# 12 类和动态内存分配

# 前言

- 使用new和delete，以及如何处理由于使用动态内存而引起的一些微妙的问题
  - 涉及的主题影响构造函数和析构函数的设计以及运算符的重载
- OOP准则之一
  - 最好在程序运行时，而不是编译时确定内存大小
- 有关动态内存管理，例如，字符串
  - 可以使用string，但应该深入学习内存管理

# 示例和静态类成员

# 1 动态内存和类

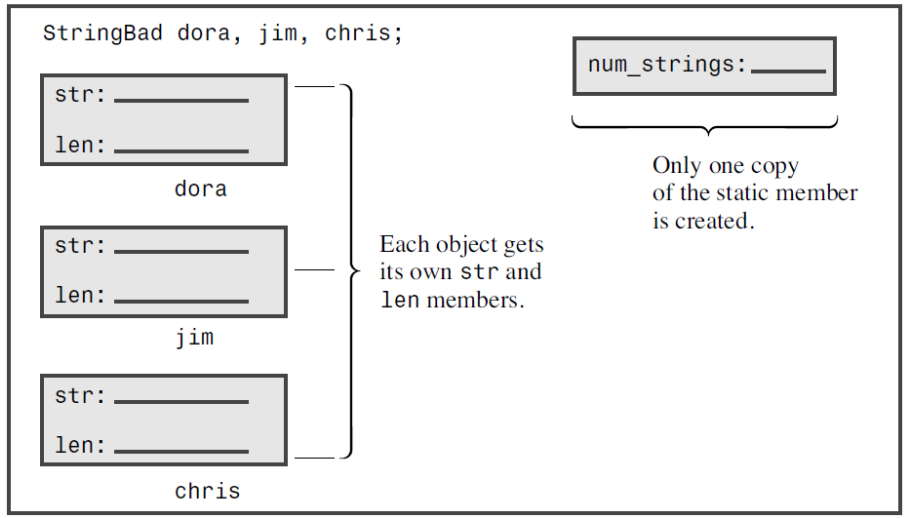
- 问题
  - 需要在运行时决定内存分配
  - 非编译时决定
- new/delete
  - 会导致一系列问题
  - 问题的本质：多个指针指向同一块内存，删除这些指针会出问题

# 1.1 复习示例和静态类成员

P12.1 strngbad.h P12.2 strngbad.cpp P12.3 vegnews.cpp

- static数据成员：static int num_strings
  - 所有对象，共享同一个静态成员
  - 需要在cpp中初始化(声明中不分配内存)

```
Class StringBad
{
private:
   char * str;
   int len;
   static int num_strings;
public:
   ...
};
```




```
1.  class StringBad
2.  {
3.  private:
4.      char *str;                // pointer to string
5.      unsigned int len;         // length of string
6.      static int num_strings; // number of objects
7.  public:
8.      StringBad(const char *s); // constructor
9.      StringBad();              // default constructor
10.     ~StringBad();             // destructor
11.                               // friend function
12.     friend std::ostream &operator<<(std::ostream &os,
13.                                     const StringBad &st);
14. };

15. // initializing static class member
16. int StringBad::num_strings = 0;
```

# 内存相关函数

- L1：在类声明外初始化静态成员变量
  - 声明描述了如何分配内存，但并不分配内存
  - 静态数据成员在包含类方法的文件中初始化
- num_strings记录String对象的总数
- 构造函数：必须分配足够的内存来存储字符串，然后将字符串复制到内存中
  - 首先，分配好内存
  - 其次，拷贝数据
  - 注意，不能str = s; //只保存地址，没有创建字符串副本
- 析构函数
  - 对于使用new[]分配的内存，用delete[]释放内存

```
1.  int StringBad::num_strings = 0;
2.  StringBad::StringBad(const char *s){
3.      len = std::strlen(s);    // set size
4.      str = new char[len + 1]; // allot storage
5.      std::strcpy(str, s);     // initialize pointer
6.      num_strings++;           // set object count
7.  }

8.  StringBad::StringBad() { // default constructor
9.      len = 4;
10.     str = new char[4];
11.     std::strcpy(str, "C++"); // default string
12.     num_strings++;
13. }

14. StringBad::~StringBad() { // necessary destructor
15.     --num_strings;                                  // I
16.     delete[] str;                                   //
17. }
```

# 内存相关函数

- 所有问题都是由编译器自动生成的成员函数引起，这些函数的行为与类设计不符
  - 隐式复制构造函数：StringBad sailor = sports;
  - 隐式赋值运算符：knot = headline1;

```
1.  StringBad::StringBad(const char *s){
2.      len = std::strlen(s);     // set size
3.      str = new char[len + 1]; // allot storage
4.      std::strcpy(str, s);      // initialize pointer
5.   }
6.  StringBad::StringBad(){ // default constructor
7.      len = 1;
8.      str = new char[4];
9.      str[0] =‘\0’;
10.  }
11. StringBad::~StringBad(){   // necessary destructor
12.      if (str)  delete[] str;
13. }
```

```
1.  void callme1(StringBad & rsb){
2.      cout << "    \"" << rsb << "\"\n";
3.  }

4.  void callme2(StringBad sb){
5.      cout << "    \"" << sb << "\"\n";
6.  }

7.  int main(){
8.      StringBad headline1("Celery Stalks");
9.      StringBad headline2("Lettuce Prey");
10.     StringBad sports("Spinach Leaves Bowl");
11.     callme1(headline1);
12.     callme2(headline2);
13.     StringBad sailor = sports;
14.     StringBad knot;
15.     knot = headline1;
16.     return 0;
17. }
```

# 1.2 特殊成员函数

- C++自动提供的成员函数(如果类中没有定义)
  - 默认构造函数
  - 默认析构函数
  - 复制构造函数
  - 赋值运算符
  - 地址运算符(一般不会有问题)

# 默认构造函数

- 如果没有提供任何构造函数， C++将创建/补上默认构造函数 MyClass()
  - 无参数
  - 不执行任何操作
- 如果定义了构造函数， C++将不会定义默认构造函数
- 带参数的构造函数也可以是默认构造函数(只要所有参数都有默认值)
  - Klunk(int n = 0) { klunk ct = n;)
    - Klunk() { klunk ct = 0) / / constructor 1
    - Klunk(int n = 0) { klunk ct= n;) // ambiguous constructor 2
  - 可能带来二义性
    - Klunk kar(10); // clearly matches Klunt(int n)
    - Klunk bus; // could match either constructor

# 复制构造函数

- 复制构造函数，用于将一个对象复制到新创建的对象中
  - 它用于初始化过程中(包括按值传递参数)，不是常规的赋值过程中
  - `Class_name(const Class_name &)`

- 它接受一个指向类对象的常量引用作为参数
  - `StringBad(const StringBad &)`

# 何时调用复制构造函数

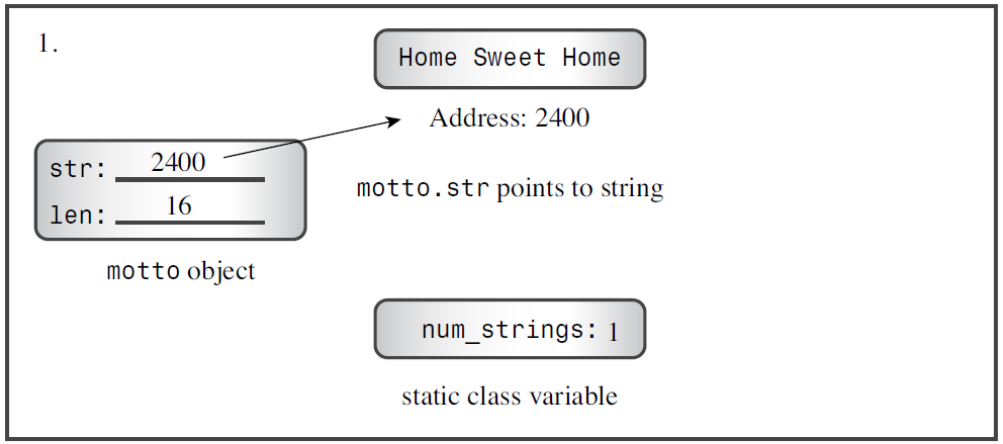
- 新建一个对象并将其初始化为同类现有对象时，复制构造函数都将被调用，4种情况

```
StringBad ditto(motto);                  // calls StringBad(const StringBad &)
StringBad metoo = motto;                 // calls StringBad(const StringBad &)
StringBad also = StringBad(motto);       // calls StringBad(const StringBad &)
StringBad *pStringBad = new StringBad(motto); // calls StringBad(const StringBad &)
```

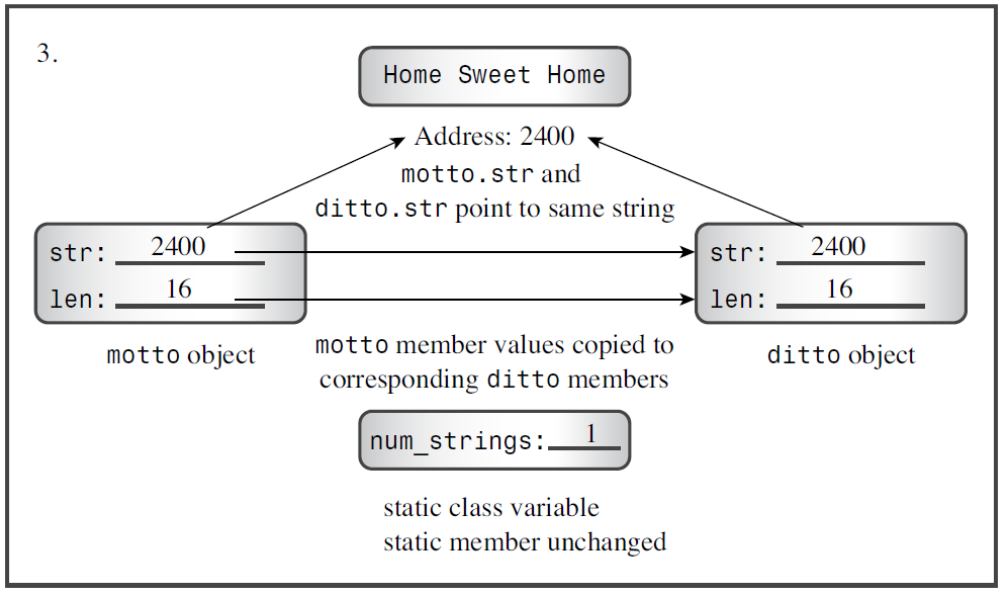
- 注
  - 中间2种声明可能直接使用复制构造函数创建metoo和also，也可能复制构造生成临时对象，然后将临时对象的内容赋给metoo和also，取决于具体的实现
  - 后一种声明使用motto初始化一个匿名对象，并将新对象的地址赋给 pstring 指针。
- 每当程序生成对象副本时，编译器将使用复制构造函数
  - 当函数按值传递对象，或函数返回对象时，都将使用复制构造函数
    - 按值传递意味着创建原始变量的一个副本
  - 编译器生成临时对象时，也将使用复制构造函数
    - 生成临时对象随编译器而异，但无论是哪种编译器，当按值传递和返回对象时，都将调用复制构造函数

# 默认的复制构造函数的功能

- 默认的复制构造函数
  - 逐个复制非静态成员(成员复制也称为浅复制)
    - 复制的是成员的值。没有内存分配操作
  - 如果成员本身就是类对象
    - 使用这个类的复制构造函数来复制成员对象
  - 静态成员不受影响
    - 因为它们属于整个类，而不是各个对象




```
2.  String ditto(motto); // default copy constructor
```

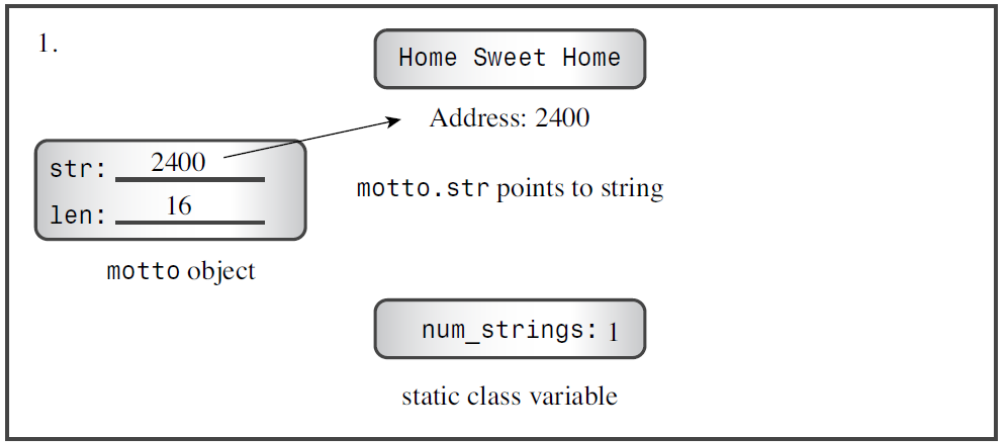


# 1.3 回到Stringbad：复制构造函数的哪里出了问题

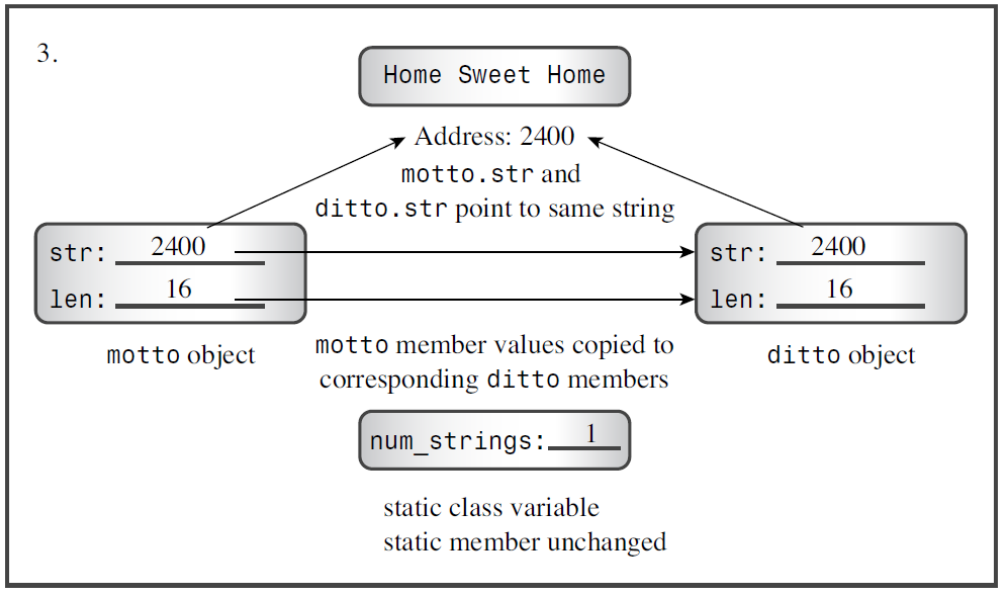
- 首先，输出表明，析构函数的调用次数比构造函数的调用次数多2，原因可能是用默认的复制构造函数(没有输出记录)创建了两个对象
  - 解决办法：增加显式复制构造函数

```
StringBad::StringBad(const String &s)
{
    num_strings++;
    // important stuff to go here
}
```

- 其次，隐式复制构造函数是按值进行复制
  - 隐式复制构造函数，复制的并不是字符串，而是一个指向字符串的指针
  - 释放内存时，多次delete同一个地址导致异常终止
  - 解决方法：进行深度复制(deep copy)




```
2.  String ditto(motto); // default copy constructor
```



# Stringbad的其他问题：赋值运算符

- StringBad metoo = knot;
  - 可能(和具体实现有关)有临时对象生成和赋值=运算
- 对于由默认赋值运算符不合适而导致的问题，解决办法是提供赋值运算符(进行深度复制)定义
  - 由于目标对象可能引用了以前分配的数据，函数应使用delete[]来释放这些数据。
  - 函数应当避免将对象赋给自身；否则，给对象重新赋值前，释放内存操作可能删除对象的内容
    - S0 = S0;
  - 函数返回一个指向调用对象的引用。为了保证以下
    - S0 = S1 = S2;

```
1.  StringBad & StringBad::operator=(const StringBad & st)
2.  {
3.      if (this == &st) // object assigned to itself
4.              return *this; // all done
5.      delete [] str; // free old string

6.      len = st.len;
7.      str = new char [len + 1];// for new string
8.      std::strcpy(str, st.str); // copy the string
9.      return *this; // return reference to invoking object
10. }
```

# 改进后的新string类

# 2 改进后的新string类

- 首先，添加复制构造函数和赋值运算符，使类能够正确管理类对象使用的内存
- 其次，添加一些方法

```
int length() const { return len; }
friend bool operator<(const String &st, const String &st2);
friend bool operator>(const String &st1, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend istream &operator>>(istream &is, String &st);
char &operator[](int i);
const char &operator[](int i) const;
static int HowMany();
```

# 2.1 修订后的默认构造函数

- 默认构造
  - new char[1]; //不能new char，原因为了在析构函数，统一使用 delete[ ]

```
String::String()
{
    len = 0;
    str = new char[1];
    str[0] = '\0'; // default string
}
```

# 2.2 比较成员函数

- 把if/else简化成一个函数

```
bool operator<(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) < 0);
}
bool operator>(const String &st1, const String &st2)
{
    return st2 < st1;
}
bool operator==(const String &st1, const String &st2)
{
    return (std::strcmp(st1.str, st2.str) == 0);
}
```

# 2.3 使用中括号表示法访问字符

- 使用中括号来访问其中的字符：
  - 重载时， C++将区分常量和非常量函数的特征标，因此需要单独提供一个仅供const String对象使用的operator[ ]()版本

```
// read-write char access for non-const String
char &String::operator[](int i)
{
    return str[i];
}

// read-only char access for const String
const char &String::operator[](int i) const
{
    return str[i];
}
```

# 2.4 静态类成员函数

- 静态成员函数不能通过对象调用

```
static int HowMany() { return num_strings; }
int count = String::HowMany(); // 只能通过String::HowMany()这样的方式调用
```

- 由于静态成员函数不与特定的对象相关联
  - 所以，静态成员函数只能使用静态数据成员
  - 如，静态方法HowMany可以访问静态成员num_string，但不能访问str和len

# 2.5 进一步重载赋值运算符

P12.4 string1.h P12.5 string1.cpp P12.6 sayings1.cpp

赋值操作前先要释放内存，并为新字符串分配足够的内存

```
// assign a C string to a String
String &String::operator=(const char *s)
{
    if (str)
        delete[] str;
    len = std::strlen(s);
    str = new char[len + 1];
    std::strcpy(str, s);
    return *this;
}
```

# 在构造函数中使用new时应注意的事项

# 3 在构造函数中使用new时应注意的事项

- new和delete
  - 构造函数new，析构函数delete
  - new/delete，new[ ]/delete[ ]
  - 多构造函数时，不宜new，delete[]混用
- 空指针
  - C++11 提供了关键字nullptr
- 应定义一个复制构造函数，通过深度复制将一个对象初始化为另一个对象
  - String::String(const String & st)
- 应定义一个赋值运算符，通过深度复制将一个对象复制给另一个对象
  - String & String::operator=(const String & st)

# 3.1 应该和不应该

```
1.  String::String()
2.  {
3.      str = "default string"; // oops, no new []
4.      len = std::strlen(str);
5.  }
6.  String::String(const char *s)
7.  {
8.      len = std::strlen(s);
9.      str = new char;       // oops, no []
10.     std::strcpy(str, s); // oops, no room
11. }
12. String::String(const String &st)
13. {
14.     len = st.len;
15.     str = new char[len + 1];       // good, allocate space
16.         std::strcpy(str, st.str); // good, copy value
17. }
```

# 3.2 包含类成员的类的逐成员复制

- 如果对应的数据成员有良好的复制构造函数和复制运算符，则无需单独定义

# 有关返回对象的说明

# 4 有关返回对象的说明

- 当成员函数或独立的函数返回对象时可以返回
  - 指向对象的引用
  - 指向对象的const引用
  - const对象

# 4.1 返回指向const对象的引用

- 返回对象会调用复制构造函数
- 返回引用不会
- const引用需要保持一致

# 4.2 返回指向非const对象的引用

- 重载赋值运算符
  - 效率。返回值用于连续赋值。
  - 不能常量，不能引用
- <<
  - 必须。ostream没有公用的复制构造函数

# 4.3 返回对象

- 局部变量的返回
  - 存在调用复制构造函数来创建被返回的对象的开销，无法避免

```
Vector Vector::operator+(const Vector &b) const
{
    return Vector(x + b.x, y + b.y);
}
```

# 4.4 返回const对象

- 不推荐使用！

# 使用指向对象的指针

# 5 使用指向对象的指针

P12.7 sayings2.cpp

- 指针如果只是指向已有的对象
  - 这些指针并不要求使用 new 来分配内存
  - 也不需要用delete释放内存

# 5.1 再谈new和delete

- 在下述情况下析构函数将被调用
- 局部变量
  - 如果对象是动态变量，则当执行完定义该对象的程序块时，将调用该对象的析构函数
- 全局变量
  - 如果对象是静态变量(外部、静态、静态外部或来自名称空间)，则在程序结束时将调用对象的析构函数
- 指针
  - 如果对象是用new 创建的，则仅当显式使用delete删除对象时，其析构函数才会被调用

```
class Act { ... };
...
Act nice; // external object
...
int main()
{
    Act *pt = new Act; // dynamic object
    {
        Act up; // automatic object
        ...
    }
    delete pt;
    ...
}
```

destructor for automatic object up called when execution reaches end of defining block

destructor for dynamic object *pt called when delete operator applied to the pointer pt

destructor for static object nice called when execution reaches end of entire program

# 指针和对象小结

# 5.2 指针和对象小结

- 使用常规表示法来声明指向对象的指针

```
String* glamour;
```

- 将指针初始化为指向已有的对象

```
String* first= &sayings[0];
```

- 可以使用new来初始化指针，创建一个新的对象

```
String* favorite= new String(sayings[choice]);
```

- 使用new来初始化新创建的对象

```
// invokes default constructor
String *gleep = new String;
// invokes the String(const char*) constructor
String *glop = new String("my my my");
// invokes the String(const String&) constructor
String *favorite = new String(sayings[choice]);
```

- 可以使用->运算符通过指针访问类方法：

```
if (sayings[i] .length() < shortest->length())
```

| | |
|---|---|
| Declaring a pointer to a class object: | `String * glamour;` |
| Initializing a pointer to an existing object: | `String * first = &sayings[0];` (String object) |
| Initializing a pointer using new and the default class constructor: | `String * gleep = new String;` |
| Initializing a pointer using new and the `String(const char*)` class constructor: | `String * glop = new String("my my my");` |
| Initializing a pointer using new and the `String(const String &)` class constructor: | `String * favorite = new String(sayings[choice])` (String object) |
| Using the -> operator to access a class method via a pointer: | `if (sayings[i].length() < shortest->length())` (object; pointer to object) |
| Using the * deferencing operator to obtain an object from a pointer: | `if (sayings[i] < *first)` (object; pointer to object) |

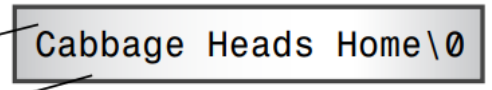
# 指针和对象小结

```
String *pveg = new String("Cabbage Heads Home");
```

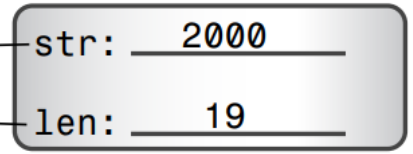
1. Allocate memory for object:


Address: 2400

2. Call class constructor, which
   - allocates space for "Cabbage Heads Home"
   - copies "Cabbage Heads Home" to allocated space
   - assigns address of "Cabbage Heads Home" string to string to `str`
   - assigns value of 19 to len
   - updates num_strings (not shown)




Address: 2000

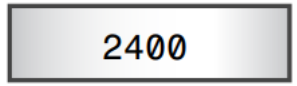



Address: 2400

3. Create the pveg variable:

*pveg* – Address: 2800

4. Assign address of new object to the pveg variable:




*pveg* – Address: 2800

# 5.3 再谈定位new运算符

# 复习各种技术

# 6 复习各种技术

- 进行综合应用练习是必要的

# 6.1 重载<<运算符

- 要重新定义＜＜运算符，以便将它和cout 一起用来显示对象的内

# 6.2 转换函数

- 要将单个值转换为类类型，需要创建原型如下所示的类构造函数

```
c_name(type_name value);
```

- 要将类转换为其他类型，需要创建原型如下所示的类成员函数

```
operator type_name();
```

# 6.3 其构造函数使用new的类

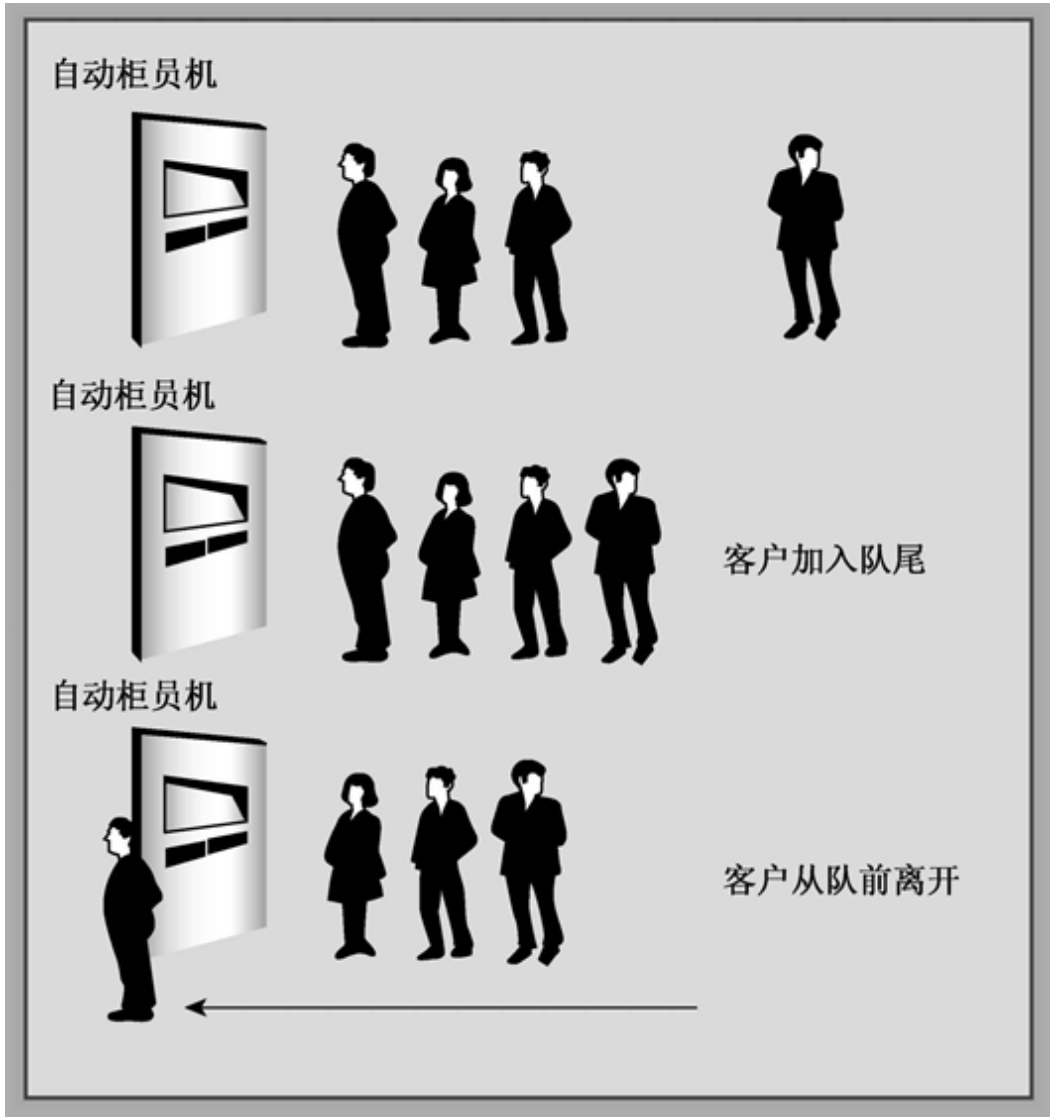
- 如果类使用new 运算符来分配类成员指向的内存，在设计时应采取一些预防措施(编译器并不知道这些规则，因此无法发现错误)
  - 对于指向的内存是由new分配的所有类成员，应在析构函数中对其使用delete释放内存
  - 如析构函数使用delete释放内存，则每个构造函数都应使用new来初始化指针，或设置为空指针
  - 构造函数中和析构函数中的new/delete要配套
  - 应定义一个分配内存(而不是将指针指向已有内存)的复制构造函数
    - 这样程序将能够将类对象初始化为另一个类对象
    - `className(const className &)`
  - 应定义一个重载赋值运算符的类成员函数
    - `c_name & c_name: :operator=(const c_name & en)`

# 队列模拟

# 7 队列模拟

# 7.1 队列类

- Queue特征
  - 队列存储有序的项目序列
  - 队列所能容纳的项目数有一定的限制
  - 应当能够创建空队列
  - 应当能够检查队列是否为空
  - 应当能够检查队列是否是满的
  - 应当能够在队尾添加项目
  - 应当能够从队首删除项目
  - 应当能够确定队列中项目数
- P12.10 queue.h，P12.11 queue.cpp，P12.12 bank.cpp



# 8 总结

- 各种默认函数
- 浅拷贝，深拷贝