第6章初始化和清除

本章内容

- 初始化和清除运用不当，都可能在程序中造成严重的问题。
- 在C++中，初始化和清除的概念是简化库使用的关键所在。

构造函数与析构函数

- 构造函数和析构函数是在类体中说明的两种特殊的成员函数。
- 构造函数是在创建对象时，使用给定的值来将对象初始化。
- 析构函数的功能正好相反，是在系统释放对象前，对对象做一些善后工作。
- 一般来说，析构函数不能显式调用，因为显式调用析构函数，可以将对象析构，但对象所占用的内存仍然没有被系统收回(也就是说，对象所占用的那块内存还没有被标记为空闲)，如果不做进一步处理，这样就会造成内存泄漏。

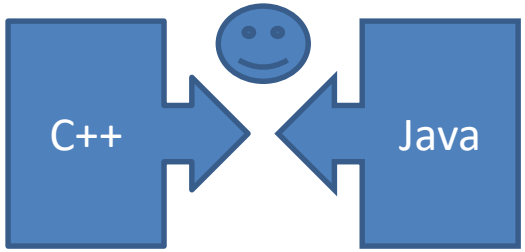
用构造函数确保初始化

- 类的设计者可以通过提供一个叫做构造函数的特殊函数保证每个对象都被初始化；
- 如果一个类带有构造函数，编译器在创建对象时就自动调用这一函数，这一切在程序员使用他们的对象之前就已经完成了；
- 传到构造函数的第一个（secret）参数是this指针；

用构造函数确保初始化

- 一个带构造函数的类的示例：

```
class X{
        int i;
   public:
        X();
}
void f(){
        X a;
}
```



- 构造函数没有返回值
- 在C++中，定义一个变量和初始化该变量是合二为一的。

构造函数几点说明

- 构造函数的函数名必须与类名相同。构造函数的主要作用是完成初始化对象的数据成员以及其它的初始化工作。
- 在定义构造函数时，不能指定函数返回值的类型，也不能指定为void类型。
- 一个类可以定义若干个构造函数。当定义多个构造函数时，必须满足函数重载的原则

构造函数几点说明

- 构造函数可以指定参数的缺省值
- 若定义的类要说明该类的对象时，构造函数必须是公有的成员函数。如果定义的类仅用于派生其它类时，则可将构造函数定义为保护的成员函数。
- 由于构造函数属于类的成员函数，它对私有数据成员、保护的数据成员和公有的数据成员均能进行初始化。

构造函数几点说明

- 对局部对象，静态对象，全局对象的初始化
  - 对于局部对象，每次定义对象时，都要调用构造函数。
  - 对于静态对象，是在首次定义对象时，调用构造函数的，且由于对象一直存在，只调用一次构造函数。
  - 对于全局对象，是在main函数执行之前调用构造函数的

```
class A{
        float x,y;
public:
  A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
  A(){ x=0; y=0; cout<<"初始化静态局部对象\n";}
  A(float a){ x=a; y=0; cout<<"初始化全局对象\n"; }
 void Print(void){ cout<<x<<'\t'<<y<<endl; }
};
A a0(100.0);//定义全局对象
void f(void)
{ cout<<" 进入f()函数\n";A a2(1,2);
   static A a3; //初始化局部静态对象
 }
void main(void)
{   cout<<"进入main函数\n";
   A a1(3.0, 7.0);//定义局部自动对象
   f();   f();  }
```

初始化全局对象
进入main函数
初始化自动局部对象
进入f()函数
初始化自动局部对象
初始化局部静态变量
进入f()函数
初始化自动局部对象

有goto语句的情况

- 分析下面X x的初始化和使用的情况

  NoJump.cpp：

```
int i,j;
......
if(i>0)
        goto L1;
X x;
......
L1：
        x.func();
return 0;
```

缺省的构造函数

- 在定义类时，若没有定义类的构造函数，则编译器自动产生一个缺省的构造函数，其格式为：
- ```
  className::className() { }
  ```
- 缺省的构造函数并不对所产生对象的数据成员赋初值；即新产生对象的数据成员的值是不确定的。
  ```

缺省的构造函数

- 在定义类时，只要显式定义了一个类的构造函数，则编译器就不产生缺省的构造函数
- 所有的对象在定义时，必须调用构造函数

用析构函数确保清除

- 清除一个对象就像初始化一样重要，在C++中，通过析构来保证清除的执行。
- 在对象的生命期结束时，释放系统为对象所分配的空间，即要撤消一个对象。
- 对于定义在代码块中变量
  - 在被声明的位置实例化(调用构造函数)
  - 离开作用域，即离开所定义的块时清除(调用析构函数)
- 例：Constructor1.cpp

析构函数的特点

- 析构函数是成员函数，函数体可写在类体内，也可写在类体外。
- 析构函数是一个特殊的成员函数，函数名必须与类名相同，并在其前面加上字符“~”，以便和构造函数名相区别。
- 析构函数不能带有任何参数，不能有返回值，不指定函数类型。

析构函数的特点

- 一个类中，只能定义一个析构函数，析构函数不允许重载。
- 析构函数是在撤消对象时由系统自动调用的。
- 在程序的执行过程中，当遇到某一对象的生存期结束时，系统自动调用析构函数，然后再收回为对象分配的存储空间。

```
class A{
        float x,y;
public:
  A(float a,float b)
 {      x=a;y=b;cout<<"调用非缺省的构造函数\n";}
  A()  { x=0; y=0; cout<<"调用缺省的构造函数\n" ;}
  ~A() {      cout<<"调用析构函数\n";}
   void Print(void) {   cout<<x<<'\t'<<y<<endl; }
};
void main(void)
{       A  a1;
        A  a2(3.0,30.0);
        cout<<"退出主函数\n";
}
```

调用缺省的构造函数

调用非缺省的构造函数

退出主函数

调用析构函数

调用析构函数

使用new运算符

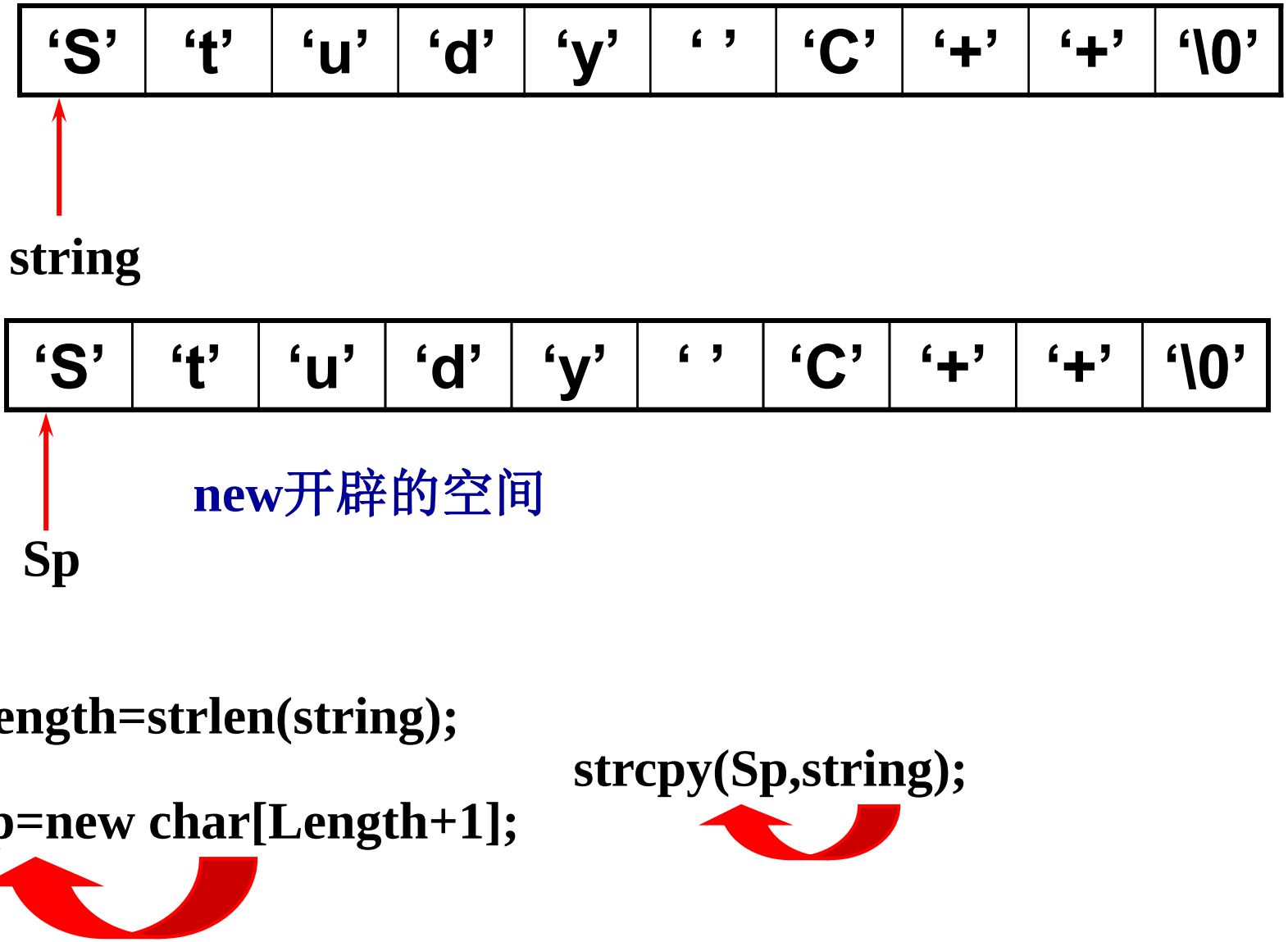
- 在程序的执行过程中，对象如果用new运算符开辟了空间，则在类中应该定义一个析构函数，并在析构函数中使用delete删除由new分配的内存空间。因为在撤消对象时，系统自动收回为对象所分配的存储空间，而不能自动收回由new分配的动态存储空间。

```
class Str{
        char *Sp;    int Length;
public:
        Str(char *string)
        {   if(string){        Length=strlen(string);
                        Sp=new char[Length+1];
                        strcpy(Sp,string);
                    }
            else      Sp=0;
        }
        void Show(void){  cout<<Sp<<endl; }
        ~Str(){  if(Sp)       delete []Sp; }
};
void main(void)
{       Str s1("Study C++");
        s1.Show();
}
```

在构造函数中将成员数据指针指向动态开辟的内存

用初值为开辟的内存赋值

析构函数，当释放对象时收回用new开辟的空间


‘S’ ‘t’ ‘u’ ‘d’ ‘y’ ‘ ’ ‘C’ ‘+’ ‘+’ ‘\0’
string
‘S’ ‘t’ ‘u’ ‘d’ ‘y’ ‘ ’ ‘C’ ‘+’ ‘+’ ‘\0’
new开辟的空间
Sp
Length=strlen(string);
strcpy(Sp,string);
Sp=new char[Length+1];

```
class  A{
        float   x,y;
public:
 A(float a, float b){   x=a;    y=b;    cout<<"调用了构造函数\n";}
 void  Print(void){  cout<<x<<'\t'<<y<<endl;  }
 ~A()  {  cout<<"调用了析构函数\n";  }
};
void main(void)
{  cout<<"进入main()函数\n";
   A   *pa1;
   pa1=new  A(3.0, 5.0);//调用构造函数
   pa1->Print();
   delete  pa1;  //调用析构函数
   cout<<"退出main()函数\n";
}
```

```
进入main()函数
调用了构造函数
 3        5
调用了析构函数
退出main()函数
```

构造与析构函数的调用

- 对于全局定义的对象（在函数外定义的对象），在程序开始执行时，调用构造函数；到程序结束时，调用析构函数。
- 对于局部定义的对象（在函数内定义的对象），当程序执行到定义对象的地方时，调用构造函数；在退出对象的作用域时，调用析构函数
- 用static定义的局部对象，在首次到达对象的定义时调用构造函数；到程序结束时，调用析构函数

构造与析构函数的调用

- 对于用new运算符动态生成的对象，在产生对象时调用构造函数，只有使用delete运算符来释放对象时，才调用析构函数。若不使用delete来撤消动态生成的对象，程序结束时，对象仍存在，并占用相应的存储空间，即系统不能自动地调用析构函数来撤消动态生成的对象。

```
class  A{
        float   x,y;
public:
  A(float a, float b){x=a;y=b;cout<<"初始化自动局部对象\n";}
  A(){  x=0;  y=0;  cout<<"初始化静态局部对象\n";}
  A(float  a){  x=a;  y=0;  cout<<"初始化全局对象\n"; }
 ~A(){  cout<<“调用析构函数” <<endl;  }
};
A  a0(100.0);//定义全局对象
void f(void)
{  cout<<" 进入f()函数\n";
   A  ab(10.0,  20.0);//定义局部自动对象
   static  A  a3;  //初始化局部静态对象
}
void main(void)
{   cout<<"进入main函数\n";
f();    f();   }
```

初始化全局对象
进入main函数
进入f()函数
初始化自动局部对象
初始化静态局部对象
调用析构函数
进入f()函数
初始化自动局部对象
调用析构函数
调用析构函数
调用析构函数

缺省的析构函数

- 若在类的定义中没有显式地定义析构函数时，则编译器自动地产生一个缺省的析构函数，其格式为：
- ClassName::~ClassName() { };
- 任何对象都必须有构造函数和析构函数，但在撤消对象时，要释放对象的数据成员用new运算符分配的动态空间时，必须显式地定义析构函数。

析构函数的调用

- 对对象成员的构造函数的调用顺序取决于这些对象成员在类中说明的顺序，与它们在成员初始化列表中的顺序无关。
- 当建立类ClassName的对象时，先调用各个对象成员的构造函数，初始化相应的对象成员，然后才执行类ClassName的构造函数，初始化类ClassName中的其它成员。析构函数的调用顺序与构造函数正好相反。

```
class A{
        float  x;
public:
    A(int a){  x=a;  cout<<“调用了A的构造函数\n”;}
   ~A(){cout<<“调用了A的析构函数\n”;}
};
class B{
        float  y;
public:
    B(int a){  y=a;  cout<<“调用了B的构造函数\n”;}
   ~B(){cout<<“调用了B的析构函数\n”;}
};
class C{
        float z; B  b1;  A a1;
public:
    C(int a,int b,int c): a1(a),b1(b){z=c;cout<<“调用了C的构造函数\n”;}
   ~C(){cout<<“调用了C的析构函数\n”;}
};
void main(void)
{   C  c1(1,2,3);   }
```

调用了B的构造函数

调用了A的构造函数

调用了C的构造函数

调用了C的析构函数

调用了A的析构函数

调用了B的析构函数

带有构造和析构函数的Stash

- C06:Stash2.h
- C06:Stash2.cpp
- C06:Stash2Test.cpp
- 取代了initialize和cleanup方法避免了类库使用者的疏忽

带有构造函数和析构函数的Stack

- C06:Stack3.h
- C06:Stack3.cpp
- C06:Stack3Test.cpp

数组的初始化

- C++
  - int a[5] = {1,2,3,4,5};
  - Int a[5];
  - int a[] = {1,2,3,4,5};
    - sizeof a / sizeof *a
  - int a[] = new int[5];
- Java
  - int a[] = {1,2,3,4,5};
  - int a[] = new int[5];

数组的访问

- C++

```
for(int i=0; i < sizeof a / sizeof *a; i++)
    cout << c[i];
```

- Java

```
for(int element : c)
    System.out.print(element );
```

带构造函数的结构或类的数组

- Multiarg.cpp
  - 只要有构造函数，就必须通过构造函数完成初始化

```
11  class Z {
12    int i, j;
13  public:
14    Z(int ii, int jj);
15    void print();
16  };
17
18  Z::Z(int ii, int jj) {
19    i = ii;
20    j = jj;
21  }
22
23  void Z::print() {
24    cout << "i = " << i << ", j = " << j << endl;
25  }
26
27  int main() {
28    Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
29    for(int i = 0; i < sizeof zz / sizeof *zz; i++)
30      zz[i].print();
31  } ///:~
32
```

一些思考

- 为什么例子中的构造函数和析构函数都是公有的？
- 如果私有会怎样？
- 利用私有后的特性能做什么？

单例Singlton设计模式

- 目的是使得类的一个对象成为系统中的唯一实例。
- 《设计模式》（艾迪生维斯理, 1994）：“保证一个类仅有一个实例，并提供一个访问它的全局访问点。”

- 定义全局变量可以吗？

单例Singlton设计模式

- 1. 单例模式的类只提供私有的构造函数
- 2. 类定义中含有一个该类的静态私有对象
- 3. 该类提供了一个静态的公有的函数用于创建或获取它本身的静态私有对象

单例Singlton设计模式

```
1   // version 1.0
2   public class Singleton {
3       private static Singleton singleton = null;
4       private Singleton() {  }
5       public static Singleton getInstance() {
6           if (singleton== null) {
7               singleton= new Singleton();
8           }
9           return singleton;
10      }
11  }
```

```
public class SingletonPatternDemo {
   public static void main(String[] args) {
      //不合法的构造函数
      //编译时错误：构造函数 Singleton() 是不可见的
      //Singleton object = new Singleton();

      //获取唯一可用的对象
        Singleton object = Singleton.getInstance();
   }
}
```

单例SingIton设计模式

- 私有（private）的构造函数，表明这个类是不可能形成实例了。这主要是怕这个类会有多个实例。
- 即然这个类是不可能形成实例，那么，我们需要一个静态的方式让其形成实例：getInstance()。注意这个方法是在new自己，因为其可以访问私有的构造函数，所以他是可以保证实例被创建出来的。
- 在getInstance()中，先做判断是否已形成实例，如果已形成则直接返回，否则创建实例。
- 所形成的实例保存在自己类中的私有成员中。
- 取实例时，只需要使用Singleton.getInstance()就行了。

单例Singlton设计模式

- 单例模式的优点:
  - 1. 在内存中只有一个对象,节省内存空间
  - 2. 避免频繁的创建销毁对象,可以提高性能
  - 3. 避免对共享资源的多重占用
  - 4. 可以全局访问

- 单例模式的适用场景:
  - 1. 需要频繁实例化然后销毁的对象
  - 2. 创建对象耗时过多或者耗资源过多,但又经常用到的对象
  - 3. 有状态的工具类对象
  - 4. 频繁访问数据库或文件的对象
  - 5. 其他要求只有一个对象的场景

- public enum Singleton{
- INSTANCE;
- }

```
public class Employee {
   private int id;
   private String name;
   public Employee() {}
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName( String name ) {
      this.name = name;
   }
}
```

```
public enum Employee { // changed "class" to "enum"

   INSTANCE; // added name of the (single) instance

   private int id;
   private String name;
   Employee() {} // removed "public"
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName( String name ) {
      this.name = name;
   }
}
```