# 2 类和对象

## 2.1 面向对象程序设计（OOP）

### 2.1.1 什么是OOP?

- OOP，与其说是一种程序设计技巧，倒不如说是一种程序设计思想。这种思想来源于人类面对世间万物最朴素，最自然的感觉、想法和观点。只是直到OOP被做为一种程序设计方法被正式提出，这种人类与生俱来的智慧才真正影响到计算机程序设计的工业化进程
- 大型面向对象程序的基本思想就是把大型软件（常常由数以百万计的代码行组成）看成是一个由对象组成的社会
    - 对象拥有足够的智能，能够理解从其它对象接收到的信息，并以适当的行为对此作出反应
    - 对象能够从高层对象继承属性和行为，并允许低层对象从自己继承属性和行为
- 拥有相同属性和行为的对象被分成一组，形成一个特定的类
- 除了拥有一个以前所定义的类的所有属性和行为外，另外还有一些额外的、更加具体的属性和行为，那么它们便被认为是以前所定义的那个类的子类

### 2.1.2 为什么要OOP?

- OOP已成为几乎所有图形用户界面（GUI）编程的首选。OOP在这方面的能力是如此出众，以至于人们在那些并不直接支持OOP的语言（例如C）中也创建了一些模拟OOP的软件结构，以便进行GUI编程（例如GNOME/GTK+工具）
- OOP在数据库和网络编程方面也逐渐展现出强大的威力
- 得益于数据抽象、代码重用等OOP所固有的特征，软件开发的效率获得极大地提升，成本大幅降低，进一步加速了计算机软件行业的工业化进程
- OOP更适合于开发大型软件，相比于分而治之的结构化程序设计，OOP更加强调大处着眼的统一与整合，在软件工程领域发挥着无可替代的作用

### 2.1.3 怎样OOP?

- 除了掌握语法之外，必须精通每种语言中所蕴藏的，真正的OOP基石：封装、继承和多态
- 精通OOP的必由之路
    - 精通一种“元”语言，例如UML（Unified Modeling Language，统一建模语言），它允许我们在概念层次上可视化地描述我们的设计方案
    - 学习设计模式。在多年的实践中，人们已经总结出大量的设计模式，它们已经成为许多子问题的标准解决方案

## 2.2 类的基本语法

### 2.2.1 类的定义

```
class 类名 {
};
```

例如：

```
class User {
};
```

### 2.2.2 成员变量

```
class 类名 {
    类型 变量名;
};
```

例如：

```
class User {
    ...
    string name;
    int age;
    ...
};
```

### 2.2.3 成员函数

```
class 类名 {
    返回类型 函数名(形参表) {
        函数体;
    }
};
```

例如：

```
class User {
    ...
    void print(void) {
        cout << name << ", " << age << endl;
    }
    ...
};
```

也可将声明与实现分开：

```
class 类名 {
    返回类型 函数名(形参表);
};
```

```
返回类型 类名::函数名(形参表) {
    函数体;
}
```

例如：

```
class User {
    ...
    void print(void);
    ...
};
```

```
void User::print(void) {
    cout << name << ", " << age << endl;
}
```

## 2.2.4 访问控制

- public：公有成员，谁都可以访问
- private：私有成员，只有自己能访问
- protected：保护成员，只有自己和子类可以访问

例如：

```
class User {
public:
    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};
```

| 访控限定符 | 访控属性 | 自己 | 子类 | 外部 |
| --- | --- | --- | --- | --- |
| public | 公有成员 | Yes | Yes | Yes |
| protected | 保护成员 | Yes | Yes | No |
| private | 私有成员 | Yes | No | No |

注意：类的缺省访问控制属性为私有，而结构的缺省访问控制属性为公有。

## 2.2.5 构造函数

```
class 类名 {
    类名(形参表) {
        函数体;
    }
};
```

例如：

```
class User {
public:
    User(void) {
```

```
        name = "Anonymous";
        age = -1;
    }

    User(string name, int age) {
        this->name = name;
        this->age = age;
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};
```

## 2.2.6 对象的创建与销毁

### 2.2.6.1 在栈中创建单个对象

```
类名 对象名;
类名 对象名(实参表);
```

例如:

```
User user;
User user("Zaphod", 49);
```

另一种写法:

```
类名 对象名 = 类名();
类名 对象名 = 类名(实参表);
```

例如:

```
User user = User();
User user = User("Zaphod", 49);
```

### 2.2.6.2 在栈中创建对象数组

```
类名 对象数组名[元素个数];
类名 对象数组名[元素个数] = {类名(实参表), 类名(实参表), ...};
类名 对象数组名[] = {类名(实参表), 类名(实参表), ...};
```

例如:

```
User users[3];
User users[4] = {User("Zaphod", 49), User("Antony", 37), User("Stephen", 35)};
User users[] = {User("Zaphod", 49), User("Antony", 37), User("Stephen", 35)};
```

### 2.2.6.3 在堆中创建/销毁单个对象

```
类名* 对象指针 = new 类名;
类名* 对象指针 = new 类名();
类名* 对象指针 = new 类名(实参表);

delete 对象指针;
```

例如：

```
User* user = new User;
User* user = new User();
User* user = new User("Zaphod", 49);

delete user;
```

### 2.2.6.4 在堆中创建/销毁对象数组

```
类名* 对象指针 = new 类名[元素个数];
类名* 对象指针 = new 类名[元素个数] {类名(实参表), 类名(实参表), ...}; // >= C++11
类名* 对象指针 = new 类名[] {类名(实参表), 类名(实参表), ...}; // >= C++11

delete[] 对象指针;
```

例如：

```
User* users = new User[3];
User* users = new User[4] {User("Zaphod", 49), User("Antony", 37),
User("Stephen", 35)};
User* users = new User[] {User("Zaphod", 49), User("Antony", 37),
User("Stephen", 35)};

delete[] users;
```

```
// clsobj.cpp

// 类的定义、声明、实现和实例化

#include <iostream>

using namespace std;

// 类的定义
class User {
public:
    User(void) {
        name = "Anonymous";
        age = -1;
    }

    User(string name, int age) {
        this->name = name;
        this->age = age;
```

```
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};
/*
// 类的声明
class User {
public:
    User(void);
    User(string name, int age);
    void print(void);

private:
    string name;
    int age;
};

// 类的实现

User::User(void) {
    name = "Anonymous";
    age = -1;
}

User::User(string name, int age) {
    this->name = name;
    this->age = age;
}

void User::print(void) {
    cout << name << ", " << age << endl;
}
*/
// 类的实例化
int main(void) {
    // 在栈中创建单个对象

    User user1;
    user1.print();

    User user2("Zaphod", 49);
    user2.print();

    User user3 = User();
    user3.print();

    User user4 = User("Antony", 37);
    user4.print();

    // 在栈中创建对象数组
```

```
76
77      User users1[3];
78      for (int i = 0; i < 3; ++i)
79          users1[i].print();
80
81      User users2[4] = {
82          User("Zaphod", 49),
83          User("Antony", 37),
84          User("Stephen", 35)};
85      for (int i = 0; i < 4; ++i)
86          users2[i].print();
87
88      User users3[] = {
89          User("Zaphod", 49),
90          User("Antony", 37),
91          User("Stephen", 35)};
92      for (int i = 0; i < 3; ++i)
93          users3[i].print();
94
95      // 在堆中创建/销毁单个对象
96
97      User* user5 = new User;
98      user5->print();
99      delete user5;
100
101     User* user6 = new User();
102     user6->print();
103     delete user6;
104
105     User* user7 = new User("Zaphod", 49);
106     user7->print ();
107     delete user7;
108
109     // 在堆中创建/销毁对象数组
110
111     User* users4 = new User[3];
112     for (int i = 0; i < 3; ++i)
113         users4[i].print();
114     delete[] users4;
115
116     User* users5 = new User[4] {
117         User("Zaphod", 49),
118         User("Antony", 37),
119         User("Stephen", 35)};
120     for (int i = 0; i < 4; ++i)
121         users5[i].print();
122     delete[] users5;
123
124     User* users6 = new User[] {
125         User("Zaphod", 49),
126         User("Antony", 37),
127         User("Stephen", 35)};
128     for (int i = 0; i < 3; ++i)
129         users6[i].print();
130     delete[] users6;
131
```

```
    return 0;
}
```

## 2.2.7 将类的声明与实现分别放在独立的文件中

```
// user.h

// 声明User类

#pragma once

#include <string>

using namespace std;

class User {
public:
    User(void);
    User(string name, int age);
    void print(void);

private:
    string name;
    int age;
};
```

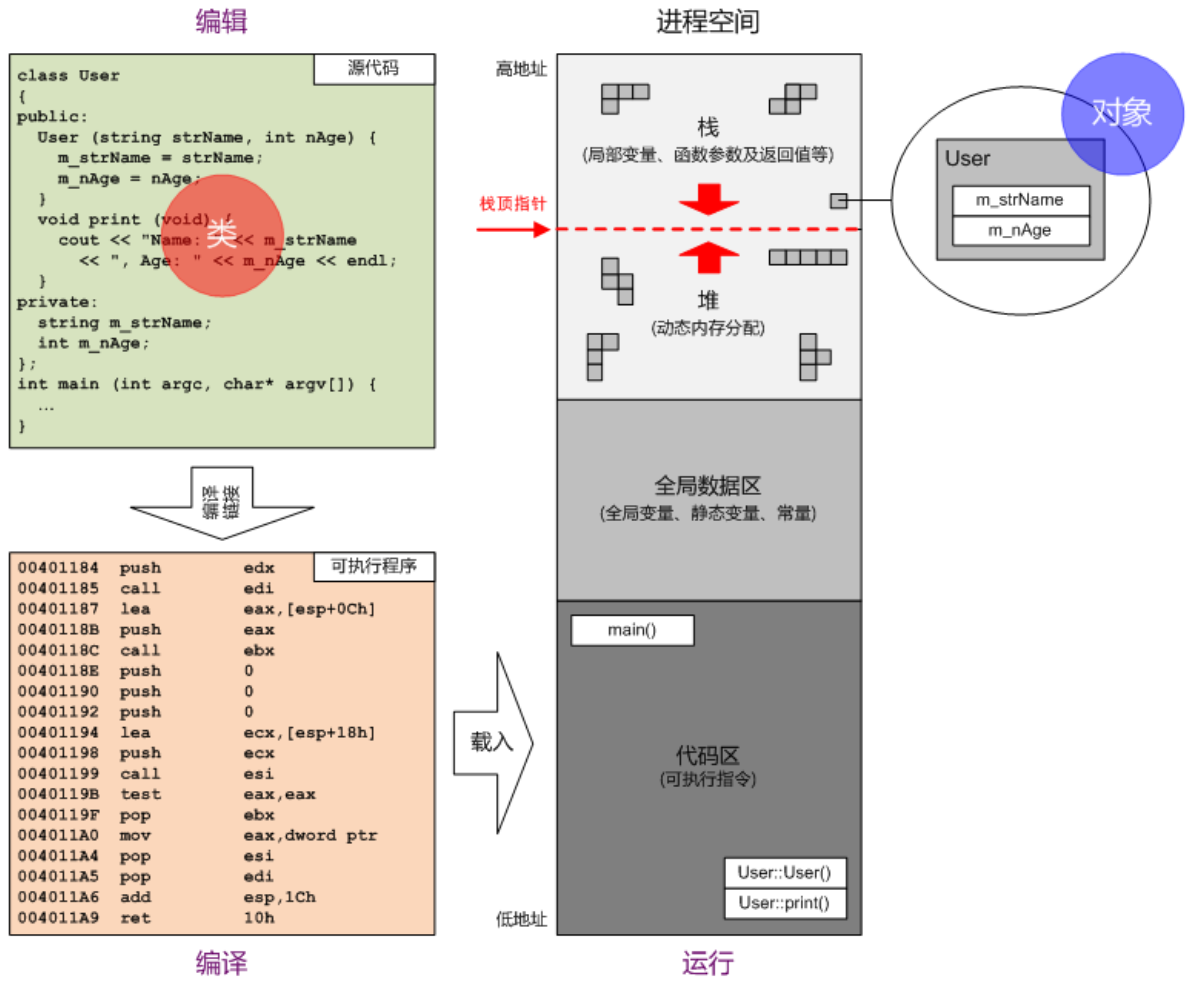
```
// user.cpp

// 实现User类

#include <iostream>

#include "user.h"

using namespace std;

User::User(void) {
    name = "Anonymous";
    age = -1;
}

User::User(string name, int age) {
    this->name = name;
    this->age = age;
}

void User::print(void) {
    cout << name << ", " << age << endl;
}
```

```
// main.cpp

// 使用User类
```

```
4
5  #include "user.h"
6
7  int main(void) {
8      // 在栈中创建单个对象
9
10     User user1;
11     user1.print();
12
13     User user2("Zaphod", 49);
14     user2.print();
15
16     User user3 = User();
17     user3.print();
18
19     User user4 = User("Antony", 37);
20     user4.print();
21
22     // 在栈中创建对象数组
23
24     User users1[3];
25     for (int i = 0; i < 3; ++i)
26         users1[i].print();
27
28     User users2[4] = {
29         User("Zaphod", 49),
30         User("Antony", 37),
31         User("Stephen", 35)};
32     for (int i = 0; i < 4; ++i)
33         users2[i].print();
34
35     User users3[] = {
36         User("Zaphod", 49),
37         User("Antony", 37),
38         User("Stephen", 35)};
39     for (int i = 0; i < 3; ++i)
40         users3[i].print();
41
42     // 在堆中创建/销毁单个对象
43
44     User* user5 = new User;
45     user5->print();
46     delete user5;
47
48     User* user6 = new User();
49     user6->print();
50     delete user6;
51
52     User* user7 = new User("Zaphod", 49);
53     user7->print ();
54     delete user7;
55
56     // 在堆中创建/销毁对象数组
57
58     User* users4 = new User[3];
59     for (int i = 0; i < 3; ++i)
```

```
60          users4[i].print();
61      delete[] users4;
62
63      User* users5 = new User[4] {
64          User("Zaphod", 49),
65          User("Antony", 37),
66          User("Stephen", 35)};
67      for (int i = 0; i < 4; ++i)
68          users5[i].print();
69      delete[] users5;
70
71      User* users6 = new User[] {
72          User("Zaphod", 49),
73          User("Antony", 37),
74          User("Stephen", 35)};
75      for (int i = 0; i < 3; ++i)
76          users6[i].print();
77      delete[] users6;
78
79      return 0;
80  }
```

## 2.2.8 类和对象的关系



## 2.3 构造函数

### 2.3.1 构造函数可以重载

```cpp
// consover.cpp

// 构造函数可以重载

#include <iostream>

using namespace std;

class User {
public:
    // 无参构造函数，又名缺省构造函数或默认构造函数
    User(void) {}

    User(string name) {
        this->name = name;
    }

    User(int age) {
        this->age = age;
    }

    User(string name, int age) {
        this->name = name;
        this->age = age;
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main(void) {
    // 在栈中创建对象，未显式赋初值的基本类型成
    // 员变量为随机数，类类型成员变量被缺省构造

    // User user1(); // 编译错误，被误解为函数声明
    User user1;
    user1.print();

    User user2("Stephen");
    user2.print();

    User user3(37);
    user3.print();

    User user4("Zaphod", 49);
    user4.print();
```

```
52
53      // 在堆中创建对象，未显式赋初值的基本类型成
54      // 员变量为零值，类类型成员变量被缺省构造
55
56      User* user5 = new User();
57      user5->print();
58      delete user5;
59
60      User* user6 = new User;
61      user6->print();
62      delete user6;
63
64      User* user7 = new User("Stephen");
65      user7->print();
66      delete user7;
67
68      User* user8 = new User(37);
69      user8->print();
70      delete user8;
71
72      User* user9 = new User("Zaphod", 49);
73      user9->print();
74      delete user9;
75
76      return 0;
77  }
```

## 2.3.2 缺省构造函数

- 缺省构造函数亦称无参构造函数，但其未必真的没有任何参数。为一个有参构造函数的每个参数都提供一个缺省值，同样可以达到无参构造函数的效果

```
1  class UserGroup {
2  public:
3      UserGroup(User* users = NULL, int priority = 10) {
4          this->users = users;
5          this->priority = priority;
6      }
7
8  private:
9      User* users;
10     int priority;
11 };
```

- 对于没有定义任何构造函数的类，系统会为其提供一个缺省构造函数

```
1  // defcons.cpp
2
3  // 若未定义构造函数，则系统会提供一个缺省构造函数
4
5  #include <iostream>
6
7  using namespace std;
8
```

```
class User {
public:
    // 未定义构造函数，系统提供缺省构造函数

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main(void) {
    // 在栈中缺省构造单个对象或对象数组，基本类型成员变量
    // 不做初始化，类类型成员变量被缺省构造

    User user1;
    user1.print();

    User users1[3];
    for (int i = 0; i < 3; ++i)
        users1[i].print();

    // 在堆中缺省构造单个对象或对象数组，基本类型成员变量
    // 被初始化为适当类型的零值，类类型成员变量被缺省构造

    User* user2 = new User;
    user2->print();
    delete user2;

    User* users2 = new User[3];
    for (int i = 0; i < 3; ++i)
        users2[i].print();
    delete[] users2;

    return 0;
}
```

  - 在栈中缺省构造单个对象或对象数组，基本类型成员变量不做初始化，类类型成员变量被缺省构造
  - 在堆中缺省构造单个对象或对象数组，基本类型成员变量被初始化为适当类型的零值，类类型成员变量被缺省构造
- 对于已经定义至少一个构造函数的类，系统不会为其提供缺省构造函数

```
// nodefcons.cpp

// 若已定义构造函数，则系统不再提供缺省构造函数

#include <iostream>

using namespace std;

class User {
```

```
public:
    // 已定义构造函数，系统不提供缺省构造函数

    User(string name, int age) {
        this->name = name;
        this->age = age;
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main(void) {
    User user1; // 编译错误，未定义缺省构造函数
    user1.print();

    User users1[3]; // 编译错误，未定义缺省构造函数
    for (int i = 0; i < 3; ++i)
        users1[i].print();

    User* user2 = new User; // 编译错误，未定义缺省构造函数
    user2->print();
    delete user2;

    User* users2 = new User[3]; // 编译错误，未定义缺省构造函数
    for (int i = 0; i < 3; ++i)
        users2[i].print();
    delete[] users2;

    return 0;
}
```

- 有时必须自己定义缺省构造函数，即使它什么也不做，尤其是在使用数组或容器的时候。某些基于早期标准的编译器，可能不支持对对象数组的初始化语法

```
// hastodef.cpp

// 有时必须自己定义缺省构造函数，即使它什么也不做，尤其是在使用数组或容
// 器的时候。某些基于早期标准的编译器，可能不支持对对象数组的初始化语法

#include <iostream>

using namespace std;

class User {
public:
    // 自定义缺省构造函数
    User(void) {}

    User(string name, int age) {
        this->name = name;
```

```
        this->age = age;
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main (int argc, char* argv[]) {
    User user1; // 编译通过
    user1.print();

    User users1[3]; // 编译通过
    for (int i = 0; i < 3; ++i)
        users1[i].print();

    User* user2 = new User; // 编译通过
    user2->print();
    delete user2;

    User* users2 = new User[3]; // 编译通过
    for (int i = 0; i < 3; ++i)
        users2[i].print();
    delete[] users2;

    return 0;
}
```

- 在has-a关系中，缺省构造具有传染性

```
// hasacons1.cpp

// 在has-a关系中，缺省构造具有传染性

#include <iostream>

using namespace std;

class Order {
public:
    // 已定义构造函数，系统不提供缺省构造函数

    Order(int number) {
        this->number = number;
    }

    int number;
};

class User {
public:
    // 未定义构造函数，系统提供缺省构造函数
```

```
23
24      void print(void) {
25          cout << name << ", " << age << ", " << order.number << endl;
26      }
27
28  private:
29      string name;
30      int age;
31      Order order;
32  };
33
34  int main(void) {
35      // User类的缺省构造将引发其类类型成员变量order
36      // 被缺省构造，但后者并没有缺省构造函数
37
38      User user1; // 编译错误
39      user1.print();
40
41      User* user2 = new User; // 编译错误
42      user2->print();
43      delete user2;
44
45      return 0;
46  }
```

因此，有时必须为一个类提供缺省构造函数，仅仅因为它可能作为另一个对象的子对象而被缺省构造

```
1   // hasacons2.cpp
2
3   // 有时必须为一个类提供缺省构造函数，仅仅因为
4   // 它可能作为另一个对象的子对象而被缺省构造
5
6   #include <iostream>
7
8   using namespace std;
9
10  class Order {
11  public:
12      // 自定义缺省构造函数
13      Order(void) {}
14
15      Order(int number) {
16          this->number = number;
17      }
18
19      int number;
20  };
21
22  class User {
23  public:
24      // 未定义构造函数，系统提供缺省构造函数
25
26      void print(void) {
27          cout << name << ", " << age << ", " << order.number << endl;
```

```
    }

private:
    string name;
    int age;
    Order order;
};

int main(void) {
    // User的缺省构造将以缺省方式构造其Order子对象

    User user1; // 编译通过
    user1.print();

    User* user2 = new User; // 编译通过
    user2->print();
    delete user2;

    return 0;
}
```

若子对象不宜缺省构造，则需要为父对象提供缺省构造函数，并在其中显式地以非缺省方式构造该子对象

```
// hasacons3.cpp

// 若子对象不宜缺省构造，则需要为父对象提供缺省构
// 造函数，并在其中显式地以非缺省方式构造该子对象

#include <iostream>

using namespace std;

class Order {
public:
    // 已定义构造函数，系统不提供缺省构造函数

    Order(int number) {
        this->number = number;
    }

    int number;
};

class User {
public:
    // 自定义缺省构造函数，并显式地以非缺省方式构造其类类型成员变量
    User(void) : order(1000) {}

    void print(void) {
        cout << name << ", " << age << ", " << order.number << endl;
    }

private:
    string name;
```

```
    int age;
    Order order;
};

int main(void) {
    // User的缺省构造将以非缺省方式构造其Order子对象

    User user1; // 编译通过
    user1.print();

    User* user2 = new User; // 编译通过
    user2->print();
    delete user2;

    return 0;
}
```

- 类的常量和引用型成员变量必须被显式地初始化

```
// constref1.cpp

// 类的常量和引用型成员变量必须被显式地初始化

#include <iostream>

using namespace std;

class User {
public:
    // 未定义构造函数，系统提供缺省构造函数

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    const string name;
    int& age;
};

int main(void) {
    // User类包含常量和引用型成员变量，且未被显式地初始化

    User user1; // 编译错误
    user1.print();

    User* user2 = new User; // 编译错误
    user2->print();
    delete user2;

    return 0;
}
```

```
// constref2.cpp

```

```cpp
// 类的常量和引用型成员变量必须被显式地初始化

#include <iostream>

using namespace std;

int age = -1;

class User {
public:
    // 在自定义构造函数中显式初始化常量和引用型成员变量
    User(void) : name("Anonymous"), age(::age) {}

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    const string name;
    int& age;
};

int main(void) {
    // User类包含常量和引用型成员变量，且已被显式地初始化

    User user1; // 编译通过
    user1.print();

    User* user2 = new User; // 编译通过
    user2->print();
    delete user2;

    return 0;
}
```

## 2.3.3 构造和析构顺序

### 2.3.3.1 单个对象

- 构造函数的调用顺序：基类的构造函数→成员类型的构造函数→子类的构造函数
- 析构函数的调用顺序：子类的析构函数→成员类型的析构函数→基类的析构函数

```cpp
// consorder.cpp

// 构造函数的调用顺序：基类->成员->子类
// 析构函数的调用顺序：子类->成员->基类

#include <iostream>

using namespace std;

class Base {
public:
    Base(void) {
        cout << "Base::Base() invoked" << endl;
```

```
    }

    ~Base(void) {
        cout << "Base::~Base() invoked" << endl;
    }
};

class Member {
public:
    Member(void) {
        cout << "Member::Member() invoked" << endl;
    }

    ~Member(void) {
        cout << "Member::~Member() invoked" << endl;
    }
};

class Super : public Base
{
public:
    Super(void) {
        cout << "Super::Super() invoked" << endl;
    }

    ~Super(void) {
        cout << "Super::~Super() invoked" << endl;
    }

private:
    Member member;
};

int main(void) {
    Super super;

    return 0;
}
```

### 2.3.3.2 对象数组

- 数组元素的构造顺序：低地址→高地址
- 数组元素的析构顺序：高地址→低地址

```
// arrayconsorder.cpp

// 数组元素的构造顺序：低地址->高地址
// 数组元素的析构顺序：高地址->低地址

#include <iostream>

using namespace std;

class Object {
public:
```

```
    Object(void) {
        cout << "Object::Object() invoked, " << this << endl;
    }

    ~Object(void) {
        cout << "Object::~Object() invoked, " << this << endl;
    }
};

int main(void) {
    Object* objects = new Object[3];

    delete[] objects;

    return 0;
}
```

## 2.3.4 支持自定义类型转换的构造函数

```
// consconv.cpp

// 支持自定义类型转换的构造函数

#include <iostream>

using namespace std;

class User {
public:
    // 支持从char*到User的类型转换
    User(const char* name, int age = -1) {
        this->name = name;
        this->age = age;
    }

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main(void) {
    User user("Zaphod", 49);
    user.print();

    user = "Antony"; // char*->User
    user.print();

    return 0;
}
```

## 2.4 初始化表

- 通过在类的构造函数中使用初始化表，初始化该类的成员变量

```cpp
// initlist.cpp

// 通过在类的构造函数中使用初始化列表，初始化该类的成员变量

#include <iostream>

using namespace std;

class User {
public:
    // 带有初始化列表的构造函数
    User(string name, int age) : name(name), age(age) {}

    void print(void) {
        cout << name << ", " << age << endl;
    }

private:
    string name;
    int age;
};

int main(void) {
    User user1("Zaphod", 49);
    user1.print();

    User user2 = User("Antony", 37);
    user2.print();

    User* user3 = new User("Stephen", 35);
    user3->print();
    delete user3;

    User users1[] = {
        User("Zaphod", 49),
        User("Antony", 37),
        User("Stephen", 35)};
    for (int i = 0; i < 3; ++i)
        users1[i].print();

    User* users2 = new User[] {
        User("Zaphod", 49),
        User("Antony", 37),
        User("Stephen", 35)};
    for (int i = 0; i < 3; ++i)
        users2[i].print();
    delete[] users2;

    return 0;
}
```

- 类的成员变量按其在类中的定义顺序被依次初始化，而非其出现在初始化表中的顺序

```cpp
// initorder1.cpp

// 类的成员变量按其在类中的定义顺序被依次初始化，而非其出现在初始化表中的顺序

#include <iostream>

using namespace std;

class MyString {
public:
    // 按照成员变量的定义顺序，先初始化len，后初始化str，
    // 因此通过str.size()初始化len的做法显然是错误的
    MyString(const char* psz) : str(psz), len(str.size()) {}

    void show(void) {
        cout << str << ", " << len << endl;
    }

private:
    int len;
    string str;
};

int main(void) {
    MyString str("hello world");
    str.show();

    return 0;
}
```

```cpp
// initorder2.cpp

// 类的成员变量按其在类中的定义顺序被依次初始化，而非其出现在初
// 始化表中的顺序，因此在初始化表中，应尽量避免成员变量间的耦合

#include <string.h>

#include <iostream>

using namespace std;

class MyString {
public:
    MyString(const char* psz) : str(psz), len(strlen(psz)) {}

    void show(void) {
        cout << str << ", " << len << endl;
    }

private:
    int len;
    string str;
};
```

```
24
25 int main(void) {
26     MyString str("hello world ");
27     str.show();
28
29     return 0;
30 }
```

- 类的常量和引用型成员变量，必须初始化表中初始化
- 类的成员子对象和基类子对象，如果没有缺省构造函数，也必须在初始化表中初始化

## 2.5 this指针

- 一般而言，关键字this是一个指针，对于一般成员函数，它指向调用该函数的对象，而对于构造和析构函数，它则指向这个正在被创建和销毁的对象

```
1  // calling.h
2
3  // 声明Student类
4
5  #pragma once
6
7  class Student {
8  public:
9      Student(const string& name);
10
11     void who(void) const;
12
13     string name;
14 };
```

```
1  // calling.cpp
2
3  // 实现Student类
4
5  #include <iostream>
6
7  #include "Calling.h"
8
9  using namespace std;
10
11 Student::Student(const string& name) : name(name) {}
12 /*
13 void Student::who(void) const {
14     cout << "我是" << name << "。" << endl;
15 }
16 */
17 extern "C" void _ZNK7Student3whoEv(const Student* _this) {
18     cout << "我是" << _this->name << "。" << endl;
19 }
20
21 int main(void) {
22     Student student("张飞");
23
24     student.who();
```

```
25
26      return 0;
27  }
```

- 很多时候，为了方便起见，常常希望一个类的某个数据成员和这个类的构造函数的相应参数取相同的标识符。为此，在构造函数的内部，通过this关键字，可以将二者有效地区分开来

```
1   // this.cpp
2
3   // 显式使用this指针
4
5   #include <iostream>
6
7   using namespace std;
8
9   class User {
10  public:
11      User(string name, int age) {
12          cout << "User::User(): this=" << this << endl;
13
14          this->name = name;
15          this->age = age;
16      }
17
18      void print(void) const {
19          cout << "User::print(): this=" << this << endl;
20
21          cout << name << ", " << age << endl;
22          cout << this->name << ", " << this->age << endl;
23      }
24
25  private:
26      string name;
27      int age;
28  };
29
30  int main(void) {
31      User user("Paul", 33);
32
33      cout << "main(): &user="<< &user << endl;
34
35      user.print();
36
37      return 0;
38  }
```

- 基于this指针的自身引用还被广泛地应用于那些支持多重串联调用的函数中

```
1   // retthis.cpp
2
3   // 从函数中返回this
4
5   #include <iostream>
6
7   using namespace std;
```

```
8
9  class Integer {
10 public:
11     Integer(int value = 0) : value(value) {}
12
13     Integer& inc(int inc = 1) {
14         value += inc;
15         return *this;
16     }
17
18     Integer& dec(int dec = 1) {
19         value -= dec;
20         return *this;
21     }
22
23     void show(void) const {
24         cout << value << endl;
25     }
26
27 private:
28     int value;
29 };
30
31 int main(void) {
32     Integer i;
33
34     i.inc().inc(2).inc(3).show();
35     i.dec().dec(2).dec(3).show();
36
37     return 0;
38 }
```

- 将this指针作为函数的参数，可协助完成对象间的交互

```
1  // paramthis.cpp
2
3  // 将this指针作为函数的参数
4
5  #include <iostream>
6
7  using namespace std;
8
9  class Student;
10
11 class Teacher {
12 public:
13     void educate(Student* student);
14
15     void reply(const char* answer) {
16         cout << answer << endl;
17     }
18 };
19
20 class Student {
21 public:
22     void ask(const char* question, Teacher* teacher) {
```

```
        cout << question << endl;

        teacher->reply("一般而言，关键字this是一个指针，对于一般成员函数，"
            "它指向调用该函数的对象，而对于构造和析构函数，"
            "它则指向这个正在被创建和销毁的对象。");
    }
};

void Teacher::educate(Student* student) {
    student->ask("什么是this指针？", this); // 把自己传给student
}

int main (void) {
    Teacher teacher;
    Student student;

    teacher.educate(&student);

    return 0;
}
```

## 2.6 常量型成员函数与常量型对象

- 常量型成员函数中的this指针为常量型，以此防止对成员变量的意外修改。对常量型对象只能调用其常量型成员函数

```
// constfunc.cpp

// 常量型成员函数中的this指针为常指针，以此防止对成员变量的意外修改

#include <iostream>

using namespace std;

class Date {
public:
    Date(int year, int mon, int day) : year(year), mon(mon), day(day) {}

    void update(int year, int mon, int day) {
        this->year = year;
        this->mon = mon;
        this->day = day;
    }

    void show(void) const {
        // cout << year-- << mon << "-" << day << endl; // 编译错误，this指针为常指针
        cout << year << "-" << mon << "-" << day << endl;
    }

private:
    int year, mon, day;
};

int main(void) {
```

```
    Date d1(2024, 6, 24);
    d1.show();

    // 对常量型对象只能调用其常量型成员函数

    const Date d2 = d1;
    d2.show();
    // d2.update(2025, 7, 25); // 编译错误

    const Date& d3 = d2;
    d3.show();
    // d3.update(2025, 7, 25); // 编译错误

    const Date* d4 = &d3;
    d4->show();
    // d4->update(2025, 7, 25); // 编译错误

    return 0;
}
```

- 被声明为mutable的成员变量可以被常量型成员函数修改

```
// mutable.cpp

// 被声明为mutable的成员变量可以被常量型成员函数修改

#include <iostream>

using namespace std;

class Document {
public:
    Document(void) : printTimes(0) {}

    void print(void) const {
        if (printTimes >= 3) {
            cout << "Unable to print over 3 times!" << endl;
            return;
        }

        // ...

        ++printTimes; // mutable成员变量可以被常量型成员函数修改
    }

private:
    mutable unsigned int printTimes;
};

int main(void){
    Document doc;

    doc.print();
    doc.print();
    doc.print();
    doc.print();
```

```
35
36      return 0;
37  }
```

- 常量型成员函数和非常量型成员函数构成重载关系，通过常量型对象调用常量型成员函数，通过非常量型对象调用非常量型成员函数，但如果没有非常量型成员函数，通过非常量型对象也能调用常量型成员函数

```
1  // constover.cpp
2
3  // 常量型成员函数和非常量型成员函数构成重载关系
4
5  #include <iostream>
6
7  using namespace std;
8
9  class A {
10 public:
11     // 非常量型版本
12     // void foo(void) {
13     //     cout << "A::foo() invoked" << endl;
14     // }
15
16     // 常量型版本
17     void foo(void) const  {
18         cout << "A::foo() const invoked" << endl;
19     }
20 };
21
22 int main(void) {
23     A a1; // 非常量型对象
24     a1.foo(); // 匹配到非常量型版本
25
26     const A& a2 = a1; // 常量型对象（引用）
27     a2.foo(); // 匹配到常量型版本
28
29     A* a3 = const_cast<A*>(&a2); // 非常量型对象（指针）
30     a3->foo(); // 匹配到非常量型版本
31
32     return 0;
33 }
```

# 2.7 析构函数

## 2.7.1 基本用法

```
1  class 类名 {
2      ~类名(void) {
3          函数体;
4      }
5  };
```

析构函数没有参数，因此无法重载。

```cpp
// destructor.cpp

// 析构函数的基本用法

#include <iostream>

using namespace std;

class Array {
public:
    Array(size_t size) : array(new int[size]), size(size) {
        cout << "Array::Array() invoked, size=" << size << endl;
    }

    // 析构函数，没有参数，不能重载
    ~Array(void) {
        cout << "Array::~Array() invoked, size=" << size << endl;

        delete[] array; // 释放在构造函数中分配的内存
    }

private:
    int* array;
    size_t size;
};

int main(void) {
    Array a1(10);

    {
        Array a2(20);
    }

    Array* a3 = new Array(30);

    Array* a4 = new Array[] {Array(40), Array(50), Array(60)};

    delete[] a4;

    delete a3;
    // delete a3; // 运行错误，一个对象的析构函数只调用一次

    return 0;
}
```

## 2.7.2 缺省析构函数和自定义析构函数

- 对于未定义析构函数的类，系统会提供缺省析构函数，该析构函数会调用成员及基类的析构函数

```cpp
// defdescan.cpp

// 若未定义析构函数，系统会提供缺省析构函数，该
// 析构函数会调用其成员及基类子对象的析构函数

#include <iostream>
```

```
7
8  using namespace std;
9
10 class A {
11 public:
12     A(void) {
13         cout << "A::A() invoked" << endl;
14     }
15
16     ~A(void) {
17         cout << "A::~A() invoked" << endl;
18     }
19 };
20
21 class B {
22     // 缺省析构函数负责调用A的析构函数，销毁其A类子对象
23
24 private:
25     A a;
26 };
27
28 int main(void) {
29     B b;
30
31     return 0;
32 }
```

- 缺省析构函数不负责释放动态分配的资源

```
1  // defdescannot.cpp
2
3  // 缺省析构函数不负责释放动态分配的资源
4
5  #include <iostream>
6
7  using namespace std;
8
9  class A {
10 public:
11     A(void) {
12         cout << "A::A() invoked" << endl;
13     }
14
15     ~A(void) {
16         cout << "A::~A() invoked" << endl;
17     }
18 };
19
20 class B {
21 public:
22     B(void) : a(new A) {}
23
24     // 缺省析构函数不负责销毁成员a指向的A类对象
25
26 private:
27     A* a;
```

```
};

int main(void) {
    B b;

    return 0;
}
```

- 对于动态分配的资源，必须通过自己定义的析构函数进行释放

```
// mustdes.cpp

// 对于动态分配的资源，必须通过自己定义的析构函数进行释放

#include <iostream>

using namespace std;

class A {
public:
    A(void) {
        cout << "A::A() invoked" << endl;
    }

    ~A(void) {
        cout << "A::~A() invoked" << endl;
    }
};

class B {
public:
    B(void) : a(new A) {}

    // 自定义析构函数负责销毁成员a指向的A类对象
    ~B(void) {
        delete a;
    }

private:
    A* a;
};

int main(void) {
    B b;

    return 0;
}
```

## 2.7.3 析构函数所释放的资源不仅限于内存资源

```
// desmore.cpp

// 析构函数所释放的资源不仅限于内存资源

#include <stdio.h>
```

```cpp
#include <assert.h>

#include <iostream>

using namespace std;

class File {
public:
    File(const char* filename, const char* mode) : fp(fopen(filename, mode))
{
        assert(fp);

        cout << "File::File(): Open file" << endl;
    }

    ~File(void) {
        fclose(fp); // 关闭文件

        cout << "File::~File(): Close file" << endl;
    }

    size_t read(void* buffer, size_t count) {
        return fread(buffer, 1, count, fp);
    }

    size_t write(const void* buffer, size_t count) {
        return fwrite(buffer, 1, count, fp);
    }

private:
    FILE* fp;
};

int main(void) {
    File file("./desmore.cpp", "r");

    char buffer[1024] = {};
    file.read(buffer, sizeof(buffer) - 1);
    cout << buffer << endl;

    return 0;
}
```

# 2.8 拷贝构造与拷贝赋值

## 2.8.1 拷贝构造函数

- 默认的拷贝构造就是按字节复制一份

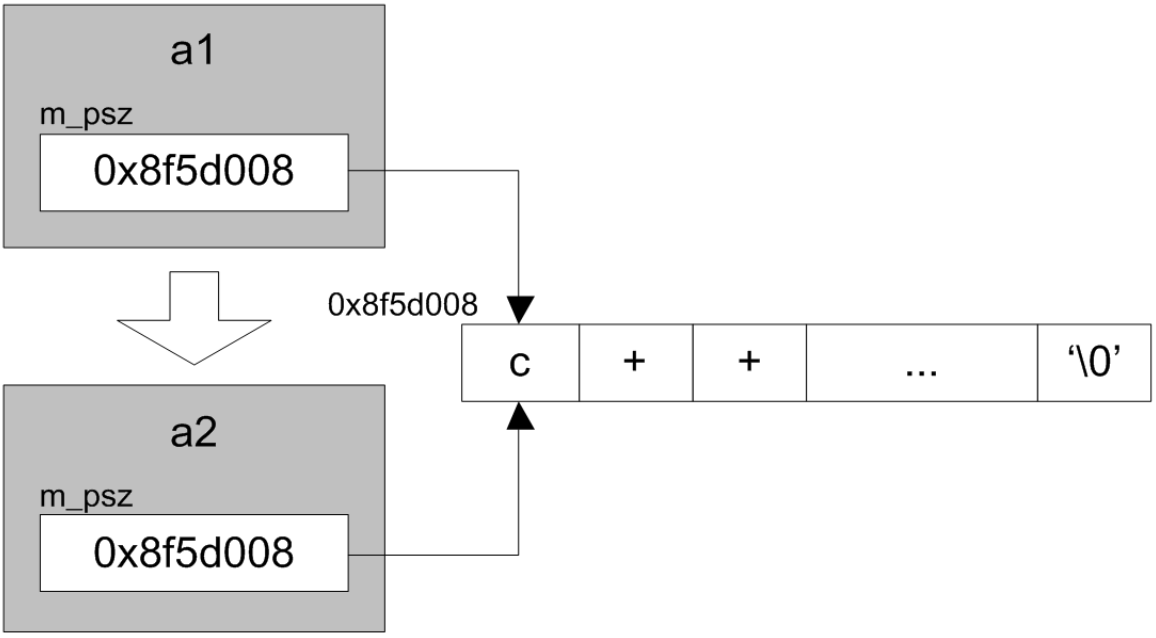
```cpp
// copybytes.cpp

// 默认方式的拷贝构造就是按字节复制一份

#include <string.h>

```

```cpp
#include <iostream>

using namespace std;

class A {
public:
    A (int n, double d, const char* s) : n(n), d(d) {
        strcpy(this->s, s);
    }

    // 未定义拷贝构造函数，采用默认的字节复制的方式进行拷贝构造

    void show(void) const {
        cout << n << ", " << d << ", " << s << endl;
    }

private:
    int n;
    double d;
    char s[1024];
};

int main(void) {
    A a1(123, 4.56, "789");
    a1.show();

    // A a2(a1); // 拷贝构造
    A a2 = a1; // 拷贝构造
    a2.show();

    return 0;
}
```

- 默认的拷贝构造不能实现深拷贝

```cpp
// defcpcons.cpp

// 默认的字节复制方式的拷贝构造不能实现深拷贝

#include <string.h>

#include <iostream>

using namespace std;

class A {
public:
    A (int n, double d, const char* s) : n(n), d(d),
        s(strcpy(new char[strlen(s)+1], s)) {}

    // 未定义拷贝构造函数，采用默认的字节复制的方式进行拷贝构造

    ~A (void) {
        cout << "A::~A(): s=" << static_cast<void*>(s) << endl;
        delete[] s;
    }
```

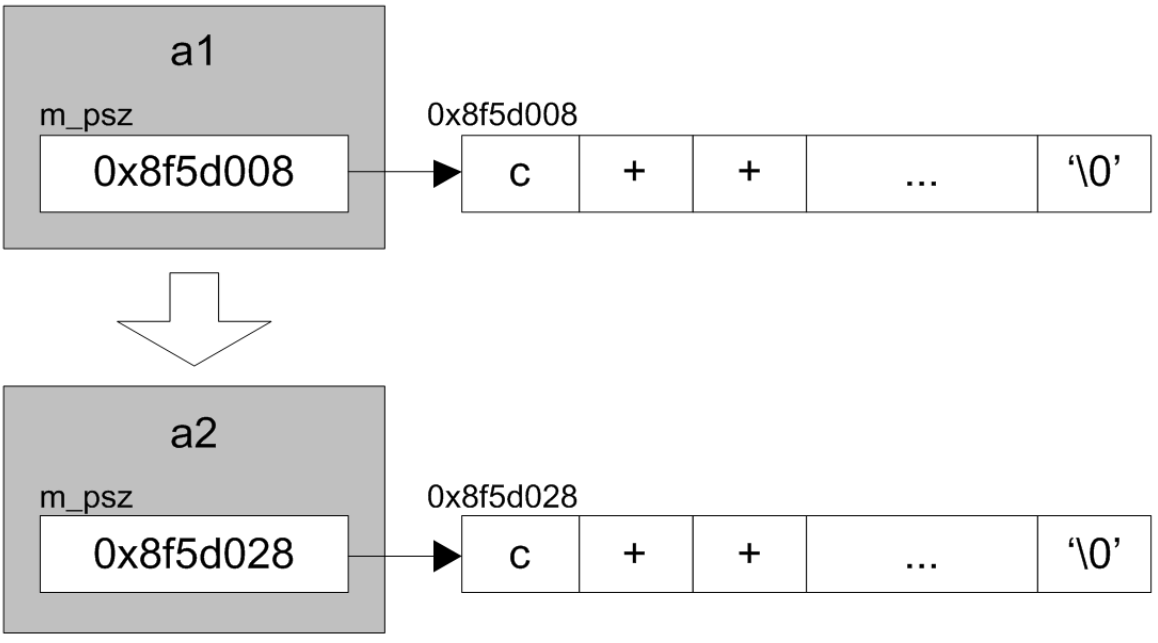
```
22
23      void show(void) const {
24          cout << n << ", " << d << ", " << s << endl;
25      }
26
27  private:
28      int n;
29      double d;
30      char* s;
31  };
32
33  int main(void) {
34      A a1(123, 4.56, "789");
35      a1.show();
36
37      // 拷贝构造
38      A a2 = a1; // 字节复制方式的拷贝构造将导致a1和a2中的s指向同一块内存，
39                 // 该内存在两个对象的析构函数中各被释放一次，导致崩溃
40      a2.show();
41
42      return 0;
43  }
```




- 自定义拷贝构造函数，实现对象间的深拷贝，进而获得完整意义上的副本

```
1   // cpcons.cpp
2
3   // 通过自定义的拷贝构造函数，可实现对象间的深拷贝，进而获得完整意义上的副本
4
5   #include <string.h>
6
7   #include <iostream>
8
9   using namespace std;
10
11  class A {
12  public:
```

```
13      A (int n, double d, const char* s) : n(n), d(d),
14          s(strcpy(new char[strlen(s)+1], s)) {}
15
16      // 自定义的支持深拷贝的拷贝构造函数
17      A (const A& a) : n(a.n), d(a.d),
18          s(strcpy(new char[strlen(a.s)+1], a.s)) {}
19
20      ~A (void) {
21          cout << "A::~A(): s=" << static_cast<void*>(s) << endl;
22          delete[] s;
23      }
24
25      void show(void) const {
26          cout << n << ", " << d << ", " << s << endl;
27      }
28
29  private:
30      int n;
31      double d;
32      char* s;
33  };
34
35  int main(void) {
36      A a1(123, 4.56, "789");
37      a1.show();
38
39      // 拷贝构造
40      A a2 = a1; // a1和a2中的s各自指向独立的内存，在各自的析构函数中被释放
41      a2.show();
42
43      return 0;
44  }
```




| 自定义 | 系统缺省提供 |
| --- | --- |
| 无 | 缺省构造函数和拷贝构造函数 |

| 自定义 | 系统缺省提供 |
| --- | --- |
| 除拷贝构造以外的其它构造函数 | 拷贝构造函数 |
| 拷贝构造函数 | 无 |

## 2.8.2 拷贝构造的时机

- 构造对象的副本
- 以对象为参数调用函数
- 从函数中返回对象
- 以对象的方式捕获异常

```
// whencpcons.cpp

// 拷贝构造的时机:

// 1) 构造对象的副本
// 2) 以对象为参数调用函数
// 3) 从函数中返回对象
// 4) 以对象的方式捕获异常

#include <string.h>
#include <iostream>
using namespace std;

class A
{
public:
    A (void) {}

    A (const A& a) {
        cout << "A::A(A) invoked" << endl;
    }
};

void foo(A a) {
    cout << "foo() invoked" << endl;
}

A bar(void) {
    cout << "bar() invoked" << endl;

    A a;
    return a;
}

int main(void) {
    A a1;

    A a2 = a1; // 构造对象的副本

    foo(a2); // 以对象为参数调用函数
```

```
41
42      bar(); // 从函数中返回对象
43
44      try {
45          throw A();
46      }
47      catch (A a) { // 以对象的方式捕获异常
48      }
49
50      return 0;
51  }
52
53  // g++ -fno-elide-constructors whencpcons.cpp
```

## 2.8.3 拷贝赋值操作符函数

- 自定义拷贝构造只能解决创建对象副本时的深拷贝问题，默认的拷贝赋值仍然是按字节复制

```
1   // defcpassign.cpp
2
3   // 拷贝构造只能解决构造对象副本时的深拷贝问题，默认方式的拷贝赋值仍然是按字节复制
4
5   #include <string.h>
6
7   #include <iostream>
8
9   using namespace std;
10
11  class A {
12  public:
13      A (int n, double d, const char* s) : n(n), d(d),
14          s(strcpy(new char[strlen(s)+1], s)) {}
15
16      A (const A& a) : n(a.n), d(a.d),
17          s(strcpy(new char[strlen(a.s)+1], a.s)) {}
18
19      // 未定义拷贝赋值操作符，采用默认的字节复制的方式进行拷贝赋值
20
21      ~A (void) {
22          cout << "A::~A(): s=" << static_cast<void*>(s) << endl;
23          delete[] s;
24      }
25
26      void show(void) const {
27          cout << n << ", " << d << ", " << s << endl;
28      }
29
30  private:
31      int n;
32      double d;
33      char* s;
34  };
35
36  int main(void) {
37      A a1(123, 4.56, "789");
```

```
    a1.show();

    A a2(987, 6.54, "321");
    a2.show();

    // 拷贝赋值
    a2 = a1; // 字节复制方式的拷贝赋值将导致a1和a2中的s指向同一块内存，
             // 该内存在两个对象的析构函数中各被释放一次，导致崩溃
    a2.show();

    return 0;
}
```

- 自定义拷贝赋值操作符函数，实现对象间的深拷贝，进而获得完整意义上的副本

```
// cpassign1.cpp

// 通过自定义的拷贝赋值操作符函数，可实现对象间的深拷贝，进而获得完整意义上的副本

#include <string.h>

#include <iostream>

using namespace std;

class A {
public:
    A (int n, double d, const char* s) : n(n), d(d),
        s(strcpy(new char[strlen(s)+1], s)) {}

    A (const A& a) : n(a.n), d(a.d),
        s(strcpy(new char[strlen(a.s)+1], a.s)) {}

    // 自定义的支持深拷贝的拷贝赋值操作符函数
    void operator=(const A& a) {
        if (&a != this) { // 防止自赋值
            n = a.n;
            d = a.d;

            delete[] s;

            s = strcpy(new char[strlen(a.s)+1], a.s);
        }
    }

    ~A (void) {
        cout << "A::~A(): s=" << static_cast<void*>(s) << endl;
        delete[] s;
    }

    void show(void) const {
        cout << n << ", " << d << ", " << s << endl;
    }

private:
    int n;
```

```
    double d;
    char* s;
};

int main(void) {
    A a1(123, 4.56, "789");
    a1.show();

    A a2(987, 6.54, "321");
    a2.show();

    // 拷贝赋值
    a2 = a1; // a1和a2中的s各自指向独立的内存，在各自的析构函数中被释放
    a2.show();

    return 0;
}
```

- 更好的拷贝赋值操作符函数，支持连续赋值

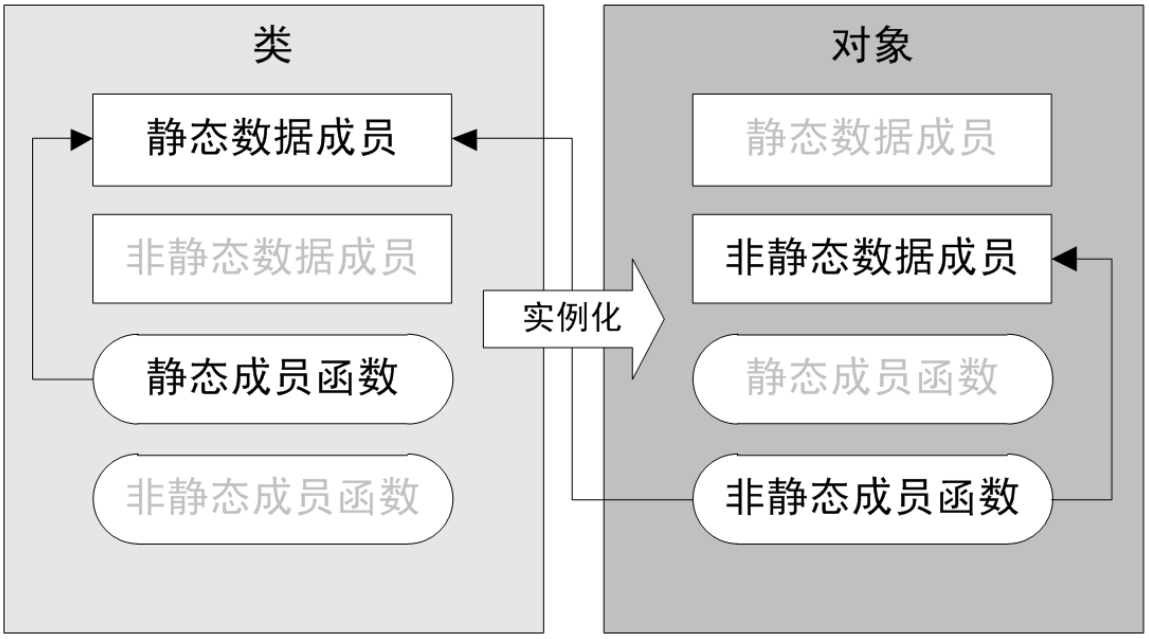
```
// cpassign2.cpp

// 更好的拷贝赋值操作符函数，支持连续赋值

#include <string.h>

#include <iostream>

using namespace std;

class A {
public:
    A (int n, double d, const char* s) : n(n), d(d),
        s(strcpy(new char[strlen(s)+1], s)) {}

    A (const A& a) : n(a.n), d(a.d),
        s(strcpy(new char[strlen(a.s)+1], a.s)) {}

    // 更好的支持连续赋值拷贝赋值操作符函数
    A& operator= (const A& a) {
        if (&a != this) {
            n = a.n;
            d = a.d;

            A _a = a;

            swap(s, _a.s);
        }

        return *this; // 返回自引用
    }

    ~A (void) {
        cout << "A::~A(): s=" << static_cast<void*>(s) << endl;
        delete[] s;
    }
```

```
37
38      void show(void) const {
39          cout << n << ", " << d << ", " << s << endl;
40      }
41
42  private:
43      int n;
44      double d;
45      char* s;
46  };
47
48  int main(void) {
49      A a1(123, 4.56, "789");
50      a1.show();
51
52      A a2(987, 6.54, "321");
53      a2.show();
54
55      A a3(222, 5.55, "888");
56      a3.show();
57
58      // 连续拷贝赋值
59      a3 = a2 = a1; // a1->a2, a2->a3
60      // (a3 = a2) = a1; // a2->a3, a1->a3
61      a2.show();
62      a3.show();
63
64      return 0;
65  }
```

# 2.9 静态成员

## 2.9.1 静态成员属于类而非对象

- 静态成员变量的定义和初始化，只能在类的外部而不能在构造函数中进行
- 静态成员变量被该类的多个对象实例共享
- 访问静态成员，既可以通过类也可以通过对象，但最好通过类
- 静态成员函数只能访问静态成员，而非静态成员函数既可以访问静态成员，也可以访问非静态成员




```cpp
// staticmem.cpp

// 类的静态成员属于类而非静态成员属于对象，类在
// 对象间共享并被对象所访问，但类不能访问对象

#include <iostream>

using namespace std;

class Account {
public:
    Account (const string& name, double balance) :
        name(name), balance(balance) {}

    static void setRate(double rate) {
        // 静态成员函数只能访问静态成员
        Account::rate = rate;
    }

    static double getRate(void) {
        // 静态成员函数只能访问静态成员
        return rate;
    }

    double settle(void) {
        // 非静态成员函数既可以访问静态成员，也可以访问非静态成员
        return balance *= (1 + rate / 100.);
    }

private:
    string name;
    double balance;
    static double rate; // 所有账户对象均共享同一利率
};

// 静态成员变量需在类外定义并初始化
```

```cpp
double Account::rate = 0.5;

int main(void) {
    Account acc1("Richard Lee", 10000.0);
    Account acc2("Marshall Cline", 10.0);

    acc1.setRate(0.4); // 通过对象访问静态成员
    cout << Account::getRate() << endl; // 通过类访问静态成员

    for (int i = 0; i < 3; ++i)
        cout << acc1.settle() << ", " << acc2.settle() << endl;

    Account::setRate(0.6); // 通过类访问静态成员
    cout << acc1.getRate() << endl; // 通过对象访问静态成员

    for (int i = 0; i < 3; ++i)
        cout << acc1.settle() << ", " << acc2.settle() << endl;

    return 0;
}
```

## 2.9.2 单例模式

- 饿汉单例

```cpp
// hungry.cpp

// 饿汉单例

#include <iostream>

using namespace std;

class A {
public:
    // 获取唯一对象实例的静态方法
    static A& inst(void) {
        return a;
    }

private:
    // 构造函数私有化
    A(void) {}
    A(const A& a) {}

    static A a; // 唯一的对象实例
};

A A::a;

int main(void) {
    A& a1 = A::inst();
    A& a2 = A::inst();
    A& a3 = A::inst();
    cout << &a1 << ", " << &a2 << ", " << &a3 << endl;
```

```
31
32      // A a4; // 直接构造被禁止
33      // A a5(a3); // 拷贝构造被禁止
34
35      return 0;
36  }
```

- 懒汉单例

```
1   // lazy.cpp
2
3   // 懒汉单例
4
5   #include <iostream>
6
7   using namespace std;
8
9   class A {
10  public:
11      // 获取唯一对象实例的静态方法
12      static A& inst(void) {
13          if (!a) a = new A;
14
15          return *a;
16      }
17
18  private:
19      // 构造函数私有化
20      A(void) {}
21      A(const A& a) {}
22
23      static A* a; // 指向唯一对象实例的指针
24  };
25
26  A* A::a = NULL;
27
28  int main(void) {
29      A& a1 = A::inst();
30      A& a2 = A::inst();
31      A& a3 = A::inst();
32      cout << &a1 << ", " << &a2 << ", " << &a3 << endl;
33
34      // A a4; // 直接构造被禁止
35      // A a5(a3); // 拷贝构造被禁止
36
37      return 0;
38  }
```

# 2.10 成员指针

## 2.10.1 指向成员变量的指针

- 定义语法：

```
成员变量类型 类名::*指针变量名;
```

例如：

```
string Account::*pname;
double Account::*pbalance;
```

- 赋值及初始化语法：

```
指针变量名 = &类名::成员变量名;
```

例如：

```
pname = &Account::name;
pbalance = &Account::balance;
```

- 解引用语法：

```
对象.*指针变量名
对象指针->*指针变量名
```

例如：

```
Account acc("Richard Lee", 10000.0);
cout << acc.*pname << endl;
Account* pacc = &acc;
cout << pacc->*pbalance << endl;
```

## 2.10.2 指向成员函数的指针

- 定义语法：

```
成员函数返回类型 (类名::*指针变量名)(形参表);
```

例如：

```
double (Account::*psettle)(void);
```

- 赋值及初始化语法：

```
指针变量名 = &类名::成员函数名;
```

例如：

```
psettle = &Account::settle;
```

- 解引用语法：

```
(对象.*指针变量名)(实参表)
(对象指针->*指针变量名)(实参表)
```

例如：

```
Account acc("Richard Lee", 10000.0);
cout << (acc.*psettle)() << endl;
Account* pacc = &acc;
cout << (pacc->*psettle)() << endl;
```

## 2.10.3 静态成员指针

```
double* prate = &Account::rate;
void (*psetRate)(double) = &Account::setRate;
double (*pgetRate)(void) = &Account::getRate;

cout << *prate << endl;
(*psetRate)(0.4);
cout << pgetRate() << endl;
```

```
// memptr.cpp

// 指向成员的指针

#include <iostream>

using namespace std;

class Account {
public:
    Account (const string& name, double balance) :
        name(name), balance(balance) {}

    static void setRate(double rate) {
        Account::rate = rate;
    }

    static double getRate(void) {
        return rate;
    }

    double settle(void) {
        return balance *= (1 + rate / 100.);
    }

    string name;
    double balance;
    static double rate;
};
```

```cpp

double Account::rate = 0.5;

int main(void) {
    string Account::*pname = &Account::name;
    double Account::*pbalance = &Account::balance;
    double* prate = &Account::rate;

    void (*psetRate)(double) = &Account::setRate;
    double (*pgetRate)(void) = &Account::getRate;
    double (Account::*psettle)(void) = &Account::settle;

    Account acc("Richard Lee", 10000.0), *pacc = &acc;
    cout << acc.*pname << ", " << pacc->*pbalance << ", " << *prate << endl;

    (*psetRate)(0.4);
    cout << (*pgetRate)() << endl;

    for (int i = 0; i < 3; ++i)
        cout << (acc.*psettle)() << endl;

    psetRate(0.6);
    cout << pgetRate() << endl;

    for (int i = 0; i < 3; ++i)
        cout << (pacc->*psettle)() << endl;

    return 0;
}
```