



C++ Programming

类和对象 II Classes and Objects II

2025年3月10日

学而不厌 诲人不倦

- ➡ **3.1 构造与析构函数**
- ➡ **3.2 对象数组与对象指针**
- ➡ **3.3 共用数据的保护**
- ➡ **3.4 对象动态建立、释放、赋值与复制**
- ➡ **3.5 静态成员和友元**
- ➡ **3.6 类模板**

3.1 构造函数与析构函数

Constructor

➤ 对象的初始化：构造函数

构造函数是一种特殊的成员函数，在程序中不需要写调用语句，在系统建立对象时由系统**自觉调用执行**，**用于为对象分配空间和进行初始化**。

```
#include <iostream>
using namespace std;

class Time //定义CPoint类
{
private:
    int nHour;//小时
    int nMinute;//分钟
    int nSecond;//秒
public:
    Time();//构造函数
    //构造函数重载
    Time(int h, int m, int s);
    void ShowTime();
};
```

构造函数的特点：

- (1) 构造函数的名字与它的类名必须相同。
- (2) 没有类型，与不返回值
- (3) 可以带参数，也可以不带参数。
- (4) 构造函数可以重载
- (5) 可以写成带参数初始化表的形式

```
Box::Box( int h, int w, int len): height ( h),width(w), length( len ){ }
```

3.1 构造函数与析构函数

Constructor

➤ 对象的初始化：构造函数

构造函数不需要显式地调用，构造函数是在**建立对象时**由**系统自动执行的**，**且只执行一次**。构造函数一般定义为**public**类型。

```
Time::Time()
{
    nHour = 0;
    nMinute = 0;
    nSecond = 0;
}

Time::Time(int h, int m, int s)
{
    nHour = h;
    nMinute = m;
    nSecond = s;
}
```

```
int main()
{
    Time t1;
    t1.ShowTime();
    Time t2(12,20,15);
    t2.ShowTime();
    return 0;
}
```

用参数初始化
表对数据成员
初始化

```
void Time::ShowTime()
{
    cout<<"Current Time:
"<<nHour<<":"<<nMinute<<":"<<nSecond<<endl;
}
```



3.1 构造函数与析构函数

➤ 使用默认参数值的构造函数

Constructor

```
#include <iostream>
using namespace std;
class Box
{
public:
    Box(int,int,int);
    Box(int w=10,int h=10,int len=10);
    int volume();
private:
    int height;
    int width;
    int length;
};

// 长方体构造函数
Box::Box(int h,int w,int len)
{
    height=h;
    width=w;
    length=len;
}
```

```
// 计算长方体的体积
int Box::volume()
{
    return(height*width*length);
}

int main()
{
    Box box1;
    cout<<"box1 体积= "<<box1.volume()<<endl;
    Box box2(15);
    cout<<"box2 体积 "<<box2.volume()<<endl;
    Box box3(15,30);
    cout<<"box3 体积 "<<box3.volume()<<endl;
    Box box4(15,30,20);
    cout<<"box4 体积"<<box4.volume()<<endl;
    return 0;
}
```

3.1 构造函数与析构函数

➤ 对象的销毁：析构函数

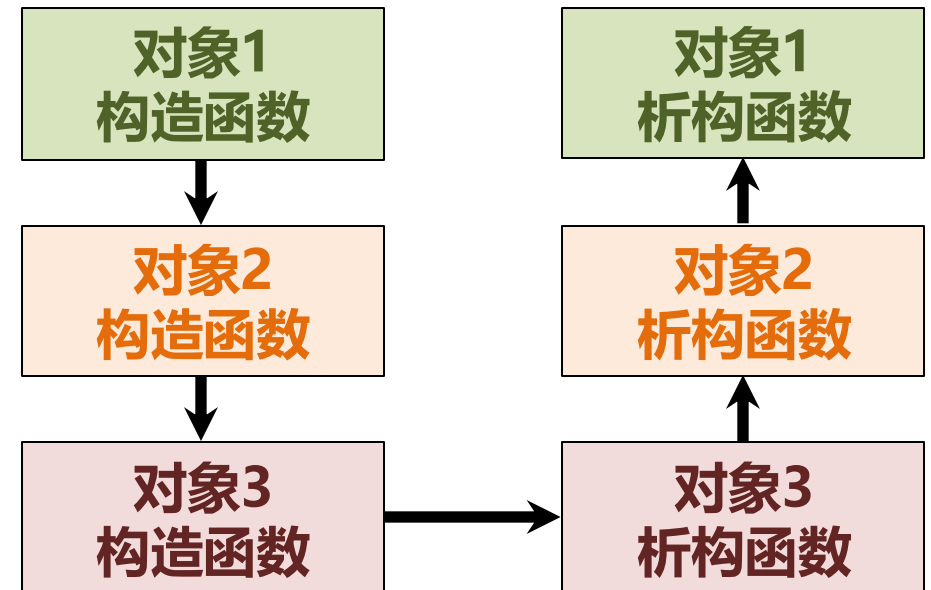
Destructor

析构函数也是个特殊的成员函数，作用与构造函数相反，当对象的生命周期结束时，系统**自动调用析构函数**，收回对象占用的内存空间。

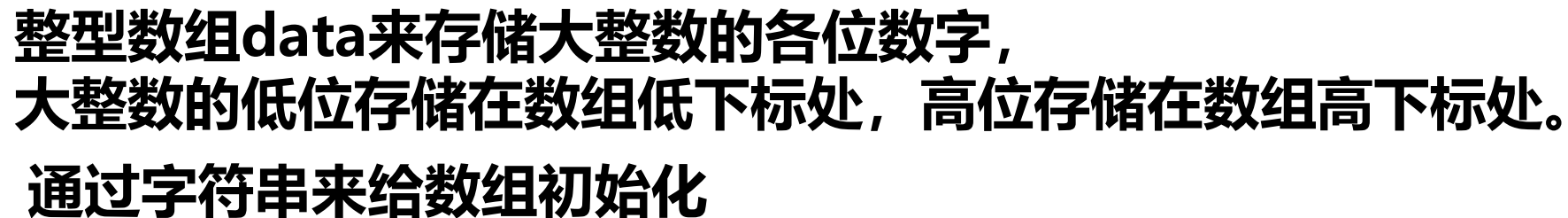
析构函数的特点：

- ①子函数结束、main函数结束、exit命令执行、delete对象时自动执行析构函数
- ②先构造的后析构，后构造的先析构
- ③以~符号开始后跟类名，没有数据类型、返回值、形参

```
~CBigInt();
```



Data = 12345678999955667



7/29



3.1 构造函数与析构函数

➤ 例题：定义一个存储大整数的数据类型CBigInt

```
#include <iostream>
using namespace std;

class CBigInt //定义CBigInt类
{
private:
    int *data;//数据
    int length;//BigInt数据长度

public:
    CBigInt();//构造函数
    CBigInt(string);//带参构造函数
    CBigInt(const CBigInt &b);//复制构造函数
    ~CBigInt();
    void Print();
};
```

```
CBigInt::CBigInt()
{
    length =0;
    data =NULL;
}

//由一个包含数字的字符串构造一个大整数
CBigInt::CBigInt(string str)
{
    length = str.length();
    data = new int [length];
    for (int i=length-1;i>=0;i--)
        data[length-i-1] = (int) (str[i]-'0');
}

//析构函数，使用delete释放存储空间
CBigInt::~CBigInt()
{
    length = 0;
    delete []data;
}
```


3.1 构造函数与析构函数

➤ 例题：定义一个存储大整数的数据类型CBigInt

```
#include <iostream>
using namespace std;

class CBigInt //定义CBigInt类
{
private:
    int *data;//数据
    int length;//BigInt数据长度

public:
    CBigInt();//构造函数
    CBigInt(string);//带参构造函数
    CBigInt(const CBigInt &b);//复制构造函数
    ~CBigInt();
    void Print();
};
```

//复制构造函数,重新申请内存空间将b中数据复制到当前对象

```
CBigInt::CBigInt(const CBigInt &b)
{
    length = b.length;
    data = new int [length];
    for (int i=0;i<length;i++)
        data[i] = b.data[i];
}
```

```
int main()
{
    string s = "12345678999955667";
    CBigInt b1(s);//自动调用有参构造函数
    b1.Print();
    CBigInt b2=b1;//自动调用复制构造函数
    b2.Print();
    //cout<<&b1<<" "<<&b2<<endl;
}
```

- ➡ 3.1 构造与析构函数
- ➡ 3.2 对象数组与对象指针
- ➡ 3.3 共用数据的保护
- ➡ 3.4 对象动态建立、释放、赋值与复制
- ➡ 3.5 静态成员和友元
- ➡ 3.6 类模板

3.2 对象数组与对象指针

➤ 对象数组：一个对象数组中各个元素都是同类对象

student std[50]; //一个班级有50个学生，
每个学生具有学号、年龄、成绩等属性

```
#include <iostream>
using namespace std;
class Box
{ public:
    Box( int h=10, int w=12, int len=15 ):height(h),
    width(w), length(len) { } // 带默认参数值和参数表
    int volume();
private:
    int height;
    int width;
    int length;
};
```

```
int Box::volume()
{ return(height*width*length); }

int main()
{
    Box a[3]={ Box(10,12,15),
    Box(15,18,20),
    Box(16,20,26) };
    cout<<"a[0]的体积是 "<<a[0].volume()<<endl;
    cout<<"a[1]的体积是 "<<a[1].volume()<<endl;
    cout<<"a[2]的体积是 "<<a[2].volume()<<endl;
    return 0;
}
```



3.2 对象数组与对象指针

➤ 对象指针：指向对象的指针

C++中定义对象的指针变量与定义其他的指针变量相似，格式如下：

类名 * 变量名表

C++的对象也可以参加**取地址运算**。
&对象名

```
Time *pt; // 定义pt是指向Time类对象的指针
Time t1; // 定义Time类对象t1
pt = &t1; // 将对象t1的地址赋予pt
```

程序在此之后就可以用指针变量访问对象的成员。

```
(*pt).hour
pt->hour
(*pt).show_time()
pt->show_time()
```

```
#include <iostream>
using namespace std;

class Time //定义CPoint类
{
private:
    int nHour;//小时
    int nMinute;//分钟
    int nSecond;//秒
public:
    Time();//构造函数
    //构造函数重载
    Time(int h, int m, int s);
    void ShowTime();
};
```

3.2 对象数组与对象指针

➤ **对象指针：指向对象公有数据成员的指针**

对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。

指向对象成员的指针分指向数据成员的指针和指向成员函数的指针

①**定义数据成员的指针变量： 数据类型 * 指针变量名**

②**计算公有数据成员的地址： &对象名.成员名**

```
Time t1;  
int * p1; // 定义一个指向整型数据的指针变量  
p1 = & t1.hour; // 假定hour是公有成员  
cout<< *p1 << endl;
```



3.2 对象数组与对象指针

➤ 对象指针：指向对象成员函数的指针

对象由成员组成。对象占据的内存区是各个数据成员占据的内存区的总和。

指向对象成员的指针分指向数据成员的指针和指向成员函数的指针

① 定义指向成员函数的指针变量

数据类型 (**类名**::* **变量名**) (**形参表**);

数据类型是成员函数的类型。

类名是对象所属的类

变量名按标识符取名

形参表：指定成员函数的形参表(形参个数、类型)

② 取成员函数的地址

&类名::成员函数名

③ 给指针变量赋初值

指针变量名 = & 类名::成员函数名;

④ 用指针变量调用成员函数

(对象名.*指针变量名)([实参表]);

3.2 对象数组与对象指针

➤ 对象指针：指向对象成员函数的指针

```
#include <iostream>
using namespace std;
class Time
{
public:
    Time(int,int,int);
    void get_time();
private:
    int hour;
    int minute;
    int sec;
};
```

```
int main()
{
    Time t1(10,13,56);
    int *p1=&t1.hour; // 定义指向成员的指针p1
    cout<<*p1<<endl;
    t1.get_time(); // 调用成员函数
    Time *p2=&t1; // 定义指向对象t1的指针p2
    p2->get_time(); // 用对象指针调用成员函数

    void (Time::*p3)(); // 定义指向成员函数的指针
    p3=&Time::get_time; // 给成员函数的指针赋值
    (t1.*p3)(); // 用指向成员函数的指针调用成员函数
    return 0;
}
```

```
void (Time::*p3) = &Time::get_time;
```

3.2 对象数组与对象指针

➤ 对象指针: **this**指针

一个类的成员函数只有一个内存拷贝。类中不论哪个对象调用某个成员函数，调用的都是内存中同一个成员函数代码。

```
int Box :: volume()  
{ return ( height * width * length ) ; }
```

C++编译成

```
int Box :: volume( * this )  
{ //*this 就是this所指的对象，即：调用成员函数的对象  
  return ( this->height* this->width * this->length);  
  //或者  
  return( (*this).height* (*this).width * (*this). length );  
}
```

C++ 通过编译程序，在对象调用成员函数时，把对象的地址赋予**this** 指针，用**this** 指针指向对象，实现了用同一个成员函数访问不同对象的数据成员。

- ➡ 3.1 构造与析构函数
- ➡ 3.2 对象数组与对象指针
- ➡ **3.3 共用数据的保护**
- ➡ 3.4 对象动态建立、释放、赋值与复制
- ➡ 3.5 静态成员和友元
- ➡ 3.6 类模板

3.3 共用数据的保护

➤ 1. const 常对象

即希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供**const**手段，将**对象**、**数据**、**成员函数**指定为常量，从而实现了只读要求，**达到保护数据的目的**。

定义格式：**const 类名 对象名(实参表);**

或：**类名 const 对象名(实参表);**

```
const Time t1( 10,15,36);
```

```
t1.get_time(); // 错误, 不能调用
```

为了访问常对象中的数据成员，要定义常成员函数：

```
void get_time() const
```

把对象定义为常对象，对象中的数据成员就是常变量，在定义时必须带实参作为数据成员的初值，在**程序中不允许修改常对象的数据成员值**。

3.3 共用数据的保护

➤ 2. const 常数据成员

即希望数据在一定范围内共享，又不愿它被随意修改，从技术上可以把数据指定为只读型的。C++提供**const**手段，将**对象**、**数据**、**成员函数**指定为常量，从而实现了只读要求，**达到保护数据的目的**。

格式： **const** **类型** **数据成员名**

将类中的数据成员定义为具有只读的性质。

例：

```
const int hour;  
Time::Time( int h)  
{   hour = h; ...} // 错误
```

应该写成：

```
Time::Time( int h ) : hour (h) {}
```

注意只能通过带参数初始表的构造函数对常数据成员进行初始化。



3.3 共用数据的保护

➤ 3. const 常成员函数

任何不修改数据成员的函数，都应该声明为const类型，以提高程序的健壮性。

定义格式：

类型 函数名 (形参表) const

const 是函数类型的一部分，在**声明函数原型**和**定义函数**时都要用const关键字。

```
float GetDistance() const;  
void ShowPoint() const;
```

```
float d = p1.GetDistance();
```

常对象只能通过常成员函数读数据成员，
常对象不能调用非const成员函数。

常成员函数不能调用非const成员函数。

```
//输出当前对象坐标  
void CPoint::ShowPoint() const {  
    cout<<"Point: ["<<x<<","<<y<<"]"<<endl;  
}
```

```
//计算当前对象到原点距离  
float CPoint::GetDistance() const {  
    x=5;//const成员函数中无法修改成员变量数值  
    return sqrt(x*x+y*y);  
}
```

3.3 共用数据的保护

➤ 3. const 常成员函数

一般成员函数可以访问或修改本类中的非 const 数据成员。而常成员函数只能读本类中的数据成员，而不能写它们。

表 3.1

数据成员	非 const 成员函数	const 成员函数
非 const 的数据成员	可以引用,也可以改变值	可以引用,但不可以改变值
const 数据成员	可以引用,但不可以改变值	可以引用,但不可以改变值
const 对象的数据成员	不允许引用和改变值	可以引用,但不可以改变值

3.3 共用数据的保护

➤ 4. 指向对象的常指针

如果在定义指向对象的指针时，使用了关键字 `const`，它就是一个常指针，必须在定义时对其初始化。并且在程序运行中不能再修改指针的值。

类名 * const 指针变量名 = 对象地址

例：Time t1(10,12,15), t2;

Time * const p1 = &t1;

在此后，程序中不能修改p1。

例：Time * const p1 = &t2; **// 错误语句**



3.3 共用数据的保护

➤ 5. 指向常对象的指针变量

const 类名 * 指针变量名 = 对象地址

- (1) 如果一个对象已被声明为常对象，只能用指向常对象的指针变量指向它。
- (2) 如果定义了一个指向常对象的指针变量，并使它指向一个非const对象，则指向的对象是不能通过该指针变量来改变的。
- (3) 常用作函数的形参，保护形参指针所指向的对象不被修改。
- (4) 指针变量本身的值是可以改变的

3.3 共用数据的保护

➤ 6. 对象的常引用

如果用引用形参又不想让函数修改实参，可以使用常引用机制。

格式： **const** 类名 & 形参对象名

常引用作为函数形参，形参与实参引用同一块内存空间，既节省存储，同时又能保证数据不能被随意修改。

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
}
```


3.3 共用数据的保护

➤ 6. 对象的常引用

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
    int flag = 0; //表示进位
    int i = 0;
    int m= a.length;
    int n= b.length;
    int len=m>n?m:n;//求长度较大者
    data = new int [len+1]; //申请空间，最高位可能有进位，因此多申请一位空间。
    while (i<m && i<n) //逐位相加，直到一个大整数结束。
    {
        data[i] = (a.data[i]+b.data[i]+flag)%10;
        flag = (a.data[i]+b.data[i]+flag)/10;
        i++;
    }
    //待续
}
```

3.3 共用数据的保护

➤ 6. 对象的常引用

```
void CBigInt::Add(const CBigInt &a, const CBigInt &b)
{
    //续
    for (;i<m;i++)//如果大整数a没有结束，则剩余位数与进位相加
    {
        data[i] = (a.data[i]+flag)%10;
        flag = (a.data[i]+flag)/10;
    }
    for (;i<n;i++)//如果大整数b没有结束，则剩余位数与进位相加
    {
        data[i] = (b.data[i]+flag)%10;
        flag = (b.data[i]+flag)/10;
    }
    length = len + flag; //a+b的长度与最后的进位有关。
    if(flag)
        data[length-1]= flag;
}
```

3.3 共用数据的保护

➤ 6. 对象的常引用

```
int main()
{
    string s ="12345678999955667";
    CBigInt b1(s); //自动调用有参构造函数
    b1.Print();
    CBigInt b2=b1; //自动调用复制构造函数
    b2.Print();
    cout<<&b1<<" "<<&b2<<endl;
    CBigInt b3;
    b3.Add(b1,b2);
    b3.Print();
}
```

```
BigInt:12345678999955667
BigInt:12345678999955667
0x16bc6f468  0x16bc6f430
BigInt:24691357999911334
```

- ➡ **3.1 构造与析构函数**
- ➡ **3.2 对象数组与对象指针**
- ➡ **3.3 共用数据的保护**
- ➡ **3.4 对象动态建立、释放、赋值与复制**
- ➡ **3.5 静态成员和友元**
- ➡ **3.6 类模板**



Thank You !

Q & A