第六章 数据抽象——对象与类

6.1 数据抽象概述

6.1.1 数据抽象与封装

- **抽象**是指该程序实体外部可观察到的行为，不考虑该程序实体的内部是如何实现的。（复杂度控制）
- **封装**是指把该程序实体内部的具体实现细节对使用者隐藏起来，只对外提供一个接口。（信息保护）
- **过程抽象**
  - 用一个名字来代表一段完成一定功能的程序代码，代码的使用者只需要知道代码的名字以及相应的功能，而不需要知道对应的程序代码是如何实现的。
- **过程封装**
  - 把命名代码的具体实现隐藏起来（对使用者不可见，或不可直接访问），使用者只能通过代码名字来使用相应的代码。
    命名代码所需要的数据是通过参数来获得，计算结果通过返回值机制返回。
- 数据抽象
  - 只需要知道对数据能实施哪些操作以及这些操作之间的关系，数据的使用者不需要知道数据的具体表示形式（数组或链表等）。
- 数据封装
  - 把数据及其操作作为一个整体（封装体）来进行实现，其中，数据的具体表示被隐藏起来（使用者不可见，或不可直接访问），对数据的访问（使用）只能通过封装体对外接口中提供的操作来完成。

6.1.2 面向对象程序设计

面向对象程序设计（Object-oriented Programming）有如下特征：

- 程序由若干**对象**组成，每个对象是由一些**数据**以及对这些数据所能实施的**操作**所构成的**封装体**
- 对象的特征（数据、操作、对外接口）由相应的类来描述，一个类所描述的对象特征可以从其他类继承
- 对数据的操作是通过向包含数据的对象发送**消息**（调用对象类对外接口中的操作）来实现，体现了**数据抽象**

在面向/基于对象的程序设计中，对象/类体现了数据抽象与封装

基本概念：

1. 对象与类

对象：由数据及对数据的操作所构成的封装体。面向对象程序的基本计算单位

面向对象程序的执行过程：对象间的一系列**消息传递**

- 从程序外部向程序中的某个对象发送第一条消息启动计算过程
- 该对象在处理这条消息的过程中，又向程序中的其它对象发送消息，从而引起进一步的计算

类：描述对象的基本特征（包括数据与操作）。

- 对象包含的数据：数据成员、成员变量、实例变量、对象的局部变量……
- 对象的操作：成员函数、方法、消息处理过程

2. 继承

- 在定义一个类时，可以利用已有类的一些特征描述，即先把已有类的一些特征描述包含进来，再定义新的特征
- 涉及两个类：父类和子类（C++中：基类和派生类）。子类包含父类的特征，且拥有新的特征，并能对父类的特征重新定义
- 单继承和多继承（第 7 章）

3. 多态性与动态绑定

多态性：某一论域中的一个元素存在多种形式和解释。

- **一名多用**：函数名重载和操作符重载（6.7节）
- **类属**：一个程序实体能对多种类型的数据进行操作或描述的特性
  - 类属函数：一个函数能对多种类型的参数进行操作
  - 类属类型：一个类型可以描述多种类型的数据
- 面向对象程序特有的多态（继承机制带来的）：
  - **对象类型**的多态：子类对象既属于子类，也属于父类
  - **对象标识**的多态：父类的引用或指针可以引用或指向父类对象，也可以引用或指向子类对象
  - **消息**的多态：发给父类对象的消息也能发给子类对象，但会给出不同的解释（处理）

绑定：确定对多态元素的某个使用是多态元素的哪一种形式。

- **静态绑定**：在编译时刻确定对多态元素的使用
- **动态绑定**：在程序运行时来确定对多态元素的使用
- 大多数形式的多态可采用静态绑定。如函数名重载
- 由于存在对象标识的多态和消息的多态，消息处理有时要采用动态绑定

多态性的好处：

- 使得程序功能扩充变得容易：程序上层代码不变，只要增加底层的多态元素即可。
- 增强语言的可扩充性，如操作符重载等。

6.1.3 面向对象程序设计与过程式程序设计的对比

影响软件开发效率和软件质量的因素主要包括：

- 抽象（控制复杂度）
- 封装（保护信息）
- 模块化（组织和管理大型程序）
- 软件复用（缩短开发周期）
- 可维护性（延长软件寿命）
- 软件模型的自然度（缩小解题空间与问题空间之间的语义间隙，实现从问题到解决方案的自然过渡）

过程式程序设计的特点：

- 以功能为中心，强调过程（功能）抽象，但数据与操作分离，二者联系松散。
- 实现了操作的封装，但数据表示是公开的，数据缺乏保护。
- 按子程序划分模块，模块边界模糊。
- 子程序往往针对某个程序的具体功能而设计，这使得程序难以复用。

- 功能易变，程序维护困难。
- 基于子程序的解题方式与问题空间缺乏对应。

面向对象程序设计的特点：

- 以数据为中心，强调数据抽象，操作依附于数据，二者联系紧密。
- 实现了数据的封装，加强了数据的保护。
- 按对象类划分模块，模块边界清晰。
- 对象类往往具有通用性，再加上继承机制，使得程序容易复用。
- 对象类相对稳定，有利于程序维护。
- 基于对象及其交互的解题方式与问题空间有很好的对应。

6.2 类

- 对象构成了面向对象程序的基本计算单位，而对象的特征则由相应的类来描述。
- 对象是用类来创建的，因此，程序中首先要定义类。
- C++的类是一种用户自定义类型，定义形式如下：

```
class <类名>
{
    <成员描述>
};
```

  - 成员包括：数据成员和成员函数
  - 类成员标识符的作用域为整个类定义范围

6.2.1 数据成员

数据成员：类的对象所包含的数据（常量和变量）。在类定义中需要对数据成员的名字和类型进行说明

数据成员的类型可以是任意C++类型（包括类，void除外）

在声明一个数据成员的类型时，若未见相应的类型定义或相应的类型未定义完，则该数据成员的类型一般只能是这些类型的指针或引用类型。

```
class A;    //A是在程序其它地方定义的类，这里是声明。
class B
{   A a;    //Error，未见A的定义。
    B b;    //Error，B还未定义完，递归了
    A *p;   //OK
    B *q;   //OK
    A &aa;  //OK
    B &bb;  //OK
};
```

在类定义中描述数据成员一般不能给它们赋初值，数据成员的初始化应在类的构造函数中

```
class A;
{
    int x = 0;              //Error
    const double y = 0.0;   //Error
}
```

6.2.2 成员函数

成员函数：在类中定义的函数，描述了对类中定义的数据成员所能实施的操作

- 成员函数的实现（函数体）可以放在类定义中

```
class A
{
    ......
    void f() {...} //建议编译器按内联函数处理
};
```

成员函数的实现也可以放在类定义外，例如：

```
class A
{
    ......
    void f(); //声明
};
void A::f() { ... } //要用类名受限，区别于非成员函数（全局函数）
```

类成员函数名是可以重载的（析构函数除外），它遵循一般函数名的重载规则

```
class A
{
    ......
    void f();
    int f(int i);
    double f(double d);
};
```

6.2.3 成员的访问控制——信息隐藏

在C++的类定义中，可以用下面的成员访问控制修饰符来控制在类的外部对类成员的访问限制：

- public：访问不受限制
- private：只能在本类和友元的代码中访问
- protected：只能在本类、派生类和友元的代码中访问

```
class A
{
public:     //访问不受限制。
    int x;
    void f();
private:    //只能在本类和友元的代码中访问。
    int y;
    void g();
protected:  //只能在本类、派生类和友元的代码中访问。
    int z;
    void h();
};
```

C++中类成员默认访问控制是 `private`，且类定义中可以有多个 `public`、`private` 和 `protected` 访问控制说明

- 一般来说，类的数据成员和在类的内部使用的成员函数应该指定为 `private`，只有提供给外界使用的成员函数才指定为 `public`。
    - 具有 `public` 访问控制的成员构成了类与外界的一种**接口**（interface）。
    - 在一个类的外部只能访问该类接口中的成员。
- `protected` 类成员访问控制具有特殊的作用（在派生类中使用）。

6.3 对象

- 类属于类型范畴的程序实体，它一般存在于静态的程序（编译程序看到的）中。
- 而动态的面向对象程序（运行中的）则是由对象构成。
- 对象在程序运行时创建。

6.3.1 对象的创建

1. **直接创建对象**：在程序中定义一个类型为某个类的变量来实现的，其格式与普通变量的定义相同

```
class A
{
public:
    void f();
    void g();
private:
    int x,y;
}
......
A a1;       //创建一个A类的对象。
A a2[100];  //创建100个A类对象。
```

直接创建的对象分为：**全局对象**（在所有函数外定义的对象）和**局部对象**（在函数或复合语句内定义的对象）

2. **间接创建对象**：在程序运行中通过 `new` 操作来创建一个类型为某个类的动态变量，这样的动态变量称为**动态对象**，它们的内存空间在程序的堆区中分配，用指针变量来标识他们

- 单个动态对象的创建与撤销

```
A *p;
p = new A;  // 创建一个A类的动态对象。
… *p …      //或，p->...，通过p访问动态对象
delete p;   // 撤消p所指向的动态对象
```

- 动态对象数组的创建与撤消

```
A *q;
q = new A[100]; //创建一个动态对象数组。
...q[i]...      //或，*(q+i)，访问动态对象数组中的第i个对象
delete []q;     //撤消q所指向的动态对象数组。
```

一般不用 `malloc` 和 `free`

- 成员对象的创建

```
class A
{
……;
}
class B
{
    A a;    //数据成员的类型是另一个类
    ……;
}
B b;        //b 包含了一个成员对象b.a
```

成员对象的生存期与包含它的对象相同

6.3.2 对象的操作

通过对象类中的 `public` 成员函数实现

- 非动态对象：<对象>.<类成员>
- 动态对象：<对象指针> -> <类成员> 或 *<对象指针>.<类成员>

```
class A
{       int x;
    public:
        void f();
};
int main()
{   A a;    //创建A类的一个局部对象a。
    a.f();  //调用A类的成员函数f对对象a进行操作。
    A *p=new A;  //创建A类的一个动态对象，p指向之。
    p->f();     //调用A类的成员函数f对p所指向的对象进行操作。
    delete p;
    return 0;
}
```

在类的外部，通过对象来访问类的成员时要受到类成员访问控制的限制

- 可以对同类对象进行赋值

- 取对象地址
- 把对象作为参数传给函数
- 把对象作为函数的返回值

6.3.3 this 指针

类中描述的数据成员（静态数据成员除外）对该类的每个对象分别有一个拷贝。类中的成员函数对该类所有对象只有一个拷贝。

- 类的每一个成员函数（静态成员函数除外）都有一个隐藏的形参 `this`，其类型为该类对象的指针；在成员函数中对类成员的访问是通过 `this` 来进行的
  - 对于 A 类的成员函数 g：`void g(int i) { x = i; }`
  - 编译程序将会把它编译成：`void g(A *const this, int i) { this->x = i; };`
- 当通过对象访问类的成员函数时，将会把相应对象的地址传给成员函数的参数 `this`
  - 对于下面的成员函数调用：`a.g(1);` 和 `b.g(2);`
  - 编译程序将会把它编译成：`g(&a,1);` 和 `g(&b,2);`

一般情况下，类的成员函数中不必显式使用 `this` 指针，编译程序会自动加上。但如果成员函数中要把 `this` 所指向的对象作为整体来操作，则需要显式地使用 `this` 指针

6.4 对象的初始化和消亡前的处理

6.4.1 构造函数

当一个对象创建时，它将获得一块存储空间，该存储空间用于存储对象的数据成员。在使用对象前，需要对对象存储空间中的数据成员进行初始化。

C++提供了一种对象初始化的机制：**构造函数**

1. 构造函数的定义
   - 在对象类中定义或声明的与类同名、无返回值类型的成员函数
   - 创建对象时，构造函数会被自动调用
   - 对构造函数的调用属于对象创建过程的一部分，对象创建之后不能再调用构造函数
   - 构造函数可以重载，其中，不带参数的（或所有参数都有默认值的）构造函数被称为**默认构造函数**
   - 在创建对象时，如果没有指定调用对象类中哪一个构造函数，则调用默认构造函数初始化。也可以显式地指定调用对象类的某个构造函数
2. 构造函数调用的指定

```
class A
{......
public:
    A();
    A(int i);
    A(char *p);
};
......
A a1;   //调用默认构造函数。也可写成：A a1=A(); 但不能写成：A a1();
A a2(1);        //调用A(int i)。也可写成：A a2=A(1); 或 A a2=1;
```

```
A a3("abcd");   //调A(char *)。也可写成：A a3=A("abcd"); 或 A a3="abcd";
A a[4];         //调用对象a[0]、a[1]、a[2]、a[3]的默认构造函数。
A b[5]={A(),A(1),A("abcd"),2,"xyz"};     //调用b[0]的A()、
                                         //b[1]的A(int)、b[2]的A(char *)、
                                         //b[3]的A(int)和b[4]的A(char *)
A *p1 = new A;          //调用默认构造函数
A *p2 = new A(2);       //调用A(int i)
A *p3 = new A("xyz");   //调用A(char *)
A *p4 = new A[20];      //创建动态对象数组时，只能调用各对象的默认构造函数
```

- 在程序中也可以通过类的构造函数来创建一些临时对象

3. 成员初始化表

- 对于常量和引用数据成员，C++旧版本不能在说明它们时初始化，也不能采用赋值操作在构造函数中对它们初始化。
- 可以在构造函数的函数头和函数体之间加入一个成员初始化表来对常量和引用数据成员进行初始化。
- 在成员初始化表中，成员的书写次序并不决定它们的初始化次序，它们的初始化次序由它们在类定义中的描述次序来决定。

6.4.2 析构函数

定义：名为“~<类名>”且没有参数和返回值类型的一个特殊的成员函数。对象消亡时，在系统收回它所占的内存空间之前，会自动调用对象类的析构函数。

一般情况下，类中不需要自定义析构函数，但如果对象创建后，自己又额外申请了资源（如：额外申请了内存空间），则可以自定义析构函数来归还它们。

析构函数可以显式调用，这时并不是让对象消亡，而是暂时归还对象额外申请的资源

6.4.3 成员对象的初始化和消亡处理

在创建包含成员对象的对象时，除了会自动调用本身类的构造函数外，还会自动去调用成员对象类的构造函数

- 通常是调用成员对象类的默认构造函数
- 如果要调用成员对象类的**非默认构造函数**，需要在包含成员对象的对象类的构造函数**成员初始化表**中显式指出

创建包含成员对象的对象时：

- 先执行成员对象类的构造函数，再执行本对象类的构造函数
- 若包含多个成员对象，这些成员对象的构造函数执行次序则按它们在本对象类中的说明次序进行
- 从实现上说，
    - 是先调用本身类的构造函数，但在进入函数体之前，会去调用成员对象类的构造函数，然后再执行本身类构造函数的函数体
    - 也就是说，构造函数的成员初始化表（即使没显式给出）中有对成员对象类的构造函数的调用代码
    - 注意：如果类中未提供任何构造函数，但它包含成员对象，则编译程序会隐式地为之提供一个默认构造函数，其作用就是调用成员对象类的构造函数

对象消亡时：

- 先执行本身类的析构函数，再执行成员对象类的析构函数
- 如果有多个成员对象，则成员对象析构函数的执行次序则按它们在本对象类中的说明次序的逆序进行

- 从实现上说，
  - 是先调用本身类的析构函数，本身类析构函数的函数体执行完之后，再去调用成员对象类的析构函数
  - 也就是说，析构函数的函数体最后有对成员对象类的析构函数的调用代码
  - 注意：如果类中未提供析构函数，但它包含成员对象，则编译程序会隐式地为之提供一个析构函数，其作用就是调用成员对象类的析构函数

```
class A
{
    int x;
public:
    A() { x = 0; }
    A(int i) { x = i; }
};
class B
{
    A a;
    int y;
public:
    B() { y = 0;}
    B(int i) { y = i; }
    B(int i, int j): a(j) { y = i; }
};
B b0;        //调用B()和A()：b0.y=0；b0.a.x=0
B b1(1);     //调用B(int)和A()：b1.y=1；b1.a.x=0
B b2(1,2);   //调用B(int,int)和A(int)：b2.y=1；b2.a.x=2
......
//b0、b1、b2消亡时，会分别去调用B和A的析构函数
```

6.4.4 拷贝构造函数

若一个构造函数的参数类型为本类的引用，则称它为拷贝构造函数（<类名>(const <类名>&);）。在创建一个对象时，若用另一个同类型的对象对其初始化，将会调用对象类中的拷贝构造函数。

在三种情况下，会调用类的拷贝构造函数：

- 创建对象时显式指出

```
A a1;
A a2(a1);  //创建对象a2，用对象a1初始化对象a2
```

- 把对象作为值参数传给函数时

```
void f(A x) { ..... };
......
A a;
f(a);  //创建形参对象x，用对象a对x进行初始化。
```

- 把对象作为函数的返回值时

```
A f()
{   A a;
    ......
    return a; //创建返回值对象，用对象a对返回值对象
              //进行初始化。
}
```

隐式拷贝构造函数

如果一个类定义中没有定义拷贝构造函数，则编译程序将会为其提供一个隐式的拷贝构造函数。

隐式拷贝构造函数将逐个成员进行拷贝初始化：

- 对于非对象成员：它采用通常的拷贝操作
- 对于成员对象：则调用成员对象类的拷贝构造函数来对成员对象进行初始化（递归定义）

一般情况下，编译程序提供的隐式拷贝构造函数的行为足以满足要求，类中不需要自定义拷贝构造函数。但在一些特殊情况下，必须要自定义拷贝构造函数，否则，将会产生设计者未意识到的严重的程序错误。

```
class String
{      int len;
       char *str;
    public:
        String(char *s)
        { len = strlen(s);
          str = new char[len+1];
          strcpy(str,s);
        }
        ~String() { delete []str; len=0; str=NULL; }
};
......
String s1("abcd");
String s2(s1);
```

隐式的拷贝构造函数将会使得 `s1` 和 `s2` 的成员指针 `str` 指向同一块内存区域

- 如果对一个对象（`s1`或`s2`）操作之后修改了这块空间的内容，则另一个对象将会受到影响。如果不是设计者特意所为，这将是一个隐藏的错误。
- 当对象 `s1` 和 `s2` 消亡时，将会分别去调用它们的析构函数，这会使得同一块内存区域将被归还两次，从而导致程序运行错误。
- 当对象 `s1` 和 `s2` 中有一个消亡，另一个还没消亡时，则会出现使用已被归还的空间问题

系统提供的隐式拷贝构造函数实施的是浅拷贝（shallow copy）：只拷贝数据成员本身的值。

为了解决上面的问题，可以在类String中显式定义一个拷贝构造函数来实现深拷贝（deep copy），为新对象的指针成员分配指向的空间，把老对象指针成员指向的内容复制过来

```
String::String(const String& s)
{   len = s.len;
    str = new char[len+1];
    strcpy(str,s.str);
}
```

当类定义中包含成员对象时，系统提供的隐式拷贝构造函数会去调用成员对象的拷贝构造函数，但自定义的拷贝构造函数则默认调用成员对象的默认构造函数，而不是成员对象的拷贝构造函数

6.5 类作为模块

模块：

- 从物理上对程序中定义的实体进行分组，是可以单独编写和编译的程序单位。
- 模块化是组织和管理大型程序的一个重要手段。
- 一个模块包含接口和实现两部分：
  - 接口：是指在模块中定义的、可以被其它模块使用的一些程序实体的声明描述。
  - 实现：是指在模块中定义的所有程序实体的具体实现描述。

划分模块的基本准则：

- 内聚性最大：模块内的各实体之间联系紧密。
- 耦合度最小：模块间的各实体之间关联较少。
- 便于程序的设计、理解和维护，能够保证程序的正确性。

过程式程序的模块划分

- 通常是基于子程序，把共同完成某独立功能的或使用相同数据的子程序及相关的实体放在一起构成模块。
- 缺点是模块边界模糊：
  - 一个子程序可能参与几个独立功能。
  - 一个子程序可能使用多个数据集。

6.6 对象与类的进一步讨论

6.6.1 对常量对象的访问——常成员函数

在程序运行的不同时刻，一个对象可能会处于不同的状态：对象的状态由对象的数据成员的值来体现

可以把类中的成员函数分成两类：获取对象状态（不改变数据成员的值）和改变对象状态（改变数据成员的值）

为了防止在一个获取对象状态的成员函数中无意中修改对象数据成员的值，可以把它说明成常成员函数：给函数加上一个 `const` 说明，指出它不能修改数据成员的值。编译器一旦发现在常成员函数中修改数据成员的值，将会报错。

对于有些修改对象状态的常成员函数，编译程序不会指出错误

```
class A
{
    int x;
    char *p;
public:
......
    void f() const
    {
        x = 10; //Error
        p = new char[20]; //Error
        strcpy(p,"ABCD"); //因为没有改变p的值，编译程序认为OK!
    }
};
```

常成员函数还有一个作用：指出对常量对象能实施哪些操作，即，对常量对象只能调用类中的常成员函数。

6.6.2 同类对象之间的数据共享——静态数据成员

若用全局变量来表示共享的数据，则缺乏对数据的保护

在C++中，采用类的**静态成员**来解决。分为：静态数据成员和静态成员函数

1. 静态数据成员

类的静态数据成员对该类的所有对象只有一个拷贝。当通过一个对象改变了静态数据成员的值时，通过同类的其他对象可以看到这个修改

好处：

- 受到类的保护，只有该类的对象才能访问它们，程序的其他地方是不能访问它们的
- 以静态数据成员来实现数据共享可使共享的数据与对象之间联系紧密，使程序便于理解与维护

2. 静态成员函数

- 静态成员函数只能访问类的静态成员
- 静态成员函数没有隐藏的 `this` 参数

静态成员除了通过对象来访问外，也可以直接通过类来访问

6.6.3 提高对象私有数据成员的访问效率——友元

根据数据封装的要求：类中定义的数据成员不能在外界直接访问，必须要通过类中定义的public成员函数来访问。在有些情况下，这种对数据的访问方式效率不高

为了提高在类的外部对类的非public成员的访问效率，在C++中，可以指定某些与一个类密切相关的、又不适合作为该类成员的程序实体能直接访问该类的非public成员，这些程序实体称为该类的友元。

友元需要在类中用friend显式指出，它们可以是：

- 全局函数
- 其它类的所有成员函数
- 其它类的某个成员函数

友元不是一个类的成员。

友元关系具有不对称性。例如：假设B是A的友元，如果没有显式指出A是B的友元，则A不是B的友元。

友元也不具有传递性。例如：假设C是B的友元、B是A的友元，如果没有显式指出C是A的友元，则C不是A的友元。

好处：提高面向对象程序设计的灵活性，是数据保护和数据存取效率之间的一个折中方案。

6.7 操作符重载

6.7.1操作符重载概述

C++允许对已有的操作符进行重载，使得用它们能对自定义类型（类）的对象进行操作。

C++操作符重载的实现途径

操作符重载可通过定义一个函数名为“operator #”（“#”代表某个可重载的操作符）的函数来实现，该函数可以作为：

- 一个类的非静态的成员函数（操作符new和delete除外）。

- 一个全局（友元）函数。

一般情况下，操作符既可以作为全局函数，也可以作为成员函数来重载。在有些情况下，操作符只能作为全局函数或只能作为成员函数来重载

操作符重载的基本原则

- 只能重载C++语言中已有的操作符，不可臆造新的操作符
- 可以重载C++中除下列操作符外的所有操作符：“.”， “.*”，“?:”，“::”，“sizeof”
- 需要遵循已有操作符的语法:
  - 不能改变操作数个数。
  - 原操作符的优先级和结合性不变。
- 尽量遵循已有操作符原来的语义：语言本身没有对此做任何规定，使用者自己把握

6.7.2 操作符重载的基本做法

1. 双目操作符的重载

（1）作为成员函数重载：只需要提供一个参数，其类型为第二个操作数的类型

- 定义格式

```
class <类名>
{   ......
    <返回值类型> operator # (<类型>); //#代表可重载的操作符
};
<返回值类型> <类名>::operator # (<类型> <参数>) { ...... }
```

- 使用格式
  <类名> a;
  <类型> b;
  a # b
  或
  a.operator#(b)

（2）作为全局（友元）函数重载：需要提供两个参数，其中至少应该有一个是类、结构、枚举或它们的引用类型。

- 定义格式

```
<返回值类型> operator #(<类型1> <参数1>,
                        <类型2> <参数2>)
{ …... }
```

- 使用格式
  <类型1> a;
  <类型2> b;
  a # b
  或
  operator#(a,b)

作为全局函数重载双目操作符时，如果操作符重载函数的参数类型为某个类，且在操作符重载函数中需要访问参数类的私有成员，则需要把该操作符重载函数说明成相应类的友元

2. 单目操作符的重载

（1）作为成员函数重载：不需要提供参数

- 定义格式

```
class <类名>
{......
  <返回值类型> operator # ();
};
<返回值类型> <类名>::operator # () { ...... }
```

- 使用格式
  <类名> a;
  #a
  或，
  a.operator#()

（2）作为全局（友元）函数重载：只需要提供一个参数，其类型必须是类、结构、枚举或它们的引用类型

- 定义格式

```
<返回值类型> operator #(<类型> <参数>) { …... }
```

- 使用格式：
  <类型> a;
  #a
  或
  operator#(a)

重载++（--）时，如果没有特殊处理，它们的后置用法和前置用法共用同一个重载函数。
为了能够区分++（--）的前置与后置用法，可以为后置用法再写一个重载函数，该重载函数应有一个额外的int型参数（函数体中可以不使用该参数的值）。

```
class Counter
{       int value;
    public:
        Counter() { value = 0; }
        Counter& operator ++()  //前置的++重载函数
        {   value++;
            return *this;
        }
        const Counter operator ++(int)  //后置的++重载函数
        {   Counter temp=*this; //保存原来的对象
            value++; //写成：++(*this);更好！调用前置的++重载函数
            return temp; //返回原来的对象
        }
};
......
Counter a,b;
++a;  //使用的是不带参数的操作符++重载函数
a++;  //使用的是带int型参数的操作符++重载函数
b = ++a;  //加一之后的a赋值给b
b = a++;  //加一之前的a赋值给b
++(++a);或 (++a)++; //OK，a加2
```

```
++(a++);或 (a++)++; //Error，编译不通过
```

6.7.3 一些特殊操作符的重载

1. 赋值操作符“=”的重载

C++编译程序会为每个类定义一个隐式的赋值操作，其行为是：逐个成员进行赋值操作。

- 对于普通成员，它采用常规的赋值操作。
- 对于成员对象，则调用该成员对象类的赋值操作进行成员对象的赋值操作。

隐式的赋值操作有时不能满足要求

自定义的赋值操作符重载函数不会自动地去进行成员对象的赋值操作，必须要在自定义的赋值操作符重载函数中显式地给出

一般来讲，需要自定义拷贝构造函数的类通常也需要自定义赋值操作符重载函数。
注意：要区别下面两个“=”的不同含义。
A a;
A b=a; //初始化，等价于：A b(a);，调用拷贝构造函数。
……
b = a; //赋值，调用赋值操作符=重载函数。

2. 访问数组元素操作符“[]”的重载

对于由具有线性关系的元素所构成的对象，可通过重载下标操作符“[]”来实现对其元素的访问。

3. 动态存储分配与去分配操作符 new 与 delete 重载

操作符new和delete用于创建和撤销动态对象

- 操作符new有两个功能：为动态对象分配空间、调用对象类的构造函数
- 操作符delete有两个功能：调用对象类的析构函数、释放动态对象的空间

系统提供的new和delete操作所涉及的空间分配和释放是通过系统的堆区管理系统来进行的，它要考虑各种大小的空间分配与释放，对某个类而言，效率常常不高。
可以针对某个类重载操作符new和delete，使得该类能以自己的方式来实现动态对象空间的分配和释放功能。
注意：重载操作符new和delete时，重载的是它们的分配空间和释放空间的功能，不影响对构造函数和析构函数的调用。

（1）重载操作符new

操作符new必须作为静态的成员函数来重载（static说明可以不写），其格式为：`void *operator new(size_t size);`

- 返回类型必须为void *
- 参数size表示对象所需空间的大小，其类型为size_t（unsigned int）

重载的new操作符的使用格式与系统提供的基本相同

重载new时，除了对象空间大小参数以外，也可以带有其它参数，如果重载的new包含其它参数，其使用格式为：`p = new (...) A(...);`

- 第一个 ... 表示提供给new重载函数的其它参数
- 第二个 ... 表示提供给A类构造函数的参数

（2）重载delete

操作符delete也必须作为静态的成员函数来重载（static说明可以不写），其格式为：`void operator delete(void *p, size_t size);`

- 返回类型必须为void。
- 第一个参数类型为void *，指向对象的内存空间。
- 第二个参数可有可无，如果有，则必须是size_t类型。

重载后，操作符delete的使用格式与未重载的相同。

4. 函数调用操作符“()”重载

在C++中，把函数调用也作为一种操作符来看待，操作符为()，操作数为函数名及各个实参，结果为函数返回值`z = f(x,y);`

重载函数调用操作符之后，相应类的对象就可当作函数来使用了。例如：

```
class A
{       int value;
    public:
        A() { value = 0; }
        A(int i) { value = i; }
        int operator () (int x) //函数调用操作符()的重载函数
        { return x+value;
        }
};
......
A a(3); //a是个对象，但可以当函数来使用它
cout << a(10) << endl; //a(10)将会去调用A中的函数调用操作符，重载函数，10作为实参，返回13
```

函数调用操作符重载主要用于具有函数性质的对象（函数对象，functor），该对象通常只有一个操作，它除了具有一般函数的行为外，它还可以拥有状态。

5. 间接类成员访问操作符“->”重载

可以针对某个类重载“->”操作符，这样就可以把该类的对象当指针来用，实现一种智能指针（smart pointers）：

- 对指针对象实施“->”操作能访问它“指向”的另一个对象的成员。例如，下面的B类中重载了操作符“->”：
- B b(&a); //b是个智能指针对象，它指向a
- b->f(); //通过b访问对象a的成员f
- 通过指针对象去访问它指向的对象的成员之前能做一些额外的事情。（智能）

为了完全模拟普通的指针功能，针对智能指针类，还可以重载“*”（对象间接访问）、“[]”、“+”、“-”、“++”、“--”、……等操作符

6. 类型转换操作符重载

类中带一个参数的构造函数可以用作从其它类型到该类的转换，也可以定义从一个类到其它类型的转换。