

Std::cout 是标准的写法

```
int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

如果没有用 std::cout, 需要在前面用 using xxxxx std

来防止不同头文件中的函数使用错误

```
using namespace std;
cout << "welcome";
cout << endl;
```

或者在前面用这种方式来声明, 这种方式是最稳定的一种方式, 会保证不会出错。

Cin 是输入 cin>>

```
int main()
{
    using std::cout;
    using std::endl;
    using std::cin;
    cout << "Hello World!\n";
    cout << endl;
    cout << "nihao";
    return 0;
}
```

换行有两种

1. cout<<endl cout 输出默认 10 进制

2. " \n"

输出具体的数

```
cout << "i have " << book << " book";
```

数学运算的头文件

```
#include <iostream>
#include <cmath>
```

构造函数

Type functionname ()

```
{
}
```

两种方法

1. 在函数构建内部加入 using，只针对这个函数
2. 在函数声明前面加上 using，会让在声明后的所有函数都可以用

```
11 void pri()
12 {
13     using namespace std;
14     cout << "hello";
15 }
```

```
1 #include <iostream>
2 #include <cmath>
3 using namespace std;
4 void pri();
5 int main()
6 {
7     //using std::cout;
8     //using std::endl;
9     //using std::cin;
10    pri();
11 }
12 void pri()
13 {
14     cout << "hello";
15 }
16 }
```

Sizeof 的调用要加上#include <climits>头文件

类型转换，低精度向高精度转换不会变化本身的值，而高精度向低精度转换其结果是不确定的

数组的空数组要写' \0'，数组中的内容要写成单引号

计算数组长度的要包含的头文件

用 strlen()

```
#include<cstring>
```

两个 Cin 是用空格作为分割，若你要用回车来区分两个输入就不能用 cin

可以用 cin.get(数组名,数组容量)和 cin.getline(),这两个都是针对数组。Cin 后面可以用 cin.get()来吃掉回车符号，让后再次输入。

```
cout << "你的生日? \n";
int year;
cin>> year;
cin.get();
cout << "你的地址";
char address[80];
cin.get(address, 80);
cout << "date"<<endl;
cout << "地址: "<<address;
```

String 类

String 相比于数组而言更见简便，可以自动处理 string 的大小。

数组直接的数不可以赋值给另一个，而 string 可以。

String 可以直接用 + 来进行相加

```
{
    string da = "sadasf";
    string ba = "dadaf";
    string k = da + ba;
    cout << k;
}
```

删除警告可以用 pragma warning ()

```
1  ~#include <iostream>
2  #include <cmath>
3  #include<climits>
4  #include<cstring>
5  #include<string>
6  #pragma warning(disable:4996)
7  using namespace std;
8  double cov(double x);
9  ~int main()
10 {
11     char charr1[20]="name";
12     char charr2[20]="duttty";
13     string da = "cod";
14     string ba = "duty";
15     string k = da + ba;
16     //strcpy_s(charr1, "dadd");
17     strcat(charr1, charr2);
18     cout << charr1<<endl;
19     cout << charr2;
20 }
21 ~double cov(double x)
22 {
23     double ma;
24     ma = x * 200;
25     cout << "num=" << ma << endl;
26     return ma;
27 }
```

Strcat_s() 添加函数，把 2 加到 1

Strcpy_s() 拷贝函数，把 2 拷贝到 1

```
strcat_s(charr1, charr2);
strcpy_s(charr1, charr2);
```

结构体的构造，结构体可以赋值

```

struct inf
{
    string name;
    float volume;
    double price;
};

```

结构体名字. 项目 其对应的类型便是其项目的类型

```

};
cout << fruit.name<<endl;
cout << fruit.price;

```

结构数组 struct

```

22     inf gifts[30] =
23     {
24         {"call",21,3},
25         {"of",12,3.22}
26     };
27     cout << fruit.name<<endl;
28     cout << fruit.price<<endl;
29     cout << gifts[1].name;

```

共用体 union

共用体是对 int 或 double 或 long，这三个里面的一种进行存储，结构体可以同时存储上面三种。

```

union pail
{
    int year;
    char name[20];
};

```

在调用的时候只能这样，不能像结构体一样直接大括号里面调用。

```

pail pail1;
pail1.year = 19;
cout << pail1.year << endl;

```

枚举 enum

```
enum spectrum {red,orange,yellow,blue,violet,indigo,ultraviolet};
spectrum band;
int color;
band = blue;
color = band +2;
cout << color<<endl;
```

枚举其中对应的是 int 整形，各类颜色代表 0，1，2，这中整形。

可以定义一个整型在等号左边进行加减,但是不能定义枚举在左边进行加减。

枚举可以进行自己定义里面的数值

```
enum bits {one=1,two=2,four=4,eight=8};
bits myflag;
myflag = bits(four);
```

指针

```
1 int a, b, * p1 = &a, * p2;
2
3 &*p1 的含义:
4 &和*都是自右向左运算符,因此先看p1是指针,*p1即为p1指向的对象,
5 因此*p1等价于a,因此a 的前面加一个&,表示的就是 a 的地址
6 因此: &*p1 ,p1 ,&a 三者等价
7 *&a 的含义: a的地址前加*表示的就是a本身,指针就是用来放地址的,地址前面加*表示的就是这个对象
8 因此: *&a ,a ,*p 三者等价
```

< 和 > 的优先级比 << 要低所以要用括号括起来,来正确运行

```
int x=100;
cout << (x < 1) << endl;
```

Cout.setf(ios_base::boolalpha)可以将 bool 值 0, 1 变成 ture 和 false

```
cout.setf(ios_base::boolalpha);
cout << (x < 1) << endl;
```

For 循环阶乘计算

```
int main()
{
    int i, k;
    long long fac[num], res = 1, f_res;
    fac[0] = 1;
    for (i = 1; i < 10; i++)
    {
        fac[i] = i;
        for (k = 0; k < i; k++)
        {
            res = fac[k] * res;
            f_res=res;
        }
        res = 1;
        cout << i-1<< "!=" << f_res << endl;
    }
}
```

通过 strcmp（名称，比较字符）来进行比较运算，当第一个与第二个相等则停止

```
int main()
{
    char word[5]="date";
    for (char ch='a'; strcmp(word, "mate"); ch++)
    {
        cout << word << endl;
        word[0] = ch;
    }
    cout << "finish!" << endl;
}
```

或者用 string 来进行比较

```
int main()
{
    string word = "date";
    for (char ch = 'a'; word != "mate"; ch++)
    {
        cout << word << endl;
        word[0] = ch;
    }
    cout << "finish!" << endl;
}
```

```

1  #include <iostream>
2  #include <cmath>
3  #include<climits>
4  #include<cstring>
5  #include<string>
6  using namespace std;
7  double cov(double x);
8  const int num = 10;
9  struct inf
10 {
11     string name;
12     float volume;
13     double price;
14 };
15 union pail
16 {
17     int year;
18     char name[20];
19 };
20 int main()
21 {
22     pail pail1;
23     pail1.year = 19;
24     cout << pail1.year << endl;
25     string word = "date";
26     for (char ch = 'a'; word != "mate"; ch++)
27     {
28         cout << word << endl;
29         word[0] = ch;
30     }
31     cout << "finish!" << endl;
32 }
33 double cov(double x)
34 {
35     double ma;
36     ma = x * 200;
37     cout << "num=" << ma << endl;
38     return ma;
39 }

```


类型别名

1. #define 新名字 int/char/double
2. typedef char byte

延时程序

```
#include <ctime>
using namespace std;
double cov(double x);
int main()
{
    cout << "enter your delay time" << endl;
    float secs;
    cin >> secs;
    clock_t delay = secs * CLOCKS_PER_SEC;
    cout << "starting\a\n";
    clock_t start = clock();
    while (clock() - start < delay)
        ;
    cout << "done \a\n";
    return 0;
}
```

For 循环遍历数组

```
int main()
{
    int price[5] = { 1,2,3,4,5 };
    for (int x : price)
    {
        cout << x << endl;
    }
}
```

检测数组内的字符个数

```
while (ch != '@')
{
    if (isalpha(ch))
        chars++;
    else if (isspace(ch))
        ws++;
    else if (ispunct(ch))
        punct++;
    else if (isdigit(ch))
        dg++;
    else
        others++;
    cin.get(ch);
}
```

? : 运算符 A? B: C

如果 A 条件成立，则 A 的结果是 B

若 A 的条件不成立, A 的结果是 B

Switch 语句

Switch 语句用于多个选择，switch 可以和枚举一起使用。

Switch 只能适用于具体数，不能适用于范围值

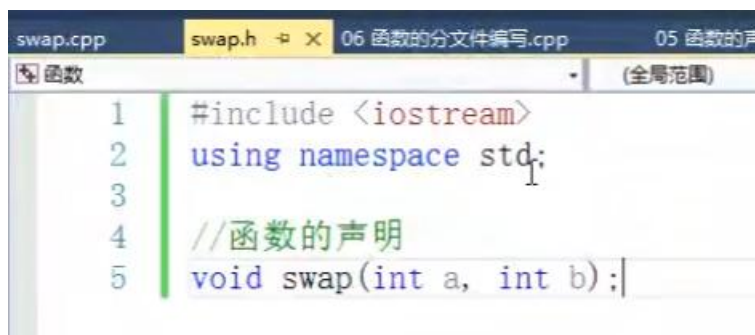
数组的大小 `nums.size();`

h3 6.7 函数的分文件编写

作用：让代码结构更加清晰

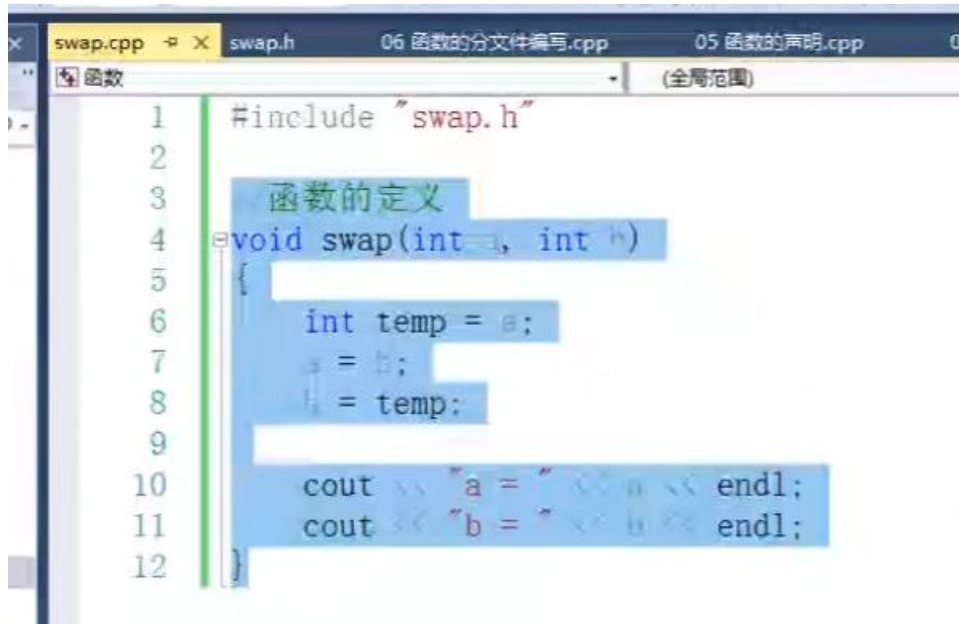
函数分文件编写一般有4个步骤

1. 创建后缀名为.h的头文件
2. 创建后缀名为.cpp的源文件
3. 在头文件中写函数的声明
4. 在源文件中写函数的定义



The screenshot shows a code editor with the file 'swap.h' selected. The code contains the following lines:

```
1 #include <iostream>
2 using namespace std;
3
4 //函数的声明
5 void swap(int a, int b);
```



The screenshot shows a code editor with the file 'swap.cpp' selected. The code contains the following lines:

```
1 #include "swap.h"
2
3 //函数的定义
4 void swap(int a, int b)
5 {
6     int temp = a;
7     a = b;
8     b = temp;
9
10    cout << "a = " << a << endl;
11    cout << "b = " << b << endl;
12 }
```

```
const int * p = &a;
```

常量指针

特点：指针的指向可以修改，但是指针指向的值不可以改

*p = 20; 错误，指针指向的值不可以改

p = &b; 正确，指针指向可以改

```
int * const p = &a;
```

指针常量



特点：指针的指向不可以改，指针指向的值可以改

*p = 20; 正确，指向的值可以改

p = &b; 错误，指针指向不可以改

```
const int * const p3 = &a;
```

//指针的指向 和 指针指向的值 都不可以改

*p3 = 100; // 错误

p3 = &b; // 错误

```
int a[5] = { 1,2,3,4,5 };
```

```
int* p = a;
```

```
cout << "地址为:" << p << endl;
```

```
cout << "内容为:" << *p << endl;
```

```
p++;
```

```
cout << "第二个为: " << *p << endl;
```

```
return 0;
```

```

int main()
{
    int arr[10] = { 4,3,6,9,1,2,10,8,7,5 };
    int len = sizeof(arr)/sizeof(arr[0]);
    bubblesort(arr, len);
    for (int i = 0; i < len - 1; i++)
    {
        cout << arr[i] << endl;
    }
}

void bubblesort(int* arr, int len)
{
    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;
            }
        }
    }
}

```

//2、通过指针指向结构体变量

```
student * p = &s;
```

//3、通过指针访问结构体变量中的数据

```
cout << "姓名: " << p->name << " 年龄: " << p->age << "
```

//将函数中的形参改为指针，可以减少内存空间，而且不会复制新的副本出来

```

void printStudents(const student *s)
{
    //s->age = 150; //加入const之后，一旦有修改的操作就会报错，可以防止我们的误操作
    cout << "姓名: " << s->name << " 年龄: " << s->age << " 得分: " << s->score
}

```

函数的默认参数

```
21
22 //2、如果函数声明有默认参数，函数实现就不能有默认参数
23 // 声明和实现只能有一个有默认参数
24 int func2(int a = 10, int b = 10);
25
26 int func2(int a = 20, int b = 20)
27 {
28
```

函数的默认值如果有传入则就是传入值，没有值则就是默认值。

函数重载

```
//函数重载的满足条件
//1、同一个作用域下
//2、函数名称相同
//3、函数参数类型不同，或者个数不同，或者顺序不同
void func()
{
    cout << "func 的调用" << endl;
}

void func(int a)
{

```

引用也可以重载

```
1 //函数重载注意事项
2 //1、引用作为重载条件
3
4 void func(int &a)
5 {
6     cout << "func (int &a) 调用 " << endl;
7 }
8
9 void func(const int &a)
10 {
11     cout << "func (const int &a) 调用 " << endl;
12 }
```


内存分区模型

C++程序在执行时，将内存大方向划分为**4个区域**

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 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，程序结束时由操作系统回收

```
int * func()  
{  
    //在堆区创建整型数据  
    //new返回是 该数据类型的指针  
    int * p = new int(10);  
}
```

```
5         //释放数组 delete 后加 []  
6         delete[] arr;  
7
```

引用的基本使用

****作用：** **给变量起别名

语法： 数据类型 &别名 = 原名

示例：

```
C++  
1  int main() {  
2  
3      int a = 10;  
4      int &b = a;
```

当别名改变时候，其原名的数值也会改变

引用注意事项

- 引用必须初始化
- 引用在初始化后，不可以改变

通过引用参数产生的效果同按地址传递是一样的。引用的语法更清楚简单

封装

封装的意义

封装是C++面向对象三大特性之一

封装的意义：

- 将属性和行为作为一个整体，表现生活中的事物
- 将属性和行为加以权限控制

语法： `class 类名{ 访问权限： 属性 / 行为 };`

4.1.2 struct和class区别

在C++中 struct和class唯一的区别就在于 默认访问权限不同

区别：

- struct 默认权限为公共
- class 默认权限为私有


```
//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;
}
```

构造函数语法: `类名(){}`

1. 构造函数, 没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数, 因此可以发生重载
4. 程序在调用对象时候会自动调用构造, 无须手动调用,而且只会调用一次

析构函数语法: `~类名(){}`

1. 析构函数, 没有返回值也不写void
2. 函数名称与类名相同,在名称前加上符号 ~
3. 析构函数不可以有参数, 因此不可以发生重载
4. 程序在对象销毁前会自动调用析构, 无须手动调用,而且只会调用一次

//1、括号法

```
//Person p1; //默认构造函数调用
//Person p2(10); //有参构造函数
//Person p3(p2); //拷贝构造函数
```

//注意事项

```
//调用默认构造函数时候, 不要加()
//因为下面这行代码, 编译器会认为是一个函数的声明
Person p1();
```

//注意事项2

```
//不要利用拷贝构造函数 初始化匿名对象 编译器会认为 Person (p3) === Person p3; 对象
//Person(p3);
```

//3、隐式转换法

```
Person p4 = 10; // 相当于 写了 Person p4 = Person(10); 有参构造
Person p5 = p4; // 拷贝构造
```

默认情况下，c++编译器至少给一个类添加3个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

构造函数的分类及调用

两种分类方式：

按参数分为： 有参构造和无参构造

按类型分为： 普通构造和拷贝构造

三种调用方式：

括号法

显示法

隐式转换法

拷贝构造函数调用时机

C++中拷贝构造函数调用时机通常有三种情况

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递的方式给函数参数传值
- 以值方式返回局部对象

构造函数调用规则

默认情况下，c++编译器至少给一个类添加3个函数

1. 默认构造函数(无参，函数体为空)
2. 默认析构函数(无参，函数体为空)
3. 默认拷贝构造函数，对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

深拷贝与浅拷贝

深浅拷贝是面试经典问题，也是常见的一个坑

浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作

初始化列表

作用：

C++提供了初始化列表语法，用来初始化属性

语法： `构造函数(): 属性1(值1), 属性2 (值2) ... {}`

```

class Person {
public:

    ////传统方式初始化
    //Person(int a, int b, int c) {
    //    m_A = a;
    //    m_B = b;
    //    m_C = c;
    //}

    //初始化列表方式初始化
    Person(int a, int b, int c) :m_A(a), m_B(b), m_C(c) {}
    void PrintPerson() {
        cout << "mA:" << m_A << endl;
        cout << "mB:" << m_B << endl;
        cout << "mC:" << m_C << endl;
    }
}

```

静态成员

静态成员就是在成员变量和成员函数前加上关键字static，称为静态成员

静态成员分为：

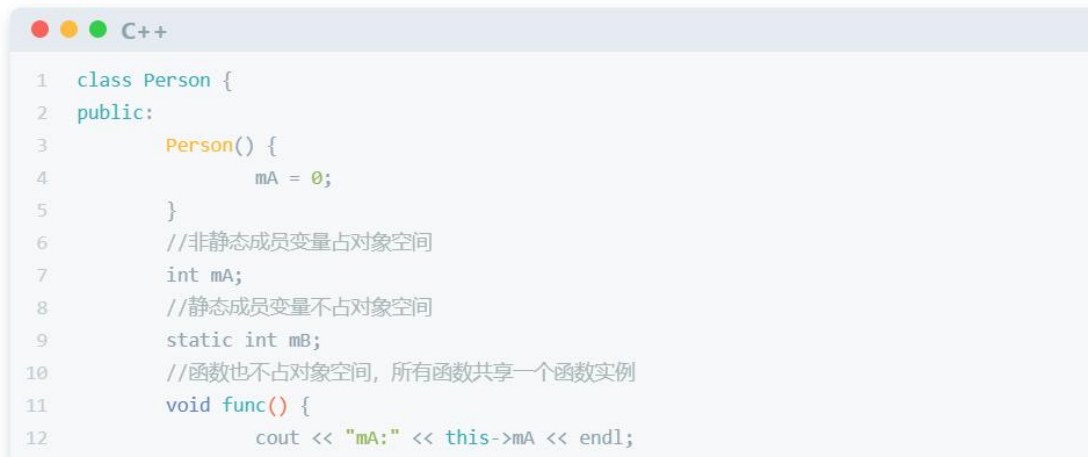
- 静态成员变量
 - 所有对象共享同一份数据
 - 在编译阶段分配内存
 - 类内声明，类外初始化
- 静态成员函数
 - 所有对象共享同一个函数
 - 静态成员函数只能访问静态成员变量

C++对象模型和this指针

成员变量和成员函数分开存储

在C++中，类内的成员变量和成员函数分开存储

只有非静态成员变量才属于类的对象上



```
1 class Person {
2 public:
3     Person() {
4         mA = 0;
5     }
6     //非静态成员变量占对象空间
7     int mA;
8     //静态成员变量不占对象空间
9     static int mB;
10    //函数也不占对象空间，所有函数共享一个函数实例
11    void func() {
12        cout << "mA:" << this->mA << endl;
```

this指针概念

通过4.3.1我们知道在C++中成员变量和成员函数是分开存储的

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码

那么问题是：这一块代码是如何区分那个对象调用自己的呢？

c++通过提供特殊的对象指针，this指针，解决上述问题。**this指针指向被调用的成员函数所属的对象**

this指针是隐含每一个非静态成员函数内的一种指针

this指针不需要定义，直接使用即可

this指针的用途：

- 当形参和成员变量同名时，可用this指针来区分
- 在类的非静态成员函数中返回对象本身，可使用return *this

空指针访问成员函数

C++中空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为**常函数**
- 常函数内不可以修改成员属性
- 成员属性声明时加关键字mutable后，在常函数中依然可以修改

友元

生活中你的家有客厅(Public)，有你的卧室(Private)

客厅所有来的客人都可以进去，但是你的卧室是私有的，也就是说只有你能进去

但是呢，你也可以允许你的好闺蜜好基友进去。

在程序里，有些私有属性 也想让类外特殊的一些函数或者类进行访问，就需要用到友元的技术

友元的目的就是让一个函数或者类 访问另一个类中私有成员

友元的关键字为 friend

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

有类做友元，全局函数做友元和成员函数做友元

运算符重载

运算符重载概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

关系运算符重载

****作用：** ****重载关系运算符**，可以让两个自定义类型对象进行对比操作

函数调用运算符重载

- 函数调用运算符 () 也可以重载
- 由于重载后使用的方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

继承

继承是面向对象三大特性之一

我们发现，定义这些类，下级别的成员除了拥有上一级的共性，还有自己的特性。

这个时候我们就可以考虑利用继承的技术，减少重复代码

继承的基本语法

例如我们看到很多网站中，都有公共的头部，公共的底部，甚至公共的左侧列表，只有中心内容不同

接下来我们分别利用普通写法和继承的写法来实现网页中的内容，看一下继承存在的意义以及好处

总结：

继承的好处：可以减少重复的代码

```
class A : public B;
```

A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中的成员，包含两大部分：

一类是从基类继承过来的，一类是自己增加的成员。

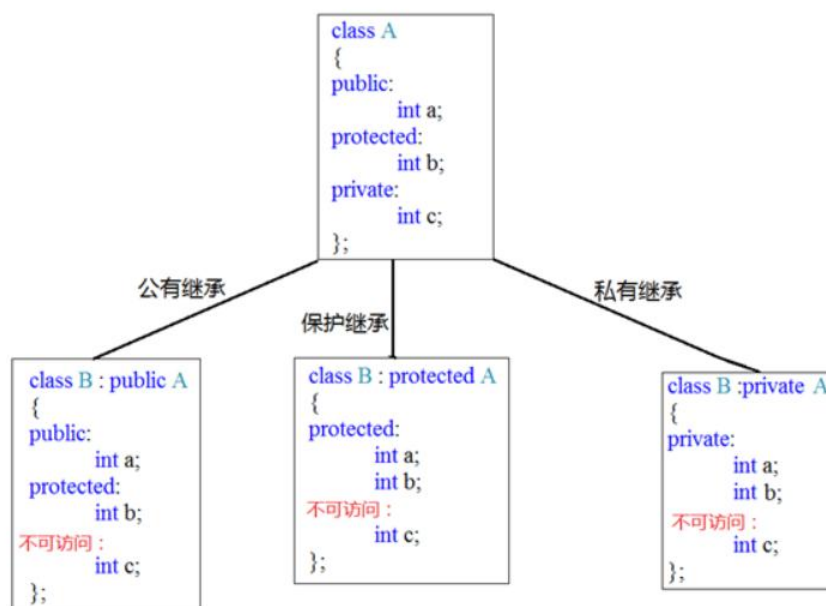
从基类继承过来的表现其共性，而新增的成员体现了其个性。

继承方式

继承的语法: `class 子类 : 继承方式 父类`

继承方式一共有三种:

- 公共继承
- 保护继承
- 私有继承



继承中构造和析构顺序

子类继承父类后, 当创建子类对象, 也会调用父类的构造函数

问题: 父类和子类的构造和析构顺序是谁先谁后?

继承同名成员处理方式

问题: 当子类与父类出现同名的成员, 如何通过子类对象, 访问到子类或父类中同名的数据呢?

- 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

总结：

1. 子类对象可以直接访问到子类中同名成员
2. 子类对象加作用域可以访问到父类同名成员
3. 当子类与父类拥有同名的成员函数，子类会隐藏父类中同名成员函数，加作用域可以访问到父类中同名函数

继承同名静态成员处理方式

问题：继承中同名的静态成员在子类对象上如何进行访问？

静态成员和非静态成员出现同名，处理方式一致

- 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

多继承语法

C++允许一个类继承多个类

语法： `class 子类：继承方式 父类1，继承方式 父类2...`

多继承可能会引发父类中有同名成员出现，需要加作用域区分

菱形继承

菱形继承概念：

两个派生类继承同一个基类

又有某个类同时继承者两个派生类

这种继承被称为菱形继承，或者钻石继承

多态

多态的基本概念

多态是C++面向对象三大特性之一

多态分为两类

- 静态多态: 函数重载 和 运算符重载属于静态多态, 复用函数名
- 动态多态: 派生类和虚函数实现运行时多态

静态多态和动态多态区别:

- 静态多态的函数地址早绑定 - 编译阶段确定函数地址
- 动态多态的函数地址晚绑定 - 运行阶段确定函数地址

总结:

多态满足条件

- 有继承关系
- 子类重写父类中的虚函数

多态使用条件

- 父类指针或引用指向子类对象

重写: 函数返回值类型 函数名 参数列表 完全一致称为重写

多态案例一-计算器类

案例描述:

分别利用普通写法和多态技术, 设计实现两个操作数进行运算的计算器类

多态的优点:

- 代码组织结构清晰
- 可读性强
- 利于前期和后期的扩展以及维护

纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类重写的内容

因此可以将虚函数改为**纯虚函数**

纯虚函数语法：`virtual 返回值类型 函数名 (参数列表) = 0 ;`

当类中有了纯虚函数，这个类也称为抽象类

抽象类特点：

- 无法实例化对象
- 子类必须重写抽象类中的纯虚函数，否则也属于抽象类

虚析构和纯虚析构

多态使用时，如果子类中有属性开辟到堆区，那么父类指针在释放时无法调用到子类的析构代码

解决方式：将父类中的析构函数改为**虚析构**或者**纯虚析构**

虚析构和纯虚析构共性：

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别：

- 如果是纯虚析构，该类属于抽象类，无法实例化对象

虚析构语法：

```
virtual ~类名(){};
```

纯虚析构语法：

```
virtual ~类名() = 0;
```

```
类名::~~类名(){};
```

总结:

1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据, 可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

```
int myAdd01(int a, int b)
{
    return a + b;
}

//函数模板
template<class T>
T myAdd02(T a, T b)
{
    return a + b;
}
```

```
cout << myAdd01(a, c) << endl; //正确, 将char类型的'c'隐式转换为int类型

//myAdd02(a, c); // 报错, 使用自动类型推导时, 不会发生隐式类型转换

myAdd02<int>(a, c); //正确, 如果用显示指定类型, 可以发生隐式类型转换
```

1.2.5 普通函数与函数模板的调用规则

调用规则如下:

1. 如果函数模板和普通函数都可以实现, 优先调用普通函数
2. 可以通过空模板参数列表来强制调用函数模板
3. 函数模板也可以发生重载
4. 如果函数模板可以产生更好的匹配, 优先调用函数模板

//2、参数模板化

```
template<class T1, class T2> <T>
void printPerson2(Person<T1, T2>&p)
{
    p.showPerson();
    cout << "T1 的类型为: " << typeid(T1).name() << endl;
    cout << "T2 的类型为: " << typeid(T2).name() << endl;
}
```

//1、如果函数模板和普通函数都可以实现，优先调用普通函数

// 注意 如果告诉编译器 普通函数是有的，但只是声明没有实现，或者不在当前文件内实现，就会报错找不到

```
int a = 10;
int b = 20;
myPrint(a, b); //调用普通函数
```

//2、可以通过空模板参数列表来强制调用函数模板

```
myPrint<>(a, b); //调用函数模板
```

//3、函数模板也可以发生重载

```
int c = 30;
myPrint(a, b, c); //调用重载的函数模板
```

//4、如果函数模板可以产生更好的匹配，优先调用函数模板

```
char c1 = 'a';
char c2 = 'b';
myPrint(c1, c2); //调用函数模板
```

```
C++
1  template<class T>
2  void f(T a, T b)
3  {
4      if(a > b) { ... }
5  }
```

在上述代码中，如果T的数据类型传入的是像Person这样的自定义数据类型，也无法正常运行

因此C++为了解决这种问题，提供模板的重载，可以为这些**特定的类型**提供**具体化的模板**

//具体化，显示具体化的原型和定意思以template<>开头，并通过名称来指出类型

//具体化优先于常规模板

```
template<> bool myCompare(Person &p1, Person &p2)
```

```

Person p1("Tom", 10);
Person p2("Tom", 10);
//自定义数据类型，不会调用普通的函数模板
//可以创建具体化的Person数据类型的模板，用于特殊处理这个类型

```

类模板

```

template<class NameType, class AgeType>
class Person
{
public:
    Person(NameType name, AgeType age)
    {
        this->mName = name;
        this->mAge = age;
    }
    void showPerson()
    {
        cout << "name: " << this->mName << " age: " << this->mAge << endl;
    }
public:
    NameType mName;
    AgeType mAge;
};

```

类模板与函数模板区别主要有两点：

1. 类模板没有自动类型推导的使用方式
2. 类模板在模板参数列表中可以有默认参数

```

//只能利用全局函数重载左移运算符
ostream & operator<<(ostream &out, Person &p) //本质 operator<< (cout, p) 简化 cout
{
    out << "m_A = " << p.m_A << " m_B = " << p.m_B;
    return out;
}

```

```

//继承的好处：减少重复代码
//语法： class 子类 : 继承方式 父类

```

//2、通过类名访问

```
cout << "通过类名访问：" << endl;
cout << "Son 下 m_A = " << Son::m_A << endl;
cout << "Base 下 m_A = " << Son::Base::m_A << endl;
```

//子类出现和父类同名静态成员函数，也会隐藏父类中所有同名成员函数
//如果想访问父类中被隐藏同名成员，需要加作用域

```
Son::Base::func(100);
```

4.1.1 函数对象概念

概念:

- 重载函数调用操作符的类，其对象常称为**函数对象**
- 函数对象使用重载的()时，行为类似函数调用，也叫**仿函数**

本质:

函数对象(仿函数)是一个**类**，不是一个函数

4.2.1 谓词概念

概念:

- 返回bool类型的仿函数称为**谓词**
 - 如果operator()接受一个参数，那么叫做一元谓词
 - 如果operator()接受两个参数，那么叫做二元谓词
-
- 实现四则运算
 - 其中negate是一元运算，其他都是二元运算

仿函数原型:

- `template<class T> T plus<T>` //加法仿函数
- `template<class T> T minus<T>` //减法仿函数
- `template<class T> T multiplies<T>` //乘法仿函数
- `template<class T> T divides<T>` //除法仿函数
- `template<class T> T modulus<T>` //取模仿函数
- `template<class T> T negate<T>` //取反仿函数

- 实现关系对比

仿函数原型:

- `template<class T> bool equal_to<T> //等于`
- `template<class T> bool not_equal_to<T> //不等于`
- `template<class T> bool greater<T> //大于`
- `template<class T> bool greater_equal<T> //大于等于`
- `template<class T> bool less<T> //小于`
- `template<class T> bool less_equal<T> //小于等于`

- 实现逻辑运算

函数原型:

- `template<class T> bool logical_and<T> //逻辑与`
- `template<class T> bool logical_or<T> //逻辑或`
- `template<class T> bool logical_not<T> //逻辑非`

STL 常用算法

- 算法主要是由头文件 `<algorithm>` `<functional>` `<numeric>` 组成。
- `<algorithm>` 是所有STL头文件中最大的一个, 范围涉及到比较、交换、查找、遍历操作、复制、修改等等
- `<numeric>` 体积很小, 只包括几个在序列上面进行简单数学运算的模板函数
- `<functional>` 定义了一些模板类,用以声明函数对象。

- `for_each` //遍历容器
- `transform` //搬运容器到另一个容器中

```
for_each(v.begin(), v.end(), print01);
cout << endl;

for_each(v.begin(), v.end(), print02());
cout << endl;
```


- `transform(iterator beg1, iterator end1, iterator beg2, _func);`

//beg1 源容器开始迭代器

//end1 源容器结束迭代器

//beg2 目标容器开始迭代器

//_func 函数或者函数对象

- `find` //查找元素
- `find_if` //按条件查找元素
- `adjacent_find` //查找相邻重复元素
- `binary_search` //二分查找法
- `count` //统计元素个数
- `count_if` //按条件统计元素个数

- `find(iterator beg, iterator end, value);`

// 按值查找元素, 找到返回指定位置迭代器, 找不到返回结束迭代器位置

// beg 开始迭代器

// end 结束迭代器

// value 查找的元素

- `find_if(iterator beg, iterator end, _Pred);`

// 按值查找元素, 找到返回指定位置迭代器, 找不到返回结束迭代器位置

// beg 开始迭代器

// end 结束迭代器

// _Pred 函数或者谓词 (返回bool类型的仿函数)

- `adjacent_find(iterator beg, iterator end);`

// 查找相邻重复元素, 返回相邻元素的第一个位置的迭代器

// beg 开始迭代器

// end 结束迭代器

- `bool binary_search(iterator beg, iterator end, value);`

// 查找指定的元素, 查到 返回true 否则false

// 注意: 在**无序序列中不可用**

// beg 开始迭代器

// end 结束迭代器

// value 查找的元素

- `count(iterator beg, iterator end, value);`

// 统计元素出现次数

// beg 开始迭代器

// end 结束迭代器

// value 统计的元素

- `count_if(iterator beg, iterator end, _Pred);`

// 按条件统计元素出现次数

// beg 开始迭代器

// end 结束迭代器

// _Pred 谓词

- `sort` //对容器内元素进行排序
- `random_shuffle` //洗牌 指定范围内的元素随机调整次序
- `merge` // 容器元素合并, 并存储到另一容器中
- `reverse` // 反转指定范围的元素

- `sort(iterator beg, iterator end, _Pred);`

// 按值查找元素, 找到返回指定位置迭代器, 找不到返回结束迭代器位置

// beg 开始迭代器

// end 结束迭代器

// _Pred 谓词

- `random_shuffle(iterator beg, iterator end);`

// 指定范围内的元素随机调整次序

// beg 开始迭代器

// end 结束迭代器
- `merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`

// 容器元素合并，并存储到另一容器中

// 注意: 两个容器必须是**有序的**

// beg1 容器1开始迭代器
// end1 容器1结束迭代器
// beg2 容器2开始迭代器
// end2 容器2结束迭代器
// dest 目标容器开始迭代器
- `reverse(iterator beg, iterator end);`

// 反转指定范围的元素

// beg 开始迭代器

// end 结束迭代器
- `copy` // 容器内指定范围的元素拷贝到另一容器中
- `replace` // 将容器内指定范围的旧元素修改为新元素
- `replace_if` // 容器内指定范围满足条件的元素替换为新元素
- `swap` // 互换两个容器的元素
- `copy(iterator beg, iterator end, iterator dest);`

// 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置

// beg 开始迭代器

// end 结束迭代器

// dest 目标起始迭代器

- `replace(iterator beg, iterator end, oldvalue, newvalue);`

// 将区间内旧元素 替换成 新元素

// beg 开始迭代器

// end 结束迭代器

// oldvalue 旧元素

// newvalue 新元素

- `replace_if(iterator beg, iterator end, _pred, newvalue);`

// 按条件替换元素, 满足条件的替换成指定元素

// beg 开始迭代器

// end 结束迭代器

// _pred 谓词

// newvalue 替换的新元素

- `swap(container c1, container c2);`

// 互换两个容器的元素

// c1容器1

// c2容器2

- 算术生成算法属于小型算法, 使用时包含的头文件为 `#include <numeric>`

算法简介:

- `accumulate` // 计算容器元素累计总和

- `fill` // 向容器中添加元素

- `accumulate(iterator beg, iterator end, value);`

// 计算容器元素累计总和

// beg 开始迭代器

// end 结束迭代器

// value 起始值

- `set_intersection` // 求两个容器的交集
- `set_union` // 求两个容器的并集
- `set_difference` // 求两个容器的差集

- `set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`

// 求两个集合的交集

// 注意:两个集合必须是有序序列

// beg1 容器1开始迭代器

// end1 容器1结束迭代器

// beg2 容器2开始迭代器

// end2 容器2结束迭代器

// dest 目标容器开始迭代器

- `set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`

// 求两个集合的并集

// 注意:两个集合必须是有序序列

// beg1 容器1开始迭代器

// end1 容器1结束迭代器

// beg2 容器2开始迭代器

// end2 容器2结束迭代器

// dest 目标容器开始迭代器

- 求并集的两个集合必须的有序序列
- 目标容器开辟空间需要两个容器相加
- `set_union`返回值既是并集中最后一个元素的位置

- `set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);`

// 求两个集合的差集

// 注意:两个集合必须是有序序列

// beg1 容器1开始迭代器

// end1 容器1结束迭代器

// beg2 容器2开始迭代器

// end2 容器2结束迭代器

// dest 目标容器开始迭代器

- 求差集的两个集合必须的有序序列
- 目标容器开辟空间需要从两个容器取较大值
- `set_difference`返回值既是差集中最后一个元素的位置