; // 后指针向前移

}

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

{

cout << score[i] << endl;

}

}

# 引用（reference）

## 什么是引用

为对象起了另外一个名字（引用即别名）

**引用必须被初始化**

注意

1. 引用并非对象，只是为一个引用存在的对象起一个别名
2. 引用只能绑定在对象上，不能与字面值或某个表达式的计算结果绑定在一起（不能引用具体的某个值， 常量除外 ）
3. 引用必须初始化，所以使用引用之前不需要测试其有效性，因此使用引用可能会比使用指针效率高
4. 不能有空引用，即引用必须指向某个已存在的内存区域的首地址
5. 不能声明引用的引用
6. 引用名也必须是一个合法的标识符，其命名规则和普通变量相同
7. 不能用常量初始化引用
8. 不能用表达式初始化引用（除非表达式的返回值是某个变量的引用）
9. 同一个变量的引用可以有多个
10. 定义引用时，可以在定义的前面加上const关键字，表明该引用是常引用

int oneInt = 10;

const int &aname = oneInt;

常引用和普通引用的区别在于，不能通过应用去修改其引用的值

## 引用和指针的关系

1. 引用对指针进行了简单的封装，底层仍然使用的是指针
2. 获取引用地址时，编译器会进行内部转换

#include <iostream>

using namespace std;

int main()

{

// 引用

int num1 = 108;

int& rel\_num = num1;

rel\_num = 118;

cout << &num1 << "\t" << &rel\_num << endl;

// 用指针实现上面的引用

int num2 = 108;

int \*ptr\_rel\_num = &num2;

\*ptr\_rel\_num = 118;

cout << &num2 << "\t" << ptr\_rel\_num << endl;

}

# 函数

## 函数的分类

1. 内置函数（STL和Boost C++）

C++标准函数

1. 语言支持
2. 国际化
3. 通用工具
4. 诊断
5. 数值
6. 输入/输出
7. 自定义函数

## 函数三要素

1. 返回值类型
2. 函数名
3. 参数列表

## 函数定义

#include <iostream>

using namespace std;

// 1. 函数原型

int sum(int, int);

int main()

{

// 3. 函数调用

cout << "result: " << sum(10, 11) << endl;

}

// 2. 函数实现

int sum(int a, int b)

{

return a + b;

}

## 函数原型

1. 在函数原型中，可以省略形参的名字，而只给出他的类型及默认值

## 参数传递的机制

### 默认值参数

提供默认值时必须按照从右至左的顺序提供，即有默认参数的形参必须在形参列表的最后

函数默认值可以写在声明函数的地方，也可以写在定义函数的地方，但不能两个地方都写

#include <iostream>

using namespace std;

void defaultValue(int = 2, double = 3.2);

int main()

{

defaultValue();

defaultValue(10);

}

void defaultValue(int a, double b)

{

cout << a \* b << endl;

}

### 按值传递

给函数传递参数时，参数值不会直接传递给函数，而是先制作参数的副本，存储在栈上，在使这个副本可用于函数，而不是使用初始值。

换句话说，参数传递的不是引用，而是副本，直接的影响就是，其中修改另一个不会跟着修改。

#include <iostream>

using namespace std;

void change(int);

const double PI = 4 \* atan(1.0);

int main()

{

int num = 10;

cout << "out" << "\t" << num << "地址" << "\t" << &num << endl;

// out 10 地址 0x7ffee8e9f55c

change(num);

cout << num << endl;

// 10

}

void change(int num)

{

num++;

cout << "in" << "\t" << num << "地址" << "\t" << &num << endl;

// in 11 地址 0x7ffee8e9f52c

}

## 引用在函数中的使用

### 函数的参数

在程序中不仅能定义变量的引用，还可以将引用用在函数中，引用既可以作为**函数的参数使用**，也可以作为**函数的返回值使用**

在c++中函数调用时参数的传递有两种方式

1. 传值（传递对象的值）
2. 传引用（传递对象的首地址值）

如果函数的形参不是引用，那么调用时实参传递给形参通常采用值传递的方式，**即将实参的值拷贝给形参**，在函数执行过程中，都是对这个拷贝的值进行操作的，函数执行完毕后，形参的值并不会拷贝会实参，**也就是说函数内部对形参的改变不会硬性外部实参的值**

**为什么函数内部对形参的改变不会硬性外部实参的值？**

调用一个函数时，他的形参即函数内部定义的变量被分配在成为栈中的内存空间，这与实参所在的空间不同，传值时，将实参拷贝带栈中对应形参的地址中，函数内部对形参操作，其实是对这个地址进行操作，而不是对实参所占用的地址进行操作，函数执行完毕后，也不是将这个地址中的值拷贝出去， 所以实参的值不受影响。

如果函数的形参是引用， 则调用时实参传递给形参采用的是引用的方式

函数调用时，实参对象名传递给形参对象名，形参对象名就成为实参对象的别名，即形参是对象实参的引用， 他们是等价的，代表同同一个对象，也可以看做将实参的地址传递给了形参，在函数内部对形参操作，都是对这个地址的内容进行的，相当于实参的值进行了操作，所以当函数执行完毕后返回，实参的变化被保留下来。

小窍门

传值的方式意味着，数据需要从堆中拷贝一个到栈中，这个开销其实是值得考虑的，当保证一个对象不能被修改，但是又不能使用传值的方式，那就可以使用const修饰引用，这样既避免了形参到实参的拷贝，也能方式实参的函数执行过程中被改变

### 函数的返回值

引用还可以是函数的返回值，变量是可作为赋值语句中左值存在，既然引用可以看做是变量的别名，那么引用也可作为赋值语句的左值，即可以放在赋值号的左侧。

在赋值语句中赋值号左侧的表达式成为左值，右侧的表达式成为右值， 左值表达式既能出现在赋值号的左侧，也能出现在赋值号的右侧，变量，指针，引用都可以作为左值

返回引用的函数原型如下：

数据类型 & 函数名(参数列表){

函数体

return

}

示例

#include <iostream>

using namespace std;

int & change(int & a)

{

return a;

}

int main()

{

int num = 100;

change(num) = 300;

cout << num << endl;

}

## 内联函数

在程序中使用函数的目的之一就是减少程序代码，实现程序代码的共享，从而提高程序的开发效率和可维护性。

但是在调用函数时，

1. 需要保存主调函数的现场和返回值，
2. 要在栈中为形参和局部变量分配内存空间，要实现实参与形参的参数传递，
3. 然后程序转移到被调用函数的起始位置继续执行，被调用函数执行结束后，
4. 要从栈中回收形参和局部变量占用的存储空间，
5. 要恢复主调函数的现场，取出返回地址，有返回值的要将返回值赋值给函数调用的本身，
6. 最后在返回地址处开始继续执行

总之，调用函数时会带来程序运行时间上的开销，使程序的执行效率变低。

如果一个函数的函数体积较大时，那么调用函数集返回时所付出的代价，与代码共享所得到的的回报相比，显得微不足道，但如果是只有很少几条语句的函数，这个函数调用产生的额外开销比函数本身执行的时间相比，就不能忽略了。特别是频繁调用时，函数调用导致时间开销可能会使程序运行时明显变慢。

为了避免这中频繁的函数调用与返回，C++引入了内联函数的概念，**使用内联函数，编译器在编译时并不产生函数调用，而是将程序中出现的每一个内联函数在调用表达式直接用该内联函数的函数体进行替换，就像整个函数体在调用处被重写一遍一样**，很明显，使用内联函数会使最终可执行程序的体积变大，这是以空间消耗换来时间开销的降低。

内联函数应该定义在前，调用在后，定义时只需要在函数有返回值类型前加上inline关键字。

#include **<iostream>  
  
using namespace** std;  
  
*// 内联函数***inline void** func() {  
 cout << **"这是一个内联函数"** << endl;  
}  
  
**int** main() {  
 func();  
 **return** 0;  
}

## 函数的重载

程序中经常出现对不同数据类型的量进行同类操作的情况

### 函数重载的目的

1. 使用相同的函数名调用功能相似的函数
2. 减少命名空间的浪费

例如：

找出两个数值之间的较大者并返回，这两个数值可能都是整数，也可能是浮点数，还有可能是字符串

当然对于字符串类型，可以按照字典序定义字符串的大小。

#include **<iostream>  
  
using namespace** std;  
  
**int** biggerInt(**int** x, **int** y) {  
 **if** (x > y) {  
 **return** x;  
 } **else** {  
 **return** y;  
 }  
}  
  
**float** biggerFloat(**float** x, **float** y) {  
 **if** (x > y) {  
 **return** x;  
 } **else** {  
 **return** y;  
 }  
}  
  
**int** main() {  
 **int** retInt = biggerInt(10, 20);  
 **float** retFloat = biggerFloat(3.5, 8.7);  
 cout << retInt << endl;  
 cout << retFloat << endl;  
 **return** 0;  
}

为了区分各个函数，需要命名不同的函数，调用时也要对号入座。

面向对象程序设计语言的特点之一就是允许函数重载，C++也不例外。

所谓函数重载是指在**程序的同一范围内**声明几个功能类似的同名函数。

例如

在同一个类中声明3个求两者中较大值的同名函数

有了函数重载的机制，C++程序员在编写程序时，可以**对完成类似功能的不同函数统一命名**，**减少了命名空间的浪费**。

因为多个函数名的名字完全相同，所以为了能有效的区分它们，各个函数的参数列表不能完全相同，函数名加上参数表做一个签名，也就是说，重载的各个函数的签名是不同的，所以区分它们也是可行的。

在调用同名函数时，编译器确定调用的函数版本的依据是什么？

编译器根据调用时调用语句中的实参类型及个数，来确定调用的具体函数的版本。

调用函数的语句给出的实参必须和参数表中的形参的个数和类型一一匹配。

换句话说调用语句要提供给编译器足够的判定信息。

实现函数重载必须满足下列条件之一

1. 参数表中对象的参数类型不同
2. 参数表中参数的个数不同

如果函数的参数表中不同类型参数的次序不同，也符合上面所说的条件。

要注意的是，**返回值类型不能用来区分函数**，也就是说，如果两个函数的名字和参数列表都是一样的，仅仅是返回值类型不同，则这两个函数不是重载，编译器会认为他们是重复定义，编译时会报错

具有相同函数名的两个及以上的函数，其形参的个数或者类型不同，编译器根据实参和形参的类型及个数的最佳匹配，自动确定调用哪个函数，这就是函数的重载及其调用

#include **<iostream>  
  
using namespace** std;  
  
**int** bigger(**int** a, **int** b) {  
 **if** (a > b) {  
 **return** a;  
 } **else** {  
 **return** b;  
 }  
}  
  
**double** bigger(**double** a, **double** b) {  
 **if** (a > b) {  
 **return** a;  
 } **else** {  
 **return** b;  
 }  
}  
  
**int** main() {  
 **double** ret = bigger(10.30, 100.12);  
 cout << ret << endl;  
 **return** 0;  
}

在有些情况下，调用的语句中的实参与函数的形参的类型不完全匹配，但存在赋值兼容的情况，此时编译器也可以确定要调用那个函数（调用函数时进行必要的类型提升）

# C++语言程序结构

c++程序以.cpp作为文件扩展名，文件中包含若干个类和若干个函数

程序中必须有且仅有一个主函数main，这是程序执行的总入口，主函数也成为主程序，程序从函数的开始处执行，按照控制结构，一直执行到结束。

程序结束通常时遇到一下两种情形之一

1. 在主函数中遇到return语句
2. 执行到主函数最后面的括号

主函数中可以调用程序中定义的其他函数，但其他函数不能调主函数，主函数仅是系统为执行该程序是所调用的

c++程序中，仍沿用c的注释风格，注释有两种

1. 从/\*开始，到\*/结束，这之间的都被视为注释
2. 从// 直到行尾，都是注释

# 面向对象

## 基本概念

### 结构化程序设计

在面向对象程序设计方法出现之前，软件界广泛流行的主流设计方式之一是结构化程序设计方法，也称面向过程的设计方法。

面向过程编程强调数据类型和程序结构，注重程序的易读性，可靠性和可维护性

在结构化程序设计中，采用自顶向下，逐步求精及模块化的思想，将复杂的大问题层层分解为许多简单的小问题，也就是说将问题看所一系列需要完成的任务，使用函数来完成这些任务，吧逻辑功能完全独立或相对独立的程序部分设计成函数，每个函数要完成的功能不多且明确

所以解决问题的焦点集中于函数的设计和函数之间相互调用上，程序员要考虑的就是如何将整个程序分解成一个个函数，那些函数直接要互相调用，以及每个函数内部将如何实现。程序设计的过程基本上就是编写函数的过程。

结构化程序设计的里面是将复杂的功能模块化繁为简，从问题的总体目标开始，忽略低层的实现细节，先专心构造高层函数的结构，然后再一层层地分解和细分，设计者把握主题，避免一开始就陷入繁杂的细节当中，程序员在编写函数时也可以只集中考虑焦点问题， 这样复杂的设计过程就变得简单明了，实现的模块也方便进行简单的验证。

## 面向对象程序设计的概念和特点

### 面向对象思想的提出

所谓面向对象的程序设计方法就是使**分析，设计和实现**尽可能的接近人们认识一个系统的方法。

通常包括3个方面

1. 面向对象的分析
2. 面向对象的设计
3. 面向对象的程序设计

面向对象的的程序设计方法要分析待解决的问题中有哪些类型事物，每类事物都有哪些特点，不同的事物种类之间是什么关系，事物之间如何相互作用

面向对象技术把问题看成是相互作用的事物集合，也就是对象集合，对象具有两个特征

1. 状态（状态是指对象本身的信息，也称为属性）
2. 行为（行为是对对象的操作）

通过多食物的抽象找出同一类对象的共同属性（静态特征）和行为（动态特征），从而得到类的概念。

对象是类的具象，类是对象的抽象，在面向对象的技术中，数据表现为数据，用属性值来描述对象的状态，而把对象的操作定义为一个函数，这样对象就是数据加函数，可以将现实生活中的对象经过抽象，映射为程序中的对象

### 面向对象程序设计的特点

1. 抽象
2. 封装
3. 继承
4. 多态

#### 抽象

将同一类事物的共同特点概括出来，这个过程就叫做抽象

类是是对现实世界中客观事物的抽象，通常将众多的具有相同属性的事物进行归纳，划分为某个类，例如：一个单位有很多员工，将员工的共同点抽取出来，可以定义一个员工类

#### 封装

每个给定类的对象包含了这个类所规定的若干私有成员，共有成员以及保护成员，这个过程就叫做封装

优点

1. 对象的内部信息对外界是隐蔽的，不允许外界直接访问对象的属性
2. 合理的安排数据的可访问范围
3. 减少程序不同部分之间的耦合度，从何提高代码扩充，代码修改，代码复用的效率

#### 继承

所谓继承就是在编写一个新类的时候，以现有的类作为基础，使得新类从现有的类派生而来，从而达到带代码扩充和代码复用的目的。

#### 多态

多态是指不同的种类的对象都具有名称相同的行为，而具体行为的实现方式却有所不同，多态性起源于一个希腊的单词，意思是“许多形态”

优点：

1. 多态可以保证对不同类型的数据进行等同的操作，名称空间更加宽松，
2. 多态还有一个重要的特点，即使用相同的操作名，能根据具体的对象自动选择对象的操作

多态仅用于通过继承而相关联的引用或指针，因此通过继承而相关的类型成为多态类型

## 类的初步知识

### 类的定义

**class** User {  
 访问范围说明符:  
 成员变量1;  
 成员变量2;  
   
 成员函数声明1;  
 成员函数声明2;  
 访问范围说明符;  
 更多成员变量;  
 更多成员函数声明;  
};

#### 类名

1. Class关键字开头，后面是类名
2. 类名的定义遵循一般标识符的命名规则（字母，数字，下划线组合，大小写敏感）
3. 类名不能重复
4. 类定义以“;”结束
5. 大括号中的部分称为类体
6. 类中可以不含有任何成员变量和成员函数，这样的类称为空类
7. 定义类时系统并不为类分配存储空间，只是把类看作一种模板或样板（类可以看作用户自定义的一种数据类型）
8. C++98标准下，类声明的任何成员不能使用auto，extern和register关键字进行修饰
9. 类中成员不能在声明时进行初始化（不能在类体内声明成员变量的同时给他赋值，类中的静态成员除外，通常对象的成员变量赋值工作都是由构造函数完成的）

#### 访问范围说明符

在类定义中可以 以任意次序出现任意多次

1. Public（公用）
   1. 可以被任意实体访问
2. Private（私有）
   1. 只允许本类的成员函数访问
3. Protected（保护）
   1. 只允许子类及本类的成员函数访问

#### 成员变量

1. 是类中的一类成员
2. 个数没有限制
3. 也称为数据成员
4. 成员变量的声明方式与普通变量的声明方式相同
5. 代表对象的属性
6. 如果成员变量时私有的，那么在类外不能访问（通常在类中会为每个成员变量定义一对访问函数，一个用来设置成员变量的值，常称为设置函数，一个用来获取成员变量的值，常称为获取函数）
7. 成员变量的类型不能是本类类型（类C中不能定义类C的成员变量），但可以定义类C的指针和引用

#### 成员函数

1. 是类中的一类成员
2. 个数不限
3. 声明方式和普通函数一样
4. 代表对该类对象所含数据进行操作的方法
5. 既可以在类体定义，也可以在类体外定义（如果成员函数定义在类体的内部，则默认是内联函数，也可以在类体内部声明函数，并加上inline关键字，然后在类体外部给出函数定义，这样的成员函数也是内联函数）
6. 原型中仅需指出参数列表中个形参的类型，不必给出参数的名字
7. 在定义成员函数时，类名，函数名，参数类型，参数名，返回值缺一不可

#### 类成员按照功能划分

1. 成员变量
2. 成员函数

#### 类成员按照权限划分

1. 共有成员
2. 私有成员
3. 保护成员

#### 全局函数

C++中还可以定义不是任何类的成员函数，这样的函数称为“全局函数”

#### 类作用域运算符

如果成员函数定义在类体外，则类体必须要有函数原型，类体外函数定义的前面必须用“类名::来限定

类名是成员函数所属类的名字，符号::是类作用域运算符，表明她后面的成员函数是属于类型标示的这个类的，返回值类型就是这个成员函数返回值的类型。

#### 类定义示例

#include **<iostream>**#include **<string>  
  
using namespace** std;  
  
**class** myDate {  
**public**:  
 myDate(); *// 构造函数* myDate(**int**, **int**, **int**); *// 构造函数* **void** setDate(**int**, **int**, **int**); *// 设置日期* **void** setDate(myDate); *// 设置日期* myDate getDate(); *// 获取日期* **void** setYear(**int**); *// 设置年* **int** getMonth(); *// 获取月* **void** printDate() **const**; *// 打印日期***private**:  
 **int year**, **month**, **day**;  
};  
  
myDate::myDate() {  
 **year** = 1970, **month** = 1, **day** = 1;  
}  
  
myDate::myDate(**int** y, **int** m, **int** d) {  
 **year** = y, **month** = m, **day** = d;  
}  
  
  
**void** myDate::setDate(**int** y, **int** m, **int** d) {  
 **year** = y, **month** = m, **day** = d;  
}  
  
**void** myDate::setDate(myDate oneDate) {  
 **year** = oneDate.**year**, **month** = oneDate.**month**, **day** = oneDate.**day**;  
}  
  
myDate myDate::getDate() {  
 **return** \***this**;  
}  
  
**void** myDate::setYear(**int** y) {  
 **year** = y;  
}  
  
**int** myDate::getMonth() {  
 **return month**;  
}  
  
**void** myDate::printDate() **const** {  
 cout << **year** << **"/"** << **month** << **"/"** << **day** << endl;  
}  
  
  
**class** Student {  
**public**:  
 **void** setStudent(string, myDate);  
  
 **void** setName(string);  
  
 string getName();  
  
 **void** setBirthday(myDate);  
  
 myDate getBirthday();  
  
 **void** printStudent();  
  
**private**:  
 string **name**;  
 myDate **birthday**;  
};  
  
**void** Student::setStudent(string s, myDate b) {  
 **name** = s;  
 **birthday** = b;  
}  
  
**void** Student::setName(string s) {  
 **name** = s;  
}  
  
string Student::getName() {  
 **return name**;  
}  
  
**void** Student::setBirthday(myDate oneDate) {  
 **birthday**.setDate(oneDate);  
}  
  
myDate Student::getBirthday() {  
 **return birthday**;  
}  
  
**void** Student::printStudent() {  
 cout << **"姓名："** << **name** << **"\t"** << **"生日："**;  
 **birthday**.printDate();  
 cout << endl;  
}  
  
**int** main() {  
 myDate \*date = **new** myDate(2020, 7, 4);  
  
 Student \*stu = **new** Student();  
 stu->setStudent(**"张三"**, \*date);  
 stu->printStudent();  
  
 **return** 0;  
}

## 类的示例程序剖析

### 程序结构

一个完整的c++程序包括以下几部分

1. 一个主函数，可以调用其他函数，但不能被调用，也称为主程序
2. 用户定义的任意多个类及全局函数
3. 全局说明，在所有函数和类定义之外的变量说明及函数原型
4. 注释
5. 头文件

对于比较大的程序，根据主函数和各用户定义的类以及全局函数的功能及相互关系，可以吧类及全局函数划分为几个文件，包括.cpp文件和.h文件。

.cpp文件是源程序文件

.h是文件的头文件

从逻辑关系上看，典型的C++程序的结构包括类的定义，类中成员函数的实现及主函数main

在由多个文件组成的C++项目中，如果多个.cpp文件都用到同一个类，可以将类定义写在一个头文件中，然后再各个.cpp文件中包含该头文件。

类的非内联成员函数的函数体只能出现在某一个.cpp文件中，不能放在头文件中被多个.cpp文件包含， 否则链接时会发生重复定义的错误。

类的内联成员函数的函数体最好写在头文件中，这样编译器在处理内联函数的调用语句时，就能在本文件包含的头文件中找到内联函数的代码，并将这些代码插入调用语句处，**内联成员函数放在头文件中被多个.cpp文件包含，不会导致重复定义的错误。**

### 成员变量与成员函数的定义

成员变量一般均定义为私有访问权限，这样的成员仅能在本类内访问，如果要从类外访问，则需要为成员变量定义响应的访问函数，然后通过调用访问函数来访问私有的成员变量。

实现成员函数时要指明类型的名称，在类体外定义的一般格式如下：

返回值类型 类名::成员函数名(参数列表) {  
 函数体  
}

成员函数并非每个对象各自存一份。成员函数和普通函数一样，在内存中只有一份，他可以作用于不同的对象，为类中各个对象共享。

通常，因为函数体代码较长，所以在类体内仅给出成员函数的原型，然后在类体外给出对应的函数体，**如果函数体定义在类体内，则系统将其视为内联函数**

类中定义的成员函数允许重载

### 创建类对象的基本形式

具有类类型的对象成为对象，完整的定义了类之后，就可以创建对象了，创建对象也称为声明，定义和生成对象。

定义了类数据类型只是告诉编译器系统该数据类型的结构或框架，并没有分配内存，类只是一个样板，根据这个样板，可以在内存中开辟出同样结构的示例，即对象，所以说，对象是类的实例。

#### 创建对象

定义一个普通对象，即类变量的基本方法有两种,

##### 方法一

类名 对象名;

或

类名 对象名(参数);

或

类名 对象名 = 类名(参数);

或

类名 对象名1, 对象名2, 对象名3;

或

类名 对象名1(参数1), 对象名2(参数2), 对象名3(参数3);

此处的对象名的命名规则和普通变量相同，必须是一个符合要求的标识符，创建对象时是否带参数，要依赖于类中构造函数的形式，创建对象后，C++会为他分配相应的空间，用来存储对象所有的成员变量，而类中定义的成员函数则被分配到存储空间中的一个公用区域，有该类的所有对象共享。

##### 方法二

类名 \*对象指针名 = **new** 类名;

或

类名 \*对象指针名 = **new** 类名();

或

类名 \*对象指针名 = **new** 类名(参数);

**用new创建对象时返回的是一个对象指针**，这个指针指向本类刚创建的这个对象，C++分配给指针的仅仅是存储指针值得空间，而对象所占用的空间分配在堆上，使用new创建的对象，必须使用delete来撤销。

使用“类名 \*对象指针名 = new 类名;”创建对象时，调用无参数的构造函数时，如果这个构造函数是由编译器为类提供的，则类中成员变量不进行初始化。

使用“类名 \*对象指针名 = new 类名();”创建对象时，也调用无参的构造函数，如果这个构造函数是有编译器为类提供的，则对类中的成员变量进行初始化。

#### 声明对象引用，指针，数组

与基本数据类型一样，还可以声明对象的引用，对象的指针，对象的数组。

声明对象引用

类名 &对象引用名 = 对象;

声明对象指针

类名 \*对象指针名 = 对象的地址;

声明对象数组

类名 对象数组名[数组大小];

同类型的对象之间可以相互赋值，对象和对象指针都可以作为函数参数，函数的返回值可以是对象或指向对象的指针

**int** main() {  
 Student s1, s2; *// 声明了Student的两个实例s1和s2* Student \*p = &s1; *// 定义了指向对象s1的Student类型指针p* Student &R = s2; *// 定义了Student的实例s1的引用R* Student students[50]; *// 声明了Student类型的数组,size为50* **return** 0;  
}

程序运行中，创建对象就是为对象分配内存，此时使用类作为模板，也就是按照类的定义中声明的成员变量为对象分配内存

## 访问对象的成员

### 使用对象访问成员变量与调用成员函数

定义了类和对象后，就可以访问的对象的成员，通过对象访问成员变量的一般格式如下

对象名.成员变量;

调用成员函数的格式如下：

对象名.成员函数(参数);

### 使用指针访问对象的成员

除了使用“对象.成员”的格式外，还可可以使用指针的方式来访问类成员，如果是通过指针访问成员变量，则“.”运算换成“->”运算，即使用“指针->成员名”的方法来访问对象的成员。

**int** main() {  
 Student ss;  
 **int** y, m, d;  
 string name;  
 Student \*sp = &ss;  
 cout << **"请输入学生的姓名和生日:"** << endl;  
 cin >> name >> y >> m >> d;

sp->setStudent(name, myDate(y, m, d));  
 sp->printStudent();

**return** 0;  
}

### 使用引用访问对象成员

**int** main() {  
 Student student = Student();  
 **int** y, m, d;  
 string name;  
 Student &sy = student;  
 cout << **"请输入姓名及生日"** << endl;  
 cin >> name >> y >> m >> d;  
 sy.setStudent(name, myDate(y, m, d));  
 sy.printStudent();  
  
 **return** 0;  
}

sy与student成为同一个对象的两个不同的名字，访问成员时仍然使用“.”操作符，及“引用名.成员”

## 类成员的可访问范围

数据封装与隐藏是面向对象编程的一个重要特点，他通过对类内成员的访问限制，可以有效的防止函数直接访问类的成员，从而实现数据封装与隐藏。

### 访问修饰符的含义

Public的含义是共有的，使用它修饰的类的成员可以在程序的任何地方被访问

Private的含义是私有的，使用它修饰的类的成员仅能在本类中访问。

Protected的含义是保护，他的作用介于于pubilc和private之间，使用它的修饰的类的成员能在本类以及子类中被访问

3种关键字，出现的次数和先后次序都没有限制，成员的课访问范围由他之前离他最近的那个访问范围说明符决定，**如果某个成员前面没有访问说明符，则对于类来说，该成员默认为私有成员。**

私有类型的成员在类外是不能访问的，通过公有函数访问的效率比直接访问的效率要低，为了权衡这方面的因素，C++提供了友元访问。只有在类内或在友元函数内才可以访问私有成员。

**class** Line {  
**private**:  
 **double length**;  
**public**:  
 **void** setLength();  
  
 **double** getLength();  
  
**private**:  
 **double area**;  
  
 **bool** isEmpty();  
};

### 成员的访问

### 隐藏的作用

设置私有成员的机制叫做“隐藏”，隐藏的一个目的就是强制对私有成员变量的访问一定通过公有成员函数进行，这样做的好处是：如果以后修改了成员变量的类型等属性，只需要更改成员函数即可，否则，所有直接访问成员变量的语句都需要修改。

隐藏的机制还可以避免对对象的不正确操作。有的成员函数，设计他们的初衷只是用来让本类中成员函数调用的，给这些函数起辅助作用，并不希望对外开放。因此就可以将他们声明为私有，隐藏起来，现代软件开发绝大多数是合作完成的。一个程序员设计一个类。很可能被许多程序员使用，在设计类的时候，应当尽可能的隐藏使用者无需知道的实现细节，只留下必要的接口(即一些成员函数)来对对象进行操作，这样能够避免类的使用者随意使用成员函数变量而导致错误。

## 标识符的作用域与可见性

在C++程序中随处可见标识符，可以说，标识符是组成程序的最小成分之一，类名，函数名，变量名，常量名，和枚举类型的取值都是标识符，这些标识符有各自的作用域和可见性。

标识符的作用域是指标识符的有效范围，即他在程序中存在的区域。标识符的可见性是指在程序的那个区域中使用，对于同一个标识符来说，这两个区域可能不完全重合的。

C++中标识符的作用域有以下几种

1. 函数原型作用域
2. 局部作用域(块作用域)
3. 类作用域
4. 命名空间作用域

### 函数原型作用域

在声明函数原型是形参的作用范围几乎是函数原型作用域，这是C++程序中最小的作用域。

**double** ares(**double** radius);

标识符radius的作用就在area形参列表的左右括号之间，在程序的其他地方不能引用这个标识符，因为函数声明中形参仅在形参列表中有效，所以，函数声明中往往不写形参名，而是仅写形参的类型

### 局部作用域

程序中使用相匹配的一对大括号括起来的程序成为块，作用域局限在块内称为局部作用域，例如，在函数定义中，形参列表中形参的作用域，从形参列表中的声明处开始到整个函数体结束之处为止，函数体内声明的变量，其作用域从声明处开始，一直到声明所在的块结束，这些都是局部作用域，具有局部作用域的变量成为局部变量。

### 类作用域

类可以被看做是一组有名字的成员的集合，类X的成员m具有类作用域，对m的访问方式有如下3种

1. 如果在类X的成员函数中没有生命同名的局部作用域标识符，那么在该函数内可以直接访问成员m,也就是说，m在这样的函数中起作用。
2. 在类外，可以通过表达式x.m或者X::m来访问， 其中x是X的实例，这正是程序中访问对象成员的基本方法，当然，这样的访问不能违反m的访问修饰符的限定。
3. 在类外，可以通过prt->m这样的表达式来访问，其中ptr为指向类X的一个实例的指针，当然，这样的访问不能违反m的访问修饰符的限定。

### 命名空间作用域

一个大型的程序通常由不同模块构成，不通过的模块甚至可能是由不同人员开发的，不同模块之间的类和函数可能发生重名，这样就会引发错误，命名空间是为了消除同名引用的歧义。

定义一个命名空间的一般形式如下

**namespace** 命名空间名  
{  
 命名空间内的各种声明(函数声明, 类声明)  
}

一个命名空间确定了一个命名空间的作用域，凡是在该命名空间之内声明的，不属于当前所述各个作用域标识符，都属于该命名空间作用域，在命名空间内部可以直接引用当前命名空间中声明的标识符，如果需要引用其他命名空间的标识符，需要使用下面的方式：

名称空间名:标识符名

## 类和对象的进阶

### 构造函数

#### 构造函数的作用

为了对 对象进行初始化， C++提供了一种称为构造函数的机制， 用于对 对象进行初始化， 实际上是用来为成员变量赋值的

构造函数是类中特殊的成员函数， 它属于类的一部分， 给出类的定义时， 由程序员编写构造函数，如果程序员没有编写类的任何构造函数，则由系统自动添加一个不带参数的构造函数。

声明对象后， 可以使用new运算符为对象进行初始化， 此时调用的是对象所属类的构造函数。

构造函数的作用是完成对象初始化的工作，用来保证对象的初始状态是确定的，在对象生成时， 系统自动调用构造函数，用户在程序中不会直接调用构造函数。

#### 构造函数的定义

定义一个类时， 需要为类定义相应的构造函数，**构造函数的函数名与类名相同**，**没有返回值，**一个构造函数可以有多个，即构造函数允许函数重载，**同一个类的多个构造函数的参数一定不能完全相同**。

形式一

类名::类名(形参1, 形参2, ....):x1(形参1), x2(形参2), .....{}

冒号后面的内容称为构造函数初始化列表，构造函数的初始化列表中，是成员变量列表， 每个成员变量后面跟一个放在圆括号中的初始化表达式，最简单的初始化表达式就是形参的值。

形式二

类名::类名(形参1, 形参2, ....) {  
 x1 = 形参1;  
 x2 = 形参2;  
 .....  
}

构造函数中语句通常是为类的成员变量赋初值，如形式2

形式三

类名::类名() {  
 x1 = 初始化表达式1;  
 x2 = 初始化表达式2;  
 .....  
}

如果函数是无参构造，则通常使用初始表达式为成员变量赋初值

#### 构造函数示例

#include **<iostream>  
  
using namespace** std;  
  
**class** myDate {  
**public**:  
 myDate();  
  
 myDate(**int**);  
  
 myDate(**int**, **int**);  
  
 myDate(**int**, **int**, **int**);  
  
 **int year**;  
 **int month**;  
 **int day**;  
  
 **void** getDate() **const** {  
 cout << **year** << **"/"** << **month** << **"/"** << **day** << endl;  
 }  
};  
  
*// 不带参数*myDate::myDate() : **year**(1970), **month**(1), **day**(1) {}  
  
*// 带一个参数*myDate::myDate(**int** y) : **month**(2), **day**(2) {  
 **year** = y;  
}  
  
*// 带两个参数*myDate::myDate(**int** y, **int** m) : **day**(3) {  
 **year** = y;  
 **month** = m;  
}  
  
*// 带三个参数*myDate::myDate(**int** y, **int** m, **int** d) {  
 **year** = y;  
 **month** = m;  
 **day** = d;  
}  
  
  
**int** main() {  
 myDate stu = myDate(2020, 10, 4);  
 stu.getDate();  
}

#### 构造函数的使用

当程序创建一个对象时，系统自动调用构造函数来初始化该对象，构造函数都是自动调用的，不由程序员在程序中显式的调用，也就不是由程序员控制的，对象生成后，对这个对象就不会在执行其他的构造函数了，C++语言规定，创建类的任何对象都一定会调用构造函数进行初始化，对象需要占据空间，生成对象时，为对象分配的这段内存空间的初始化由构造函数完成。

生成对象时，系统要选择调用的构造函数，根据创建对象的语句所提供的参数，系统就可以决定该调用那个构造函数，如果没有提供参数，则要调用无参的构造函数，如果调用语句中提供了实参与构造函数参数列表中形参不匹配，就出现编译错误，这意味着，类中定义了那种构造函数，就决定了创建对象时可以使用那种形式。

C++编译器总是要建立一个不带参数的构造函数，但是一旦在类中程序员自己定义了构造函数（任意一个），那么系统就不在提供默认构造函数，此时如果声明一个不带参数的对象就会报错。

#### 使用构造函数创建对象指针

**int** main() {  
 myDate \*stu = **new** myDate(2020, 9);  
 stu->getDate();  
}

使用new创建对象时，加括号不加括都是允许的，但系统处理时会有差异：

对于有自己编写的构造函数的类，不论有没有括号，都调用构造函数进行初始化。

如果类中没有定义自己的构造函数，即只有默认构造函数，则不加括号，系统只为成员变量分配内存空间，但是不进行内存的初始化，成员变量的值是随机值，而加了括号，系统在为成员变量分配内存的同时，将其初始化为0

**特别的，如果程序中声明了对象数组，即数组中每个元素都是一个对象，则一定要为对象所属的类定义一个无参的构造函数，因为数组中每个元素都需要调用无参的构造函数进行初始化，所以必须要有一个无参的构造函数**

#### 复制构造函数

复制构造函数是构造函数的一种，也称为拷贝构造函数，他的作用是使用一个已存在的对象去初始化另一个正在创建的对象，例如类对象间的赋值是由复制构造函数实现的。

复制构造函数只有一个参数，参数类型是本类的引用，复制构造函数的参数可以是const引用，也可以是非const引用。

一个类中可以写两个复制构造函数，一个函数的参数是const引用，另一个函数的参数非const引用，这样，当调用复制构造函数时，既能以常量对象（初始化后值不能改变的对象）作为参数，也能以非常量作为参数去初始化其他对象。

Student(**const** Student &s);

复制构造函数有些特别，这个参数是引用类自己的对象，即用一个已有的对象来建立新对象，为了不改变原有对象，通常使用const来限定，当然，也可能省略const。

如果类中没有给出复制构造函数，那么编译器会自动生成一个复制构造函数，在大多数情况下，其作用是实现从源对象到目标对象逐个字节的复制，即使得目标对象的每个成员变量都变得和源对象相等。

编译器自动生成的复制构造函数称为默认复制构造函数。

默认构造函数不一定存在，但是复制构造函数总是存在，如果程序中自定义了复制构造函数，则编译器只调用自定义的复制构造函数

Student::Student(**const** Student &s) {  
 **name** = **"COPY"** + s.**name**;  
 **birthday** = s.**birthday**;  
}  
  
**int** main() {  
 Student stu;  
 stu.printStudent();  
 Student a = Student(stu);  
 a.printStudent();  
*// init  
// 姓名: Noname 生日: 1970/1/1  
// 姓名: COPYNoname 生日: 1970/1/1*}

复制构造函数的任务是使用一个已有对象生成另一个对象，C++语法并没有限制构造函数的功能，所以复制构造函数不一定要做复制的工作，例如上面的例子给出的复制构造函数添加 了一些信息，当然如有必要也可以添加其他必要的操作。

自动调用复制构造函数的情况有3种

1. 当用一个对象去初始化本类的另一个对象时，会调用复制构造函数
2. 如果函数F的参数是类A的对象，那么当调用F时,会调用类A的复制构造函数。换句话说，**作为形参的对象，是用复制构造函数初始化的**，而且调用复制构造函数时的参数，就是调用函数是所给的实参。
3. 如果函数的返回值是类A的对象，那么当函数返回时，会调用类A的复制构造函数，也就是说，**作为函数返回值的对象使用复制构造函数初始化的**，而调用复制构造函数是的实参，就是return语句所返回的对象。

#### 类型转换构造函数

如果构造函数只有一个参数，则可以看作是类型转换的构造函数，他的作用是进行类型的自动转换。

**class** Demo {  
 **int id**;  
**public**:  
 Demo(**int** i) {  
 **id** = i;  
 cout << **"id="** << **id** << **" 构造函数"** << endl;  
 }  
  
 **void** printDemo();  
  
 ~Demo() {  
 cout << **"id="** << **id** << **" 析构函数"** << endl;  
 }  
};  
  
**void** Demo::printDemo() {  
 cout << **"id="** << **id** << **" printDemo"** << endl;  
}  
  
  
**int** main() {  
 Demo d4(4);  
 d4.printDemo();  
 d4 = 6;  
 *// 表面上看是一条赋值语句, 实际上他将6创建为一个临时的Demo对象, 然后使用这个对象对d4进行赋值* d4.printDemo();  
}

### 析构函数

与构造函数一样， 析构函数也是成员函数的一种，他的名字也与类名相同， 但是要在类名前加一个“~”，以区别去构造函数。

析构函数没有参数， 也没有返回值，一个类中有且仅有一个析构函数，如果程序中没有定义析构函数, 则编译器自动生成偶默认的析构函数， 析构函数不可以多于一个， 不会有重载的析构函数， 默认析构函数的函数体为空。

创建对象时自动调用构造函数， 那么什么时候调用析构函数呢？

可想而知，在对象消亡时自动调用析构函数，析构函数的作用是做一些善后的工作，例如,如果在创建对象时使用new运算符动态分配了内存空间， 则在析构函数中应该使用delete释放掉这一部分占用的空间， 保证空间可再利用。

**int** main() {  
 myDate \*date = **new** myDate();  
 **delete** date;  
}

对象数组与delete语句

**int** main() {  
 myDate \*date = **new** myDate[2];  
 **delete**[]date;  
}

为两个对象各调用一次析构函数

### 类的静态成员

#### 静态变量

与C语言一样， 使用static说明自动变量， 根据定义的位置不同， 分为静态全局变量和静态局部变量。

全局变量是指所有在花括号之外声明的变量， 其作用域是全局可见，即在整个项目文件内都有效， **使用static修饰**的全局变量是静态全局变量，其作用域是有所限制的，**仅在定义该变量的源文件有效**，项目中的其他源文件不能使用它。

局部变量是指块内定义的变量， 从定义之处开始到本块结束处为止是局部变量的作用域， 使用static修饰的局部变量是静态局部变量，即定义在块内的静态变量，静态局部变量具有局部作用域， 但却具有全局生命周期， 也就是说，**静态局部变量在程序的整个运行期间都存在，它占据的空间一直到程序结束才释放（多次调用函数不会重新创建）**， 但仅在定义它的块中有效， 在块外并不能访问他。

静态变量均存储在全局数据区，静态局部变量只执行一次初始化， 如果程序未显示给出初始值， 则相当于初始化为0，如果显示的给出值，则在**该静态变量所在块第一次执行时完成初始化**。

#### 类的静态成员

# 文件操作

C++标准库中有3个流类可以用于文件操作，这3个类称为文件流类

1. Ifstream：用于从文件中读取数据
2. Ofstream：用于向文件中写入数据
3. Fstream：既可以用于从文件中读取数据，又可以向文件中写入数据

使用这3个流类，程序中需要包含fstream头文件

在程序中，要使用一个文件，必须包含3个基本步骤

1. 打开文件（open）
2. 操作文件
3. 关闭文件（close）

## 打开和关闭文件

在对文件进行读写操作之前，先要打开文件，打开文件有以下两个目的：

1. 建立关联，通过指定文件名，建立起文件和文件流对象的关联，以后在对文件进行操作时，就可以通过与之关联的流对象进行操作
2. 指明文件的使用方式和文件格式，使用方式有
   1. 只读
   2. 只写
   3. 既读又写
   4. 在文件末尾追加

文件格式有

1. 文本
2. 二进制

# 内置库（头文件）

1. cmath