第三章 类和对象的使用

written by SCNU第一帅----->孔文康

对象初始化

在程序中常常需要对变量赋初始值，即对其进行初始化。
如在定义变量时初始化。

```
int a = 10;
```

在基于对象的程序中，在定义一个对象时，也需要作初始化的工作。

对象代表一个实体，每一个对象都有其确定的属性。
在系统为对象分配内存时，应该同时对其有关数据成员初始化。

直接在类的声明中对其数据进行初始化是错的，例如。

```
class Time {
 int hour = 0;
 int minute = 0;
```

```
 int sec = 0;
}
```

因为类并不是一个实体，而是一种抽象类型，并不占存储空间，显无处容纳数据。

如果一个类的所有成员都是公用的，则可以在定义对象时对数据成员进行初始化。如

```
class Time {
 int hour;
 int minute;
 int sec;
}
Time t1 = {0,0,0};
```

用构造函数实现数据成员初始化

C++为了解决对象初始化的问题，提供了构造函数。

构造函数是一种特殊的成员函数，与其他成员函数不同，不需要用户来调用他，而是在建立对象时自动执行。

构造函数的名字必须与类名相同，不能任意命名。

它不具有任何类型，不返回任何值。

在类内定义构造函数例子如下

```
#include<iostream>
using namespace std;
class Time {
 public:
 Time () {
 hour = 0;
 minute = 0;
```

```
 sec = 0;
 }
  void set_time();
  void show_time();
  private:
  int hour;
  int minute;
  int sec;
}
void Time::set_time() {
  cin>>hour;
  cin>>minute;
  cin>>sec;
}
void Time::show_time() {
  cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
```

在类外定义构造函数例子如下

```
#include<iostream>
using namespace std;
class Time {
  public:
  Time ();
  void set_time();
  void show_time();
  private:
  int hour;
  int minute;
  int sec;
}
Time::Time() {
  hour = 0;
  minute = 0;
  sec = 0;
}
void Time::set_time() {
  cin>>hour;
```

```
 cin>>minute;
 cin>>sec;
}
void Time::show_time() {
 cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
```

可以用一个类对象初始化另一个对象，如

```
Time t1; //建立对象t1,同时调用构造函数t1.Time()
Time t2 = t1; //建立对象t2，并用一个t1初始化t2
```

带参数的构造函数

构造函数首部的一般格式为

构造函数名（类型1 参数1， 类型2 参数2 ……）

定义对象的一般格式为

类名 对象名（实参1，实参2，……）

例子如下

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

```
Time::Time(int h, int m, int s) {
 hour = h;
 minute = m;
 sec = s;
}
Time t1(1,2,3);
```

用参数初始化表对数据成员初始化

C++还提供另一种初始化数据成员的方法——参数初始化表来实现对数据成员的初始化

```
Time::Time(int h, int m, int s):hour(h),minute(m),sec(s) {}
```

即在原来函数首部的末尾加一个冒号，然后列出参数的初始化表。

上面的初始化表表示，用形参h的值初始化hour的值，用形参m的值初始化minute的值，用形参s的值初始化sec的值。

这样可以减少函数体的长度，使结构函数显得精炼简单。

请注意

如果数据成员是数组，则应当在构造函数的函数体中用语句对其赋值，而不能在参数初始化表对其初始化。如

```
class Student {
 public:
 Student(int n,char s,nam[]):num(n),sex(s)
 {
 strcpy(name, nam);
 }
```

```
 private:
 int num;
 char sex;
  char name[20];
 }
 // 可以这样定义stu1
 Student stu1(100,'m','Keivn_kong');
```

构造函数的重载

在一个类中可以定义多个构造函数，以便为对象提供不同的初始化方法。

这些构造函数具有相同的名字，而参数的个数或参数的类型不同，称为构造函数的重载。

```
#include<iostream>
using namespace std;
class Student {
 public:
 Student();
 Student(int n,char s,nam[]):num(n),sex(s)
 {
 strcpy(name, nam);
 }
  private:
  int num;
  char sex;
  char name[20];
 }
 Student::Student() {
  num = 0;
  sex = 's';
  strcpy(name, "test");
 }
 int main () {
  Student stu1; //建立对象stu1不指定实参
  Student stu2(100,'m','Keivn_kong');//建立对象stu2,指定3个实参
```

```
}
```

使用默认参数的构造函数

构造函数中的参数值既可以通过实参传递，也可以指定为某些默认值。

```
class Student {
 public:
 Student(int = 5, char = 'm');
 private:
 int num;
 char sex;
}
Student::Student(int n, int s) {
 num = n;
  sex = s;
}
```

如果用户没有指定实参值，编译系统就使形参的值为默认值。

如果构造函数的全部参数都指定了默认值，则定义对象的时候可以给一个或几个，也可以不给实参。由于不需要实参也可以调用构造函数，因此全部参数都指定了默认值的构造函数也属于默认构造函数。

一个类只能有一个默认构造函数，也就是说，可以不使用参数而调用的构造函数，一个类只能有一个。其道理是为了避免调用时的歧义性。例如

```
Student(); //声明了一个无参构造函数
Student(int = 5, char = 'm');//声明了一个全参指定了默认值的构造函数
```

这样是错误的，因为当你建立对象时

```
Student stu1; //无法确定调用的是哪个构造函数
```

当你定义了全部是默认参数的构造函数后，不能在定义重载构造函数。

例如

```
Student(int = 10,int = 10, int 10);
Student();
Student(int, int);
```

若有如下定义语句

```
Student stu1; //是调用上面第一个还是第二个
Student stu2(15,30);//是调用上面第一个还是第三个
```

因此一般不应同时使用构造函数的重载和默认参数的构造函数。

析构函数

析构函数是一个特殊的成员函数，他的名字就是类名前面加一个~符号。

当对象的生命期结束的时候，会自动执行析构函数。

所以不难推出，析构函数是为了进行数据清理工作的。

析构函数的作用并不是删除对象，而是在撤销对象占用内存之前完成一些清理工作，使这部分内存可以分配给对象重新使用。

析构函数不返回任何值，没有函数类型，也没有函数参数。所以不能被重载。因此一个类可以有多个构造函数，但是只能有一个析构函数。

析构函数的作用不仅限于释放资源方面，还可以被用来执行用户所希望的最后一次使用对象之后所执行的操作。

```
class Student {
 public:
 Student(int = 5, char = 'm');
 ~Student() {
 cout<<"Destructor Called"<<endl;
 }
 private:
 int num;
 char sex;
}
Student::Student(int n, int s) {
  num = n;
  sex = s;
}
```

调用构造函数和析构函数的顺序

在一般情况下，调用析构函数的次序正好与调用构造函数的次序相反。

最先被调用的构造函数，其对应的（同一对象中）析构函数最后被调用，而最后被调用的构造函数，其对应的析构函数最先被调用。

相当于一个栈，先进后出。如

```
Student stu1(0, 'a');
Student stu2(1, 'b');
```

```
3 // 先调用的是stu2的析构函数，再调用stu1的析构函数
```

那么在什么情况下会调用析构函数呢?

- 如果在全局范围内定义了对象(即在所有函数之外)，那么它的构造函数在本文件模块中的所有函数(包括main函数)执行之前调用。但如果一个程序包含多个文件，而在不同的文件中都定义了全局对象，则这些对象的析构函数的执行顺序是不确定的。当main函数执行完毕或调用exit函数时，调用析构函数。
- 如果定义的是局部函数自动对象(例如在函数中定义对象)，则在建立对象时调用其构造函数。如果对象所在的函数被多次调用，则在每次建立对象时都要调用其构造函数。在函数调用结束、对象释放时先调用析构函数。
- 如果在函数中定义静态(static)局部对象，则只在程序第一次调用此函数定义对象时调用构造函数一次，在调用函数结束时，对象并不释放，因此也不调用析构函数。只在main函数结束或调用exit函数结束时，才调用析构函数。

对象数组

对象数组的每一个元素都是同类对象。

```
1 Student stu[50]; //假定声明Student类，定义stu数组，有50个元素
```

在建立数组时，同样要调用构造函数。如果有50个元素，需要调用50次构造函数。

如果构造函数只有一个参数，在定义数组的时候可以直接在等号后面的花括号内提供实参。

```
Student stu[3] = {1,2,3};//3个实参分别传递给3个数组元素的构造函数
```

定义数组时提供的实参个数不能超过数组元素个数。

```
Student stu[3] = {1,2,3, 4};//不合法
```

如果构造函数有多个参数，在定义对象数组时应当怎样实现初始化?

在花括号中分别写出构造函数名并在括号内指定实参。如果构造函数有3个参数，例子如下

```
Student stu[3] = {
 Student(1,2,3), //调用第一个元素的构造函数，向她提供3个实参
 Student(2,3,4), //调用第二个元素的构造函数，向她提供3个实参
 Student(3,4,5), //调用第三个元素的构造函数，向她提供3个实参
};
```

对象指针

指向对象的指针

在建立对象时，编译系统会为每一个对象分配一定的存储空间，以存放其数据成员的值。

一个对象的存储空间的起始地址就是对象的指针。可以定义一个指针变量，用来存放对象的地址，这就是指向对象的指针变量。

假设有一个Time类

```
class Time {
 public:
 int hour;
 int minute;
 int sec;
 void get_time();
};
void Time::get_time() {
 cout<<hour<<":"<<minute<<":"<<sec<<endl;
 }
```

在此基础上，有以下语句

```
Time *pt; //定义pt为指向Time类对象的指针变量
Time t1; //定义t1为Time类对象
pt = &t1; //将t1的起始地址赋给pt
```

这样pt就是指向Time类对象的指针变量，指向对象t1。

定义指向类对象的指针变量的一般形式为

类名 *对象指针名;

可以通过对象指针pt来访问对象和对象的成员，例如

```
*pt //pt所指向的对象，也就是t1
(*pt).hour //pt所指向对象中的hour成员，也即是t1.hour
pt->hour //pt所指向对象中的hour成员，也即是t1.hour
(*pt).get_time() //pt所指向对象中的get_time函数，也即是t1.get_time
pt->get_time() //pt所指向对象中的get_time函数，也即是t1.get_time
```

指向对象成员的指针

对象有地址，存放对象的起始地址的指针变量就是指向对象的指针变量。

对象中的成员也有地址，存放对象成员地址的指针变量就是指向对象成员的指针变量。

1、指向对象数据成员的指针

定义指向对象数据成员的指针变量和定义指向普通变量的指针变量方法相同。

```
int *p1;
p1 = &t1.hour; //将t1的数据成员hour的地址赋值给p1,使p1指向t1.hour
```

2、指向对象数据成员的指针

定义指向对象成员函数的指针变量方法和定义指向普通成员函数的指针变量方法有所不同。

普通成员函数指针的定义方法

类型名（*指针变量名）（参数列表）；

```
void(*p)(); //p是指向void型函数的指针变量
```

可以使p指向一个函数，并通过指针变量调用函数。

```
p = fun; //将fun函数的入口地址赋值给变量p,p就指向了函数fun
(*p)(); //调用fun函数
```

那么，应该怎样定义指向成员函数的指针变量呢？

```
void(Time::*p2)(); //定义p2为指向Time类中公用成员函数的指针变量
```

数据类型名（类名：：*指针变量名）（参数列表）；

只需要把公用成员函数的入口地址赋给一个指向公用成员函数的指针变量即可。

```
p2 = &Time::get_time;
```

指针变量名 = &类名：：成员函数名；

指向当前对象的this指针

不同的对象都调用同一个函数的目标代码。

在每一个成员函数中都包含一个特殊的指针，这个指针的名字是固定的，称为this。

它是指向本类对象的指针，它的值是当前被调用的成员函数所在对象的起始地址。

例如get_time函数

```
cout<<this->hour<<":"<<this->minute<<":"<<this->sec<<endl;
```

当this指向t1的时候，实际上是执行

```
cout<<t1.hour<<":"<<t1.minute<<":"<<t1.sec<<endl;
```

this指针是隐式调用的，本来成员函数get_time的定义是

```
void Time::get_time() {
 cout<<hour<<":"<<minute<<":"<<sec<<endl;
}
```

C++把它处理为

```
void Time::get_time(Time *this) {
 cout<<this->hour<<":"<<this->minute<<":"<<this->sec<<endl;
}

t1.get_time(&t1);
```

公用数据保护

C++虽然采取了很多有效措施（例如private）以增加数据安全性，但是有些数据往往是共享的。人们在不同的场合通过不同的途径访问一个数据对象。有时候在有意和无意之中的误操作会改变有关的数据状况。

既要使数据在一定范围共享，又要保证它不被任意修改，这时候可以把有关数据定义为常量。

常对象

可以在定义对象时加关键字const,指定对象为常对象。常对象必须有初值。

```
Time const t1(1,2,3); //定义t1是常对象
```

这样在t1的生命周期中，对象t1的数据成员的值都不能被修改。

定义常对象的一般形式为

类名 const 对象名[（实参表）]

也可以把const写在最左边

const 类名 对象名[（实参表）]

请注意

如果一个对象被声明为常对象，则通过该对象只能调用它的常成员函数，而不能调用该对象的普通成员函数（除了系统自动调用的隐式的构造函数和析构函数）。常成员函数是常对象唯一的对外接口。

```
Time const t1(1,2,3); //定义t1是常对象
t1.get_time(); //非法，企图调用常对象的普通成员函数
```

那么，怎么样才能引用常对象中的数据成员呢?

很简单，只需要把成员函数声明为const即可。

```
void get_time() const; //将函数声明为const
```

请注意

常成员函数可以访问常对象中的数据成员，但不允许修改常对象中数据成员的值。

有时候一定要修改对象中的某个数据成员的值（类如计数器count），这时候可以把该数据成员声明为mutable。

```
mutable int count;
```

这样就可以用常成员函数修改它的值。

常对象成员

可以将对象的成员声明为const，包括常数据成员和常成员函数。

1、常数据成员

只能通过构造函数的参数初始化表对常数据成员进行初始化，任何其他函数都不能对常数据成员赋值。

```
class Time {
 const int hour; //定义hour为常数据成员
}
```

不能采用在构造函数中对常数据成员赋初值的方法。

```
Time::Time(int h) {
 hour = h; //非法
}
```

因为常数据成员是不能赋值的，如果在类外定义构造函数。

```
Time::Time(int h):hour(h) {} //通过参数初始化表对常数据成员赋值。
```

2、常函数成员

声明常成员函数的一般格式为

类型名 函数名（参数表） const

数据成员	非const的普通成员函数	const成员函数
非const的普通数据成员	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	不允许	可以引用，但不可以改变值

请注意

不要误认为常对象中的成员函数都是常成员函数。常对象保证的是其数据成员是常数据成员。如果在常对象中的成员函数未加const声明，编译系统会将其作为非const成员函数处理。

常成员函数不能调用另一个非const成员函数。

指向对象的常指针

将指针变量声明为const型，这样指针变量始终保持为初值，不能改变。

```
Time t1(1,2,3); //定义对象
Time * const ptr1; //const位置在指针变量名前，指定ptr1是常指针变量
ptr1 = &t1; //ptr1指向对象t1,此后不能改变指向
ptr1 = &t2; //错误，ptr1不能改变指向
```

定义指向对象的常指针变量的一般格式为

类名 * const 指针变量名;

也可以在定义指针变量的时候初始化

```
Time * const ptr1 = &t1;
```

请注意

指向对象的常指针变量的值不能变，即始终指向同一个对象，但可以改变所指对象的值。

指向常对象的指针变量

形参	实参	合法否	改变指针所指向的变量的值
指向非const型变量的指针	非const变量地址	合法	可以
指向非const型变量的指针	const变量的地址	非法	/
指向const型变量的指针	const变量的地址	合法	不可以
指向const型变量的指针	非const变量地址	合法	不可以

请注意

如果一个对象已经被声明为常对象，只能用指向常对象的指针变量指向它。

如果定义了一个指向常对象的指针变量，并使他指向一个非const对象，其指向的对象是不能通过该指针变量进行改变的。

```
Time t1(1,2,3);
const Time *p = &t1;
t1.hour = 2; //合法，t1不是常变量
(*p).hour = 18; //非法，不能通过指针变量改变t1的值
```

如果希望任何情况下t1的值都不能改变，则应把他定义为const型

```
const Time t1(1,2,3);
```

const型数据小结

形式	含义
Time const t1;	t1是常对象，其值在任何情况下不能改变
void Time::fun() const;	fun是Time类中的成员函数，可以引用，但不能修改本类中的数据成员
Time * const p;	p是指向Time类对象的常指针变量，p的值(p的指向)不能变。
const Time *p;	p是指向Time类对象的指针变量， p指向的类的变量不能通过p来改变
const Time &t1 = t;	t1是Time类对象t的引用，二者指向同一存储空间，t的值不能改变。

对象的动态建立和释放

用前面介绍的方法定义对象是静态的，在程序运行的过程中，对象所占的空间是不能随时释放的。

例如在一个函数中定义了一个对象，只有在该函数结束时，该对象才得以释放。但有时候人们希望在需要用到对象时才建立，在不需要时候撤销他，释放它所占的内存空间以供别的数据使用，这样可以提高空间的利用率。

C++可以用new运算符动态分配内存，用delete运算符释放这些内存。

这也适用于对象。

如果已经定义了一个Box类

```
new Box;
```

执行此语句系统开辟了一段内存空间，并在此内存空间中存放一个Box类对象，同时调用该类的构造函数，以使对象初始化。但是此时用户还没发访问这个对象，因为这个对象没有对象名，用户也不知道它的地址。

这种对象称为无名对象，它确实是存在的，但它没有名字。

用new运算符动态分配内存后，将返回一个指向新对象的指针，即所分配的内存空间的起始地址。用户可以获得这个地址，并通过这个地址来访问这个对象。

这样就需要定义一个指向本类的对象的指针变量来存放该地址。

```
Box *pt; //定义一个指向Box类对象的指针变量pt
pt = new Box; //在pt中存放新建对象的起始地址
```

这样在程序中就可以通过pt访问这个新建的对象。

```
cout<<pt->height; //输出该对象的height成员
cout<<pt->volume(); //调用该对象的volume函数，计算并输出体积
```

你还可以对新建立的对象进行初始化。

```
Box *pt = new Box(1,2,3);
```

在执行new运算时，如果内存量不足，无法开辟所需的内存空间。

在目前大多数C++编译器都使new返回一个0指针值(NULL)。而C++标准提出，在执行new出现故障时，就抛出一个异常。当前，不同编译系统对new故障的处理不一样。

在不需要使用new建立的对象时，可以用delete运算符予以释放。

```
delete pt; //释放pt指向的内存空间
```

这就撤销了pt指向的对象。此后程序不能再使用该对象。如果用一个指针变量pt先后指向不同的动态对象，应注意指针变量的当前指向，以免删错对象。

在执行delete运算符时，在释放内存空间之前，自动调用析构函数。

对象的赋值和复制

对象的赋值

如果对一个类定义了两个或多个对象，则这些同类的对象之间可以互相赋值，一个对象的值可以赋给另一个同类的对象。这里所指的对象的值是指对象中所有数据成员的值。

对象之间的赋值也是通过赋值运算符=进行的。

本来赋值运算符只能用来对单个的变量赋值，现在被扩展为两个同类对象之间的赋值，这是通过对赋值运算符的重载实现的。

实际上这个过程是通过成员复制来完成的，即将一个对象的成员值一一复制给另一个对象的对应成员。

对象赋值的一般形式为

对象名1 = 对象名2

请注意

对象名1和对象名2必须属于同一个类。

类的数据成员中不能包括动态分配的数据

```
Student stud1, stud2; //定义两个同类的对象
stud2 = stud1; //将stud1赋给stud2
```

对象的复制

有时候需要用到多个完全相同的对象。在C++中有对象的复制机制。

用一个已有的对象快速地复制出多个完全相同的对象。

```
Box box2(box1);
```

这就是用已有的对象box1去克隆一个新的对象box2。

一般形式为

类名 对象2（对象1）

其实这个是调用了类的默认复制构造函数。

这个函数的形式是这个样子的。

```
Box::Box(const Box & b) {
 height = b.height;
 width = b.width;
 length = b.length;
}
```

复制构造函数也是构造函数，但是他只有一个参数，就是本类的对象，而且采取对象的引用形式。

什么时候使用呢?

```
void fun(Box b) {}

int main () {
 Box box1(1,2,3);
 fun(box1);
 return 0;
}
```

```
Box f () {
 Box box1(1,2,3);
 return box1;
}
int main () {
 Box box2;
 box2 = f();
 return 0;
}
```

静态成员

前面提到如果有N个同类对象，那么每一个对象都分别有自己的数据成员，不同对象的数据成员各有值，互不相干。

但是人们希望有一个或多个数据成员为所有对象所共有，这样可以实现数据共享。

如果想在同类的多个对象之间实现数据共享，也不要用全局变量，可以使用静态的数据成员。

静态数据成员

静态数据成员是一种特殊的数据成员。它以关键字static开头。

```
class Box {
 public:
 int volume();
 private:
 static int height; //把height定义为静态的数据成员
 int width;
 int length;
}
```

静态数据成员在内存中只占一份空间。每个对象都可以引用这个静态数据成员。如果改变它的值，则在各个对象中这个数据成员的值同时改变了。

请注意

之前强调，只声明类而未定义对象，则类的一般数据成员是不占内存空间的。但是静态数据成员不属于某一个对象。只要在类中指定了静态数据成员，即使不定义对象，也为静态数据成员分配空间，它可以被引用。

它不随对象的建立而分配空间，也不随对象的撤销而释放。

程序编译时被分配，程序结束才释放。

静态数据成员可以初始化，但只能再类体外进行初始化。

```
int Box::height = 10; //表示对Box类中的数据成员初始化
```

静态成员函数

成员函数也可以定义为静态的，在类中声明函数的前面加static就成了静态成员函数。

```
static int volume();
```

跟静态数据成员一样，静态成员函数是类的一部分，而不是对象的一部分。如果要在类外调用公用的静态成员函数，要用类名和域运算符：：。

```
Box::volume();
// 实际上也允许通过对象名调用静态成员函数
a.volume();
// 但是这并不意味着此函数属于a，而只是用a的类型而已。
```

非静态成员函数有this指针，而静态成员函数没有this指针。

由此决定了静态成员函数不能访问本类中的非静态成员。

友元

友元可以访问与其有好友关系的类中的私有成员。

友元包括友元函数和友元类。

友元函数

如果在本类以外的其他地方定义了一个函数(这个函数可以是不属于任何类的非成员函数，也可以是其他类的成员函数)。

在类体中用friend对其进行声明，此函数就称为本类的友元函数。

友元函数可以访问这个类中的私有成员。

1、将普通函数声明为友元函数

```
#include<iostream>
using namespace std;
class Time {
 public:
 Time(int, int, int);
 friend void display(Time &); //声明display函数为Time类的友元函数
 private:
 int hour;
 int minute;
  int sec;
 }
 Time::Time(int h, int m, int s) {
  hour = h;
  minute = m;
  sec = s;
 }
 // 这是普通函数，形参t是Time类的引用对象
 void display(Time & t) {
  cout<<t.hour<<":"<<t.minute<<":"<<t.sec<<endl;
 }
 int main () {
  Time t1(10, 13, 56);
  display(t1);
  return 0;
 }
```

2、友元成员函数

```
#include<iostream>
using namespace std;
class Date;
class Time {
 public:
 Time(int, int, int);
 void display(Date &); //声明display函数为Time类的友元函数
```

```
 private:
 int hour;
  int minute;
  int sec;
}
class Date {
 public:
 Date(int, int, int);
 friend void Time::display(Date &);
 private:
 int month;
 int day;
 int year;
}
```

友元类

友元类B中的所有函数都是A的友元函数。

在类体A的定义体中用以下语句声明类B为其友元类

friend B;