附录 D 答　　案

第 1 章

测验

1. 解释器读取源代码并对程序进行翻译，将程序员的代码（程序指令）直接变成操作；编译器将源代码转换成可执行程序，该程序可在以后运行。

2. 随编译器而异。务必参阅编译器自带的文档。

3. 链接器的任务是将编译后的代码与编译器厂商和其他厂商提供的库链接起来。链接器让你能够分块地创建程序，然后将它们链接成一个大程序。

4. 编辑源代码、编译、链接、测试（运行），必要时重复上述步骤。

练习

1. 该程序初始化两个 int 变量，然后打印它们的和（12）与积（35）。

2. 参见编译器手册。

3. 必须在第一行的 include 前加一个#。

4. 该程序将 Hello World 打印到控制台，然后另起一行（回车）。

第 2 章

测验

1. 每次运行编译器时，预处理器都将首先运行。预处理器读取源代码，包含程序员要求包含的文件，执行其他辅助工作。然后编译器运行，将预处理后的源代码转换为目标代码。

2. 因为每当程序被执行时都将自动调用它。它可能不会被其他函数调用，但每个程序都必须有 main( ) 函数。

3. C++风格（单行）注释以两个斜杠（//）打头，将当前行尾之前的所有文本注释掉。C 风格（多行）注释由一对标记（/*和*/）指出，这对标记之间的所有内容都被注释掉。必须确保有匹配的标记对。

4. 可以在 C 风格（多行）注释内嵌套 C++风格（单行）注释，如下所示：

```
/* This marker starts a comment. Everything including
// this single line comment,
is ignored as a comment until the end marker */
```

事实上，只要记住 C++风格注释到行尾结束，就可以在以/*打头的注释嵌套在以//打头的 C++注释中。

5. C 风格（多行）注释可以。如果 C++风格注释超过一行，必须在每行注释开头加上一对斜杠（//）。

练习

1. 下面是解决方案之一：

```
1: #include <iostream>
2: using namespace std;
3: int main()
4: {
5:    cout << "I love C++\n";
6:    return 0;
7: }
```

2. 下面的程序包含一个不执行任何操作的 main()函数，然而，这是一个完整的程序，可以编译、链接和运行。当该程序运行时，好像什么事情也没有发生，因为它不执行任何操作。

```
int main(){}
```

3. 第4行的字符串左边少了一个双引号。

4. 下面是修正后的程序：

```
1: #include <iostream>
2: main()
3: {
4:     std::cout << "Is there a bug here?";
5: }
```

该程序将下述内容打印到屏幕上：

```
Is there a bug here?
```

5. 下面是解决方案之一：

```
 1: #include <iostream>
 2: int Add (int first, int second)
 3: {
 4:     std::cout << "In Add(), received " << first << " and " << second
           << "\n";
 5:     return (first + second);
 6: }
 7:
 8: int Subtract (int first, int second)
 9: {
10:     std::cout << "In Subtract(), received " << first << " and "
        ➥ << second << "\n";
11:     return (first - second);
12: }
13:
14: int main()
15: {
16:     using std::cout;
17:     using std::cin;
18:
19:     cout << "I'm in main()!\n";
20:     int a, b, c;
21:     cout << "Enter two numbers: ";
22:     cin >> a;
23:     cin >> b;
24:
25:     cout << "\nCalling Add()\n";
26:     c=Add(a,b);
```

```
27:     cout << "\nBack in main().\n";
28:     cout << "c was set to " << c;
29:
30:     cout << "\n\nCalling Subtract()\n";
31:     c=Subtract(a,b);
32:     cout << "\nBack in main().\n";
33:     cout << "c was set to " << c;
34:
35:     cout << "\nExiting...\n\n";
36:     return 0;
37: }
```

第 3 章

测验

1. 整型变量是整数，浮点变量是实数，有“浮动”的小数点。浮点数可以用尾数和指数表示。

2. 关键字 unsinged 意味着 int 变量只能为正。在大多数使用 32 位处理器的计算机中，short int 占用 2 字节，而 long int 占用 4 字节。然而，惟一的保证是，long int 不短于 int，而后者不短于 short int。通常，long int 的长度为 short int 的两倍。

3. 符号常量含义直观，常量的名称指出了其含义。另外，可在源代码的某个地方重新定义符号常量，而程序员必须修改使用字面量的所有代码。

4. const 变量的类型是确定的，编译器能够检查使用它们的方式是否正确；另外，经过预处理器处理后，它们仍然存在，因此可以在调试器中使用它们的名称；最重要的是，C++标准不再支持使用#define 来声明常量。

5. 好的变量名指出了变量的用途；糟糕的变量名没有提供任何信息。myAge 和 PeopleOnTheBus 都是好的变量名，而 x、xjk 和 prnd1 可能没有什么用处。

6. BLUE=102

7.

a. 好；

b. 非法；

c. 合法但糟糕；

d. 好；

e. 合法但糟糕。

练习

1. 合适的选择如下：

a. unsigned short int

b. unsigned long int 或 unsigned float

c. unsigned double

d. unsigned short int

2. 下面是可行答案：

a. myAge

b. backYardArea

c. StarsInGalaxy

d. averageRainFall

3. 声明如下：

```
const float PI = 3.14159;
```

4. 下面的代码声明并初始化这样的变量：

```
float myPi = PI;
```

第4章

测验

1. 能返回一个值的任何语句。
2. x=5+7 是一个值为 12 的表达式。
3. 201 / 4 的值为 50。
4. 201 % 4 的值为 1。
5. myAge 为 41，a 为 39，b 为 41。
6. 8+2*3 的值为 14。
7. 前者将 3 赋给 x 并返回 3（被解释为 true）；后者检测 x 是否等于 3，如果是则返回 true，否则返回 flase。
8. 答案如下：

a. false
b. ture
c. true
d. false
e. true

练习

1. 下面是解决方案之一：

```
if (x > y)
      x = y;
else        // y > x || y == x
      y = x;
```

2. 参见练习 3 的答案。
3. 输入 20、10 和 50 时，结果 a 为 20，b 为 30，c 为 10。
第 14 行是赋值而不是检测是否相等。
4. 参见练习 5 的答案。
5. 第 6 行将 a-b 的值赋给 c，而 a-b 为 0。由于 0 被视为 false，因此 if 条件不满足，不打印任何内容。

第5章

测验

1. 函数原型声明函数；函数定义则定义函数。原型以分号结尾，函数定义没有。声明中可以包括关键字 inline 和参数的默认值，而定义不可以。声明中可以不包含参数的名称，定义中必须包括。
2. 不，所有参数都是根据位置而不是名称来识别的。
3. 将函数的返回类型声明为 void。
4. 未显式声明时，函数的返回类型默认为 int。一种良好的编程习惯是，总是明确地声明返回类型。
5. 局部变量是被传入代码块（通常为函数）或在代码块中声明的变量。局部变量仅在代码块中可见。
6. 作用域是指局部变量和全局变量的可见性与生存期，作用域边界通常是由一组大括号指定。
7. 递归通常指函数调用自身的能力。

8. 全局变量通常在多个函数需要访问相同数据时使用。在 C++中很少用全局变量；当你知道如何创建静态类变量后，几乎不会创建全局变量。

9. 函数重载是指编写多个名称相同，但参数数目或类型不同的函数的能力。

练习

1. unsigned long int Perimeter(unsigned short int, unsigned short int);

2. 下面是解决方案之一：

```
unsigned long int Perimeter(unsigned short int length, unsigned short int width)
{
   return (2*length) + (2*width);
}
```

3. 该函数的返回类型被声明为 void，因此不能返回值，但却试图返回一个值。

4. 在该函数的定义中，函数头末尾有一个分号，如果不是这样，它将是正确的。

5. 下面是解决方案之一：

```
short int Divider(unsigned short int valOne, unsigned short int valTwo)
{
      if (valTwo == 0)
            return -1;
      else
            return valOne / valTwo;
}
```

6. 下面是解决方案之一：

```
 1: #include <iostream>
 2: using namespace std;
 3:
 4: short int Divider(
 5:       unsigned short int valOne,
 6:       unsigned short int valTwo);
 7:
 8: int main()
 9: {
10:    unsigned short int one, two;
11:    short int answer;
12:    cout << "Enter two numbers.\n Number one: ";
13:    cin >> one;
14:    cout << "Number two: ";
15:    cin >> two;
16:    answer = Divider(one, two);
17:    if (answer > -1)
18:       cout << "Answer: " << answer;
19:    else
20:       cout << "Error, can't divide by zero!";
21:    return 0;
22: }
23:
24: short int Divider(unsigned short int valOne, unsigned short int
    ➥ valTwo)
25: {
26:    if (valTwo == 0)
```

```
27:        return -1;
28:     else
29:        return valOne / valTwo;
30: }
```

7. 下面是解决方案之一：

```
 1: #include <iostream>
 2: using namespace std;
 3: typedef unsigned short USHORT;
 4: typedef unsigned long ULONG;
 5:
 6: ULONG GetPower(USHORT n, USHORT power);
 7:
 8: int main()
 9: {
10:    USHORT number, power;
11:    ULONG answer;
12:    cout << "Enter a number: ";
13:    cin >> number;
14:    cout << "To what power? ";
15:    cin >> power;
16:    answer = GetPower(number,power);
17:    cout << number << " to the " << power << "th power is " <<
18:      answer << endl;
19:    return 0;
20: }
21:
22: ULONG GetPower(USHORT n, USHORT power)
23: {
24:    if(power == 1)
25:       return n;
26:    else
27:       return (n * GetPower(n,power-1));
28: }
```

第6章

测验

1. 句点运算符是 (.)，可用于访问类或结构的成员。

2. 变量定义分配内存；类声明不分配内存。

3. 类声明是类的接口，告诉类的客户如何与类交互。类实现是成员函数的定义，通常位于相关的 cpp 文件中。

4. 公有数据成员可被类的客户直接访问，私有数据成员只能被类的成员函数访问。

5. 可以。虽然本章没有指出，但成员函数确实可以是私有的。只有类的其他成员函数才能调用私有成员函数。

6. 虽然成员数据可以是公有的，但良好的编程习惯是将其声明为私有的，并提供存取这些数据的公有存取器函数。

7. 可以，每个类对象都有自己的数据成员。

8. 声明在右大括号后以分号结尾，函数定义没有。

9. 在 Cat 类中，不接受任何参数且返回类型为 void 的成员函数 Meow( )的函数头如下：

```
void Cat::Meow()
```

10. 调用构造函数来初始化类。这个特殊函数的名称与类名相同。

练习

1. 下面是解决方案之一：

```
class Employee
{
      int Age;
      int YearsOfService;
      int Salary;
};
```

2. 下面是解决方案之一。注意存取器方法 Get...也被声明为 const 的，因为它们不对对象做任何修改。

```
// Employee.hpp
class Employee
{
   public:
      int  GetAge() const;
      void SetAge(int age);
      int  GetYearsOfService() const;
      void SetYearsOfService(int years);
      int  GetSalary() const;
      void SetSalary(int salary);

   private:
      int itsAge;
      int itsYearsOfService;
      int itsSalary;
};
```

3. 下面是解决方案之一：

```
 1: // Employee.cpp
 2: #include <iostream>
 3: #include "Employee.hpp"
 4:
 5: int  Employee::GetAge() const
 6: {
 7:    return itsAge;
 8: }
 9: void Employee::SetAge(int age)
10: {
11:    itsAge = age;
12: }
13: int  Employee::GetYearsOfService() const
14: {
15:    return itsYearsOfService;
16: }
17: void Employee::SetYearsOfService(int years)
18: {
19:    itsYearsOfService = years;
20: }
```

```
21: int Employee::GetSalary()const
22: {
23:     return itsSalary;
24: }
25: void Employee::SetSalary(int salary)
26: {
27:     itsSalary = salary;
28: }
29:
30: int main()
31: {
32:    using namespace std;
33:
34:     Employee John;
35:     Employee Sally;
36:
37:     John.SetAge(30);
38:     John.SetYearsOfService(5);
39:     John.SetSalary(50000);
40:
41:     Sally.SetAge(32);
42:     Sally.SetYearsOfService(8);
43:     Sally.SetSalary(40000);
44:
45:     cout << "At AcmeSexist company, John and Sally have ";
46:     cout << "the same job.\n\n";
47:
48:     cout << "John is " << John.GetAge() << " years old." << endl;
49:     cout << "John has been with the firm for " ;
50:     cout << John.GetYearsOfService() << " years." << endl;
51:     cout << "John earns $" << John.GetSalary();
52:     cout << " dollars per year.\n\n";
53:
54:     cout << "Sally, on the other hand is " << Sally.GetAge();
55:     cout << " years old and has been with the company ";
56:     cout << Sally.GetYearsOfService();
57:     cout << " years. Yet Sally only makes $" << Sally.GetSalary();
58:     cout << " dollars per year! Something here is unfair.";
59: }
```

4. 下面是正确答案之一：

```
float Employee::GetRoundedThousands() const
{
      return Salary / 1000;
}
```

5. 下面是正确答案之一：

```
class Employee
{
   public:

      Employee(int age, int years, int salary);
```

```
        int  GetAge() const;
        void SetAge(int age);
        int  GetYearsOfService() const;
        void SetYearsOfService(int years);
        int  GetSalary() const;
        void SetSalary(int salary);

    private:
        int itsAge;
        int itsYearsOfService;
        int itsSalary;
};
```

6. 声明必须以分号结尾。

7. 存取器函数 GetAge( )是私有的。记住，除非特别声明，否则所有类成员都是私有的。

8. 不能直接访问 itsStation，因为它是私有的；不能对类调用 SetStation( )，只能对对象调用 SetStation( )；不能初始化 myOtherTV，因为没有匹配的构造函数。

第 7 章

测验

1. 用逗号将初始化分开，如：

```
for (x = 0, y = 10; x < 100; x++, y++),
```

2. goto 可以沿任何方向跳到任何一行代码，这使源代码难以理解，进而难以维护。

3. 可以。如果初始化后条件为 false，则 for 循环体将不会执行。下面是一个例子：

```
for (int x = 100; x < 100; x++)
```

4. 变量 x 不在作用域中，因此没有有效的值。

5. 可以，任何循环都可嵌套在其他循环中。

6. 可以，下面是 for 循环和 while 循环的例子：

```
for(;;)
{
      // This for loop never ends!
}
while(true)
{
      // This while loop never ends!
}
```

7. 程序将好像被挂起，因为它永远不会结束。这导致你必须重新启动计算机或使用操作系统的高级特性来结束任务。

练习

1. 下面是正确答案之一：

```
for (int i = 0; i< 10; i++)
{
      for ( int j = 0; j< 10; j++)
            cout << "0";
      cout << endl;
```

```
}
```

2. 下面是正确答案之一：

```
for (int x = 100; x<=200; x+=2)
```

3. 下面是正确答案之一：

```
int x = 100;
while (x <= 200)
    x+= 2;
```

4. 下面是正确答案之一：

```
int x = 100;
do
{
    x+=2;
} while (x <= 200);
```

5. counter 没有递增，while 循环永远不会结束。

6. 循环语句后有分号，因此该循环不执行任何操作。程序员的意图也许就是这样的，但如果本意是想通过 counter 打印每个值，将达不到目的，而只打印 for 循环结束后 counter 的值。

7. counter 被初始化为 100，但测试条件却为 counter 是否小于 10，因此不满足条件，循环体永远不会执行。如果将第 1 行改为 int counter=5;，循环将在数到最小的 int 变量取值时结束。int 默认为带符号的，这不是程序员的初衷。

8. case 0 可能需要一条 break 语句。如果没有，应通过注释加以说明。

第 8 章

测验

1. 地址运算符（&）。

2. 解除引用运算符（*）。

3. 指针是一个变量，它存储的是另一个变量的地址。

4. 指针存储的地址是另一个变量的地址。存在该地址中的值是任何变量的值。间接运算符（*）返回存储在指定地址中的值，而地址本身存储在指针中。

5. 间接运算符返回指针指向的地址中的值，地址运算符（&）返回变量的地址。

6. 前者将 ptrOne 声明为一个指向 int 常量的指针，使用该指针不能修改 int 常量的值；后者将 ptrTwo 声明为指向 int 变量的常量指针，初始化该指针后便不能给它重新赋值。

练习

1.

a. int *pOne;声明一个 int 指针

b. int vTwo;声明一个 int 变量

c. int * pThree=&vTwo;声明一个 int 指针并将其初始化为变量 vTwo 的地址。

2. unsigned short *pAge = &yourAge;

3. *pAge = 50;

4. 下面是解决方案之一：

```
1:  #include <iostream>
2:
3:  int main()
```

```
 4: {
 5:    int theInteger;
 6:    int *pInteger = &theInteger;
 7:    *pInteger = 5;
 8:
 9:    std::cout << "The Integer: "
10:              << *pInteger << std::endl;
11:
12:    return 0;
13: }
```

5. 应初始化 pInt。更重要的是，由于它没有被初始化，没有将任何内存地址赋给它，因此指向内存中的一个随机位置。将字面量 9 存储到这个随机位置中是一种非常危险的错误。

6. 程序员的意图可能是将 9 赋给 pVar 指向的变量 SomeVariable，然而，由于遗漏了间接运算符（*），实际上却将 9 赋给了 pVar。如果使用 pVar 来赋值将导致灾难，因为它指向地址 9，而不是变量 SomeVariable。

第 9 章

测验

1. 引用是别名，而指针是存储地址的变量。引用不能为空，也不能给它赋值。
2. 需要重新赋值或指向的目标可能为空时。
3. 空指针（0）。
4. 指向常量对象的引用的简称。
5. 按引用传递意味着不创建局部拷贝。可通过按引用或指针传递来实现。
6. 所有 3 种方式都是正确的；然而，必须选择一种方式并一直使用它。

练习

1. 下面是正确答案之一：

```
 1: //Exercise 9.1 -
 2: #include <iostream>
 3:
 4: int main()
 5: {
 6:    int  varOne = 1;    // sets varOne to 1
 7:    int& rVar = varOne;
 8:    int* pVar = &varOne;
 9:    rVar = 5;           // sets varOne to 5
10:    *pVar = 7;          // sets varOne to 7
11:
12:    // All three of the following will print 7:
13:    std::cout << "variable:  " << varOne << std::endl;
14:    std::cout << "reference: " << rVar  << std::endl;
15:    std::cout << "pointer:   " << *pVar  << std::endl;
16:
17:    return 0;
18: }
```

2. 下面是正确答案之一：

```
 1: int main()
 2: {
```

```
 3:    int varOne;
 4:    const int * const pVar = &varOne;
 5:    varOne = 6;
 6:    *pVar = 7;
 7:    int varTwo;
 8:    pVar = &varTwo;
 9:    return 0;
10: }
```

3. 不能给常量对象赋值，不能重新给常量指针赋值。这意味着第6和8行有问题。

4. 下面是一种答案。注意，由于迷失指针，运行该程序将是危险的。

```
1: int main()
2: {
3:    int * pVar;
4:    *pVar = 9;
5:    return 0;
6: }
```

5. 下面是一种答案：

```
1: int main()
2: {
3:    int VarOne;
4:    int * pVar = &varOne;
5:    *pVar = 9;
6:    return 0;
7: }
```

6. 下面是一种答案。注意，在程序中应避免内存泄漏。

```
 1: #include <iostream>
 2: int FuncOne();
 3: int main()
 4: {
 5:    int localVar = FunOne();
 6:    std::cout << "The value of localVar is: " << localVar;
 7:    return 0;
 8: }
 9:
10: int FuncOne()
11: {
12:    int * pVar = new int (5);
13:    return *pVar;
14: }
```

7. 下面是解决方案之一：

```
1: #include <iostream>
2: void FuncOne();
3: int main()
4: {
5:    FuncOne();
6:    return 0;
7: }
8:
```

```
 9: void FuncOne()
10: {
11:    int * pVar = new int (5);
12:    std::cout << "The value of *pVar is: " << *pVar ;
13:    delete pVar;
14: }
```

8. MakeCat 返回一个指向自由存储区中创建的 CAT 对象的引用。没有办法释放这些内存，这将导致内存泄漏。

9. 下面是一种解决方案：

```
 1:    #include <iostream>
 2:    using namespace std;
 3:    class CAT
 4:    {
 5:      public:
 6:        CAT(int age) { itsAge = age; }
 7:        ~CAT(){}
 8:        int GetAge() const { return itsAge;}
 9:      private:
10:        int itsAge;
11:    };
12:
13:    CAT * MakeCat(int age);
14:    int main()
15:    {
16:      int age = 7;
17:      CAT * Boots = MakeCat(age);
18:      cout << "Boots is " << Boots->GetAge() << " years old";
19:      delete Boots;
20:     return 0;
21:    }
22:
23:    CAT * MakeCat(int age)
24:    {
25:      return new CAT(age);
26:    }
```

第 10 章

测验

1. 重载的成员函数是同一个类中名称相同但参数数目或类型不同的函数。

2. 定义分配内存，而声明不。几乎所有的声明都是定义，重要的例外是类声明、函数原型和 typedef 语句。

3. 创建对象的临时拷贝以及按值传递对象时。

4. 每当以对象超出作用域或对指向对象的指针调用 delete，导致对象被销毁时，都将调用析构函数。

5. 赋值运算符用于已有的对象，复制构造函数创建一个新的对象。

6. this 指向对象本身，它是每个成员函数的一个隐含参数。

7. 前缀运算符不接受参数；后缀运算符接受一个 int 参数，它用于告诉编译器，这是后缀运算符。

8. 不。不能为内置类型重载任何运算符。

9. 合法但不是个好主意。运算符应以任何人都能够理解的方式重载。
10. 没有。与构造函数和析构函数一样，它们没有返回值。

练习

1. 下面是一种答案：

```
class SimpleCircle
{
    public:
        SimpleCircle();
        ~SimpleCircle();
        void SetRadius(int);
        int GetRadius();
    private:
        int itsRadius;
};
```

2. 下面是一种答案：

```
SimpleCircle::SimpleCircle():
itsRadius(5)
{}
```

3. 下面是一种答案：

```
SimpleCircle::SimpleCircle(int radius):
itsRadius(radius)
{}
```

4. 下面是一种答案：

```
const SimpleCircle& SimpleCircle::operator++()
{
      ++(itsRadius);
      return *this;
}

// Operator ++(int) postfix.
// Fetch then increment
const SimpleCircle SimpleCircle::operator++ (int)
{
      // declare local SimpleCircle and initialize to value of *this
      SimpleCircle temp(*this);
      ++(itsRadius);
      return temp;
}
```

5. 下面是一种答案：

```
class SimpleCircle
{
    public:
        SimpleCircle();
        SimpleCircle(int);
        ~SimpleCircle();
        void SetRadius(int);
        int GetRadius();
```

```
        const SimpleCircle& operator++();
        const SimpleCircle operator++(int);
    private:
        int *itsRadius;
};

SimpleCircle::SimpleCircle()
{
    itsRadius = new int(5);
}

SimpleCircle::SimpleCircle(int radius)
{
    itsRadius = new int(radius);
}

const SimpleCircle& SimpleCircle::operator++()
{
     ++(*itsRadius);
     return *this;
}

// Operator ++(int) postfix.
// Fetch then increment
const SimpleCircle SimpleCircle::operator++ (int)
{
    // declare local SimpleCircle and initialize to value of *this
    SimpleCircle temp(*this);
    ++(*itsRadius);
    return temp;
}
```

6. 下面是一种答案：

```
SimpleCircle::SimpleCircle(const SimpleCircle & rhs)
{
    int val = rhs.GetRadius();
    itsRadius = new int(val);
}
```

7. 下面是一种答案：

```
SimpleCircle& SimpleCircle::operator=(const SimpleCircle & rhs)
{
     if (this == &rhs)
          return *this;
     delete itsRadius;
     itsRadius = new int;
     *itsRadius = rhs.GetRadius();
     return *this;
}
```

8. 下面是一种解决方案：

```
1: #include <iostream>
```

```
 2: using namespace std;
 3:
 4: class SimpleCircle
 5: {
 6:   public:
 7:      // constructors
 8:     SimpleCircle();
 9:     SimpleCircle(int);
10:     SimpleCircle(const SimpleCircle &);
11:     ~SimpleCircle() {}
12:
13:     // accessor functions
14:     void SetRadius(int);
15:     int GetRadius()const;
16:
17:     // operators
18:     const SimpleCircle& operator++();
19:     const SimpleCircle operator++(int);
20:     SimpleCircle& operator=(const SimpleCircle &);
21:
22:   private:
23:     int *itsRadius;
24: };
25:
26:
27: SimpleCircle::SimpleCircle()
28: {itsRadius = new int(5);}
29:
30: SimpleCircle::SimpleCircle(int radius)
31: {itsRadius = new int(radius);}
32:
33: SimpleCircle::SimpleCircle(const SimpleCircle & rhs)
34: {
35:     int val = rhs.GetRadius();
36:     itsRadius = new int(val);
37: }
38:
39: SimpleCircle& SimpleCircle::operator=(const SimpleCircle & rhs)
40: {
41:     if (this == &rhs)
42:         return *this;
43:     *itsRadius = rhs.GetRadius();
44:     return *this;
45: }
46:
47: const SimpleCircle& SimpleCircle::operator++()
48: {
49:     ++(*itsRadius);
50:     return *this;
51: }
52:
53: // Operator ++(int) postfix.
```

```
54:  // Fetch then increment
55:  const SimpleCircle SimpleCircle::operator++ (int)
56:  {
57:     // declare local SimpleCircle and initialize to value of *this
58:     SimpleCircle temp(*this);
59:     ++(*itsRadius);
60:     return temp;
61:  }
62:  int SimpleCircle::GetRadius() const
63:  {
64:      return *itsRadius;
65:  }
66:  int main()
67:  {
68:      SimpleCircle CircleOne, CircleTwo(9);
69:      CircleOne++;
70:      ++CircleTwo;
71:      cout << "CircleOne: " << CircleOne.GetRadius() << endl;
72:      cout << "CircleTwo: " << CircleTwo.GetRadius() << endl;
73:      CircleOne = CircleTwo;
74:      cout << "CircleOne: " << CircleOne.GetRadius() << endl;
75:      cout << "CircleTwo: " << CircleTwo.GetRadius() << endl;
76:      return 0;
77:  }
```

9. 必须检查 rhs 是否等于 this，否则执行 a = a 操作可能导致程序崩溃。

10. operator+修改一个操作数的值，而不是用它们的和创建一个新的 VeryShort 对象。正确的方法如下：

```
VeryShort  VeryShort::operator+ (const VeryShort& rhs)
{
     return VeryShort(itsVal + rhs.GetItsVal());
}
```

第 11 章

测验

1. 过程化编程将重点放在与数据分开的函数上。面向对象编程将数据和功能集成到对象中，其重点是对象间的交互。

2. 面向对象分析与设计包括概念化（一个描述想法的简单句子）、分析（了解需求的过程）和设计（创建类模型并据此创建代码的过程）。

3. 封装是将分立实体中所有的数据和功能放到类中的特性。

4. 域是要创建的产品将用于的业务领域。

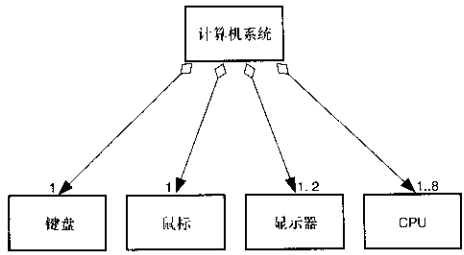
5. 参与者是位于要开发的系统外部并与之交互的人或系统。

6. 用例是有关软件将被如何使用的描述，描述了系统本身和参与者之间的交互。

7. A 正确，B 错误。

练习

1. 下图提供了一种解决方案：



2. 小汽车、摩托车、卡车、自行车、行人和应急车辆都要使用这个路口。另外有一个交通灯表示行人走/停。

路面应包括在模拟中吗？当然，路面质量将影响交通，但作为首次设计，不考虑该因素将更简单。

第一个对象可能是路口本身。路口对象可能应包含每个方向等待通过的车辆列表和等待穿过人行横道的行人列表。还应有一些方法可选择哪些以及多少车辆和行人可通过路口。

仅有一个路口，因此可能要考虑如何确保只能实例化一个这样的对象（提示：考虑使用静态方法和保护访问权限）。

行人和车辆都是路口的客户。它们有很多相同的特征：它们可以在任何时间出现，可以有任意数量，都必须等待信号灯（尽管前进线路不同）。这表明应考虑为行人和车辆创建一个共同的基类。

因此需要使用的类可能包括：

```
class Entity;          // a client of the intersection
class Vehicle : Entity...;     // the root of all cars, trucks, bicycles and
➥ // emergency vehicles.
class Pedestrian : Entity...;  // the root of all People
class Car : public Vehicle...;
class Truck : public Vehicle...;
class Motorcycle : public Vehicle...;
class Bicycle : public Vehicle...;
class Emergency_Vehicle : public Vehicle...;
class Intersection;    // contains lists of cars and people waiting to pass
```

3. 需要为该项目编写两个独立的程序：用户运行的客户端程序以及在一台独立的机器上运行的服务器。另外，客户机需要有一个管理组件，让系统管理员能够新增人员和会议室。

如果决定以客户/服务器模型来实现该项目，客户端程序将接受用户输入并向服务器发出请求。服务器处理请求并将结果发回给客户端程序。采用这种模型时，许多人可以同时安排会议。

在客户端，除管理模块外还有两个主要的子系统：用户界面子系统和通信子系统。服务器由3个主要的子系统组成：通信子系统、日程安排子系统和邮件接口。在日程安排更改后，邮件接口将通知用户。

4. 会议被定义为一组人员占用会议室一段时间。安排会议的人可能想指定会议室和时间，但调度程序必须知道会议将持续多长时间以及与会人员。

对象可能包括系统用户和会议室，别忘了包括日历类，可能还有Meeting类，用于封装会议的所有已知信息。

类原型可能包括：

```
class Calendar_Class;          // forward reference
class Meeting;                 // forward reference
class Configuration
{
   public:
      Configuration();
```

```
        ~Configuration();
        Meeting Schedule( ListOfPerson&, Delta Time duration );
        Meeting Schedule( ListOfPerson&, Delta Time duration, Time );
        Meeting Schedule( ListOfPerson&, Delta Time duration, Room );
        ListOfPerson&    People();   // public accessors
        ListOfRoom&    Rooms();      // public accessors
    protected:
        ListOfRoom    rooms;
        ListOfPerson    people;
};
typedef long     Room_ID;
class Room
{
    public:
        Room( String name, Room_ID id, int capacity,
              String directions = "", String description = "" );
        ~Room();
        Calendar_Class Calendar();

    protected:
        Calendar_Class    calendar;
        int         capacity;
        Room_ID     id;
        String         name;
        String         directions;        // where is this room?
        String         description;
};
typedef long Person_ID;
class Person
{
    public:
        Person( String name, Person_ID id );
        ~Person();
        Calendar_Class Calendar();        // the access point to add meetings
    protected:
        Calendar_Class    calendar;
        Person_ID    id;
        String         name;
};
class Calendar_Class
{
    public:
        Calendar_Class();
        ~Calendar_Class();

        void Add( const Meeting& );       // add a meeting to the calendar
        void Delete( const Meeting& );
        Meeting* Lookup( Time );          // see if there is a meeting at the
                                          // given time

        Block( Time, Duration, String reason = "" );
    // allocate time to yourself...
```

```
    protected:
        OrderedListOfMeeting meetings;
};
class Meeting
{
    public:
        Meeting( ListOfPerson&, Room room,
              Time when, Duration duration, String purpose = "" );
        ~Meeting();
    protected:
        ListOfPerson  people;
        Room          room;
        Time          when;
        Duration      duration;
        String        purpose;
};
```

可以使用 private 而不是 protected。保护成员将在第 12 章介绍。

第 12 章

测验

1. 虚函数表是 C++编译器管理虚函数的一种常用方式。该表记录了所有虚函数的地址，根据指向的对象的运行阶段类型调用正确的函数。
2. 任何类的析构函数都可声明为虚的。对指针调用 delete 时，将确定被指向对象的运行阶段类型，据此调用正确的派生类析构函数。
3. 这是一个假问题：没有虚构造函数。
4. 在类中创建一个虚方法，该方法调用复制构造函数。
5. Base::FunctionName( );
6. FunctionName( );
7. 是的，虚拟性将被继承，不能撤销。
8. 派生类的成员函数可以访问基类的保护成员。

练习

1. virtual void SomeFunction(int);
2.

```
class Square : public Rectangle
{};
```

3.

```
Square::Square(int length):
      Rectangle(length, width){}
```

4. 下面是解决方案之一：

```
class Square
{
    public:
        // ...
        virtual Square * clone() const { return new Square(*this); }
```

```
        // ...
};
```

5. 也许没有错。SomeFumction 需要一个 Shape 对象，已经传递给它一个从 Rectangle“切成”的 Shape。只要不需要 Rectangle 部分，这就是可行的。如果需要 Rectangle 部分，则必须修改 SomeFunction，使之接受一个 Shape 指针或 Shape 引用作为参数。

6. 不能将复制构造函数声明为虚函数。

第 13 章

测验

1. SomeArray[0]和 SomeArray[24]。

2. 为每一维提供一个下标。例如，SomeArray[2][3][2]是一个三维数组，第一维包含 2 个元素，第二维包含 3 个元素，第三维包含 2 个元素。

3. SomeArray[2][3][2] = { { {1,2},{3,4},{5,6} } , { {7,8},{9,10},{11,12} } };

4. 10×5×20 = 1000。

5. 数组和链表都是用于存储信息的容器，然而链表将信息连接在一起。

6. 该字符串包含 16 个字符：读者看到的 15 个和末尾的一个空字符。

7. 空字符。

练习

1. 下面是一种可能的解决方案。数组名可以不同，但为存储 3×3 的棋盘，数组名后面必须是[3][3]。

```
int GameBoard[3][3];
```

2. int GameBoard[3][3] = { {0,0,0},{0,0,0},{0,0,0} }

3. 下面是一种解决方案，这里使用了函数 strcpy( )和 strlen( )。

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

int main()
{
     char firstname[] = "Alfred";
     char middlename[] = "E";
     char lastname[] = "Numan";
     char fullname[80];
     int  offset = 0;
     strcpy(fullname,firstname);
     offset = strlen(firstname);
     strcpy(fullname+offset," ");
     offset += 1;
     strcpy(fullname+offset,middlename);
     offset += strlen(middlename);
     strcpy(fullname+offset,". ");
     offset += 2;
     strcpy(fullname+offset,lastname);

     cout << firstname << "-" << middlename << "-"
          << lastname << endl;
```

```
    cout << "Fullname: " << fullname << endl;

    return 0;
}
```

4. 该数组的元素为5×4，但代码初始化的却是4×5。

5. 编程者的本意是 i<5，却写成了 i<=5。循环在 i==5 和 j==4 时仍将运行，但数组中并没有元素 SomeArray[5][4]。

第 14 章

测验

1. 向下转换是将基类指针视为派生类指针。

2. 这指的是将共有的功能向上移到共同的基类中。如果多个类都有某个函数，应寻找它们共同的基类，将该函数放在该基类中。

3. 如果声明时没有使用关键字 virtual，将创建两个 Shapes：一个属于 Rectangle，另一个属于 Shape。如果声明类时在这两个类名前使用了关键字 virtual，将只创建一个 Shape。

4. Horse 和 Bird 都在其构造函数中初始化基类 Animal，Pegasus 也如此。创建 Pegasus 对象时，Horse 和 Bird 对 Animal 的初始化将被忽略。

5. 下面是一种答案：

```
class Vehicle
{
      virtual void Move() = 0;
}
```

6. 都不需要覆盖，除非要使类成为非抽象的，在这种情况下都必须覆盖。

练习

1. class JetPlane : public Rocket, public Airplane

2. class Seven47: public JetPlane

3. 下面是一种答案：

```
class Vehicle
{
      virtual void Move() = 0;
      virtual void Haul() = 0;
};

class Car : public Vehicle
{
      virtual void Move();
      virtual void Haul();
};

class Bus : public Vehicle
{
      virtual void Move();
      virtual void Haul();
};
```

4. 下面是一种答案：

```
class Vehicle
{
      virtual void Move() = 0;
      virtual void Haul() = 0;
};

class Car : public Vehicle
{
      virtual void Move();
};

class Bus : public Vehicle
{
      virtual void Move();
      virtual void Haul();
};

class SportsCar : public Car
{
      virtual void Haul();
};

class Coupe : public Car
{
      virtual void Haul();
};
```

第 15 章

测验

1. 可以。它们是成员变量，可以像其他成员变量那样控制其访问权限。如果它们是私有的，则只能通过使用成员函数（或更常见的是静态成员函数）来访问。

2. static int itsStatic;

3. static int SomeFunction();

4. long (* function)(int);

5. long (Car::*function)(int);

6. long (Car::*function)(int) theArray [10];

练习

1. 下面是一种答案：

```
0:    // Ex1501.cpp
1:    class myClass
2:    {
3:      public:
4:        myClass();
5:        ~myClass();
6:      private:
7:        int itsMember;
```

```
 8:       static int itsStatic;
 9:    };
10:
11:   myClass::myClass():
12:    itsMember(1)
13:   {
14:     itsStatic++;
15:   }
16:
17:   myClass::~myClass()
18:   {
19:     itsStatic--;
20:   }
21:
22:   int myClass::itsStatic = 0;
23:
24:   int main()
25:   {
26:     // do something
27:     return 0;
28:   }
```

2. 下面是一种答案：

```
 0:   // Ex1502.cpp
 1:   #include <iostream>
 2:   using namespace std;
 3:   class myClass
 4:   {
 5:     public:
 6:       myClass();
 7:       ~myClass();
 8:       void ShowMember();
 9:       void ShowStatic();
10:     private:
11:       int itsMember;
12:       static int itsStatic;
13:  };
14:
15:   myClass::myClass():
16:     itsMember(1)
17:   {
18:     itsStatic++;
19:   }
20:
21:   myClass::~myClass()
22:   {
23:     itsStatic--;
24:     cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:   }
26:
27:   void myClass::ShowMember()
28:   {
```

```
29:     cout << "itsMember: " << itsMember << endl;
30:   }
31:
32:   void myClass::ShowStatic()
33:   {
34:     cout << "itsStatic: " << itsStatic << endl;
35:   }
36:   int myClass::itsStatic = 0;
37:
38:   int main()
39:   {
40:     myClass obj1;
41:     obj1.ShowMember();
42:     obj1.ShowStatic();
43:
44:     myClass obj2;
45:     obj2.ShowMember();
46:     obj2.ShowStatic();
47:
48:     myClass obj3;
49:     obj3.ShowMember();
50:     obj3.ShowStatic();
51:    return 0;
52:   }
```

3. 下面是一种答案：

```
 0:   // Ex1503.cpp
 1:   #include <iostream>
 2:   using namespace std;
 3:   class myClass
 4:   {
 5:     public:
 6:       myClass();
 7:       ~myClass();
 8:       void ShowMember();
 9:       static int GetStatic();
10:     private:
11:       int itsMember;
12:       static int itsStatic;
13:   };
14:
15:   myClass::myClass():
16:    itsMember(1)
17:   {
18:     itsStatic++;
19:   }
20:
21:   myClass::~myClass()
22:   {
23:     itsStatic--;
24:     cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:   }
```

```
26:
27:  void myClass::ShowMember()
28:  {
29:     cout << "itsMember: " << itsMember << endl;
30:  }
31:
32:  int myClass::itsStatic = 0;
33:
34:  int myClass::GetStatic()
35:  {
36:     return itsStatic;
37:  }
38:
39:  int main()
40:  {
41:     myClass obj1;
42:     obj1.ShowMember();
43:     cout << "Static: " << myClass::GetStatic() << endl;
44:
45:     myClass obj2;
46:     obj2.ShowMember();
47:     cout << "Static: " << myClass::GetStatic() << endl;
48:
49:     myClass obj3;
50:     obj3.ShowMember();
51:     cout << "Static: " << myClass::GetStatic() << endl;
52:     return 0;
53:  }
```

4. 下面是一种答案：

```
 0:  // Ex1504.cpp
 1:  #include <iostream>
 2:  using namespace std;
 3:  class myClass
 4:  {
 5:     public:
 6:       myClass();
 7:       ~myClass();
 8:       void ShowMember();
 9:       static int GetStatic();
10:     private:
11:       int itsMember;
12:       static int itsStatic;
13:  };
14:
15:  myClass::myClass():
16:    itsMember(1)
17:  {
18:     itsStatic++;
19:  }
20:
21:  myClass::~myClass()
```

```
22:   {
23:     itsStatic--;
24:     cout << "In destructor. ItsStatic: " << itsStatic << endl;
25:   }
26:
27:   void myClass::ShowMember()
28:   {
29:     cout << "itsMember: " << itsMember << endl;
30:   }
31:
32:   int myClass::itsStatic = 0;
33:
34:   int myClass::GetStatic()
35:   {
36:     return itsStatic;
37:   }
38:
39:   int main()
40:   {
41:     void (myClass::*PMF) ();
42:
43:     PMF=myClass::ShowMember;
44:
45:     myClass obj1;
46:     (obj1.*PMF)();
47:     cout << "Static: " << myClass::GetStatic() << endl;
48:
49:     myClass obj2;
50:     (obj2.*PMF)();
51:     cout << "Static: " << myClass::GetStatic() << endl;
52:
53:     myClass obj3;
54:     (obj3.*PMF)();
55:     cout << "Static: " << myClass::GetStatic() << endl;
56:     return 0;
57:   }
```

5. 下面是一种答案：

```
0:   // Ex1505.cpp
1:   #include <iostream>
2:   using namespace std;
3:   class myClass
4:   {
5:     public:
6:       myClass();
7:       ~myClass();
8:       void ShowMember();
9:       void ShowSecond();
10:      void ShowThird();
11:      static int GetStatic();
12:    private:
13:      int itsMember;
```

```
14:       int itsSecond;
15:       int itsThird;
16:       static int itsStatic;
17:   };
18:
19:   myClass::myClass():
20:     itsMember(1),
21:     itsSecond(2),
22:     itsThird(3)
23:   {
24:     itsStatic++;
25:   }
26:
27:   myClass::~myClass()
28:   {
29:     itsStatic--;
30:     cout << "In destructor. itsStatic: " << itsStatic << endl;
31:   }
32:
33:   void myClass::ShowMember()
34:   {
35:     cout << "itsMember: " << itsMember << endl;
36:   }
37:
38:   void myClass::ShowSecond()
39:   {
40:     cout << "itsSecond: " << itsSecond << endl;
41:   }
42:
43:   void myClass::ShowThird()
44:   {
45:     cout << "itsThird: " << itsThird << endl;
46:   }
47:   int myClass::itsStatic = 0;
48:
49:   int myClass::GetStatic()
50:   {
51:     return itsStatic;
52:   }
53:
54:   int main()
55:   {
56:     void (myClass::*PMF) ();
57:
58:     myClass obj1;
59:     PMF=myClass::ShowMember;
60:     (obj1.*PMF)();
61:     PMF=myClass::ShowSecond;
62:     (obj1.*PMF)();
63:     PMF=myClass::ShowThird;
64:     (obj1.*PMF)();
65:     cout << "Static: " << myClass::GetStatic() << endl;
```

```
66:
67:     myClass obj2;
68:     PMF=myClass::ShowMember;
69:     (obj2.*PMF)();
70:     PMF=myClass::ShowSecond;
71:     (obj2.*PMF)();
72:     PMF=myClass::ShowThird;
73:     (obj2.*PMF)();
74:     cout << "Static: " << myClass::GetStatic() << endl;
75:
76:     myClass obj3;
77:     PMF=myClass::ShowMember;
78:     (obj3.*PMF)();
79:     PMF=myClass::ShowSecond;
80:     (obj3.*PMF)();
81:     PMF=myClass::ShowThird;
82:     (obj3.*PMF)();
83:     cout << "Static: " << myClass::GetStatic() << endl;
84:
85:     return 0;
86: }
```

第 16 章

测验

1. 使用公有继承性。

2. 使用聚合（包含），即一个类包含一个这样的成员：它是另一个类的对象。

3. 聚合描述的是一个类将另一个类的对象作为其数据成员的概念；代理描述的是一个类用另一个类来完成任务和目标的概念。

4. 代理描述的是一个类用另一个类来完成任务和目标的概念；以...的方式实现指的是继承另一个类的实现。

5. 友元函数是被声明为能够访问类的保护和私有成员的函数。

6. 友元类是被声明为这样的类：其所有成员函数都是另一个类的友元函数。

7. 不是，友元关系不可交换。

8. 不是，友元关系不可继承。

9. 不是，友元关系不可传递。

10. 可在类声明的任何地方。将友元函数声明放在 public、protected 还是 private 部分没有任何区别。

练习

1. 下面是一种答案：

```
class Animal:
{
    private:
        String itsName;
};
```

2. 下面是一种答案：

```
class boundedArray : public Array
```

```
{
    //...
}
```

3. 下面是一种答案：

```
class Set : private Array
{
    // ...
}
```

4. 下面是一种答案：

```
#include <iostream.h>
#include <string.h>

class String
{
  public:
     // constructors
     String();
     String(const char *const);
     String(const String &);
     ~String();

     // overloaded operators
     char & operator[](int offset);
     char operator[](int offset) const;
     String operator+(const String&);
     void operator+=(const String&);
     String & operator= (const String &);
     friend ostream& operator<<( ostream&
                  theStream,String& theString);
     friend istream& operator>>( istream&
         theStream,String& theString);
     // General accessors
     int GetLen()const { return itsLen; }
     const char * GetString() const { return itsString; }
     // static int ConstructorCount;

  private:
     String (int);        // private constructor
     char * itsString;
     unsigned short itsLen;

};

ostream& operator<<( ostream& theStream,String& theString)
{
   theStream << theString.GetString();
   return theStream;
}

istream& operator>>( istream& theStream,String& theString)
```

```
41:   {
42:      theStream >> theString.GetString();
43:      return theStream;
44:   }
45:
46:   int main()
47:   {
48:     String theString("Hello world.");
49:     cout << theString;
50:     return 0;
51:   }
```

5. 不能将友元声明放在函数中，而必须将函数声明为类的友元。

6. 下面是修改后的程序清单：

```
 0:    Bug Busters
 1:    #include <iostream>
 2:    using namespace std;
 3:    class Animal;
 4:
 5:    void setValue(Animal& , int);
 6:
 7:    class Animal
 8:    {
 9:      public:
10:        friend void setValue(Animal&, int);
11:        int GetWeight()const { return itsWeight; }
12:        int GetAge() const { return itsAge; }
13:      private:
14:        int itsWeight;
15:        int itsAge;
16:   };
17:
18:   void setValue(Animal& theAnimal, int theWeight)
19:   {
20:     theAnimal.itsWeight = theWeight;
21:   }
22:
23:   int main()
24:   {
25:     Animal peppy;
26:     setValue(peppy,5);
27:     return 0;
28:   }
```

7. 函数 setValue(Animal&, int)被声明为友元，但 setValue(Animal&, int, int)没有被声明为友元。

8. 下面是修改后的程序清单：

```
 0:    // Bug Busters
 1:    #include <iostream>
 2:    using namespace std;
 3:    class Animal;
 4:
 5:    void setValue(Animal& , int);
```

```
 6:    void setValue(Animal& ,int,int); // here's the change!
 7:
 8:    class Animal
 9:    {
10:      friend void setValue(Animal& ,int);
11:      friend void setValue(Animal& ,int,int);
12:      private:
13:         int itsWeight;
14:         int itsAge;
15:    };
16:
17:    void setValue(Animal& theAnimal, int theWeight)
18:    {
19:       theAnimal.itsWeight = theWeight;
20:    }
21:
22:   void setValue(Animal& theAnimal, int theWeight, int theAge)
23:   {
24:      theAnimal.itsWeight = theWeight;
25:      theAnimal.itsAge = theAge;
26:   }
27:
28:   int main()
29:   {
30:      Animal peppy;
31:      setValue(peppy,5);
32:      setValue(peppy,7,9);
33:      return 0;
34:   }
```

第 17 章

测验

1. 插入运算符（<<）是 ostream 对象的一个成员运算符，用于写入输出设备。

2. 提取运算符（>>）是 istream 对象的一个成员运算符，用于写入程序的变量。

3. cin.get( )的第一种形式没有参数，它返回找到的字符值，如果到达文件尾则返回 EOF；cin.get( )的第二种形式接受一个字符引用参数，将输入流中的下一个字符赋给该字符变量，并返回是一个 iostream 对象；第三种形式的 get( )接受一个数组、最多要读取的字符数和终止字符作为参数，除非遇到终止字符，否则它将比最大字符数参数少一个字符存储到数组中（在末尾加上空字符），如果遇到终止字符，则在数组末尾加上空字符，并将终止字符留在输入流中。

4. cin.read( )用于读取二进制数据结构；getline( )用于从 istream 的缓冲区中读取。

5. 宽度刚好显示整个数。

6. istream 对象引用。

7. 要打开的文件的名称。

8. ios::ate 跳到文件末尾，但用户可以在文件的任何地方写入数据。

练习

1. 下面是一种解决方案：

```
0:  // Ex1701.cpp
1:  #include <iostream>
2:  int main()
3:  {
4:     int x;
5:     std::cout << "Enter a number: ";
6:     std::cin >> x;
7:     std::cout << "You entered: " << x << std::endl;
8:     std::cerr << "Uh oh, this to cerr!" << std::endl;
9:     std::clog << "Uh oh, this to clog!" << std::endl;
10:    return 0;
11: }
```

2. 下面是一种解决方案：

```
0:  // Ex1702.cpp
1:  #include <iostream>
2:  int main()
3:  {
4:    char name[80];
5:    std::cout << "Enter your full name: ";
6:    std::cin.getline(name,80);
7:    std::cout << "\nYou entered: " << name << std::endl;
8:    return 0;
9:  }
```

3. 下面是一种解决方案：

```
0:  // Ex1703.cpp
1:  #include <iostream>
2:  using namespace std;
3:
4:  int main()
5:  {
6:    char ch;
7:    cout << "enter a phrase: ";
8:    while ( cin.get(ch) )
9:    {
10:     switch (ch)
11:     {
12:       case '!':
13:         cout << '$';
14:         break;
15:       case '#':
16:         break;
17:       default:
18:         cout << ch;
19:         break;
20:     }
21:   }
22:   return 0;
23: }
```

4. 下面是一种解决方案：

```
0:  // Ex1704.cpp
```

```
#include <fstream>
#include <iostream>
using namespace std;

int main(int argc, char**argv)   // returns 1 on error
{
  if (argc != 2)
  {
    cout << "Usage: argv[0] <infile>\n";
    return(1);
  }

  // open the input stream
  ifstream fin (argv[1],ios::binary);
  if (!fin)
  {
    cout << "Unable to open " << argv[1] <<" for reading.\n";
    return(1);
  }

  char ch;
  while ( fin.get(ch))
    if ((ch > 32 && ch < 127) || ch == '\n' || ch == '\t')
      cout << ch;
  fin.close();
}
```

5. 下面是一种解决方案：

```
// Ex1705.cpp
#include <iostream>

int main(int argc, char**argv)   // returns 1 on error
{
  for (int ctr = argc-1; ctr>0 ; ctr--)
    std::cout << argv[ctr] << " ";
}
```

第18章

测验

1. Outer::Inner::Myfunc( );

2. 程序运行到“HERE”处后，将使用全局变量X，因此为4。

3. 可以。为此，可以用名称空间来限定名称。

4. 常规名称空间中的名称可以在声明该名称空间的编译单元外部使用；未命名的名称空间中的名称只能在声明该名称空间的编译单元内使用。

5. using 编译指令和 using 声明。前者使得可以像使用常规名称那样使用名称空间中所有的名称；后者让程序能够使用指定的名称，而无需用名称空间名进行限定。

6. 未命名的名称空间是没有名称的名称空间，用于封装一组声明以免发生名称冲突。未命名的名称空间中的名称只能在声明该名称空间的编译单元中使用。

7. 标准名称空间 std 是由 C++标准库定义的，它包含标准库中所有名称的声明。

练习

1. C++标准头文件 iostream 在名称空间中 std 声明了 cout 和 endl。如果不使用名称空间来限定，不能在标准名称空间外使用它们。

2. 可以在第 0 和 1 行之间添加下述代码：

```
using namespace std;
```

也可以在第 0 和 1 行之间添加如下代码：

```
using std::cout;
using std::endl;
```

还可以将第 3 行修改成下面这样：

```
std::cout << "Hello world!" << std::endl;
```

3. 下面是一种答案：

```
Namespace MyStuff
{
     class MyClass
     {
          //MyClass stuff
     }
}
```

第 19 章

测验

1. 模板是 C++语言内置的，是类型安全的。宏是通过预处理器实现的，不是类型安全的。

2. 模板参数为每种类型创建一个模板实例。如果创建 6 个模板实例，将创建 6 种不同的类。函数参数修改函数的行为或数据，但只创建一个函数。

3. 通用模板友元函数为每种类型的参数化类创建一个函数；特定类型的模板友元函数为参数化类的每个实例创建一个特定类型的实例。

4. 可以。除创建 Array<t>::SomeFunction( )外，还可以创建 Array<int>::SomeFunction( )来修改 int 数组的行为。

5. 每个模板实例一个。

6. 必须定义了默认构造函数、复制构造函数和重载的赋值运算符。

7. STL 表示标准模板库。这个库之所以重要，是因为它包含大量已创建好、可供程序员使用的模板类。由于这些类是 C++标准的组成部分，因此任何遵循了 C++标准的编译器都支持它们。这意味着程序员无需做重复的工作，去创建这些类。

练习

1.下面是实现该模板的一种方式：

```
0:  //Exercise 19.1
1:  template <class Type>
2:  class List
3:  {
4:
5:    public:
6:      List():head(0),tail(0),theCount(0) { }
```

```
 7:     virtual ~List();
 8:
 9:     void insert( Type value );
10:     void append( Type value );
11:     int is_present( Type value ) const;
12:     int is_empty() const { return head == 0; }
13:     int count() const { return theCount; }
14:
15:   private:
16:     class ListCell
17:     {
18:       public:
19:         ListCell(Type value, ListCell *cell =
                0):val(value),next(cell){}
20:         Type val;
21:         ListCell *next;
22:     };
23:
24:     ListCell *head;
25:     ListCell *tail;
26:     int theCount;
27: };
```

2. 下面是一种答案：

```
 0: // Exercise 19.2
 1: void List::insert(int value)
 2: {
 3:     ListCell *pt = new ListCell( value, head );
 4:
 5:     // this line added to handle tail
 6:     if ( head == 0 ) tail = pt;
 7:
 8:     head = pt;
 9:     theCount++;
10: }
11:
12: void List::append( int value )
13: {
14:     ListCell *pt = new ListCell( value );
15:     if ( head == 0 )
16:         head = pt;
17:     else
18:         tail->next = pt;
19:
20:     tail = pt;
21:     theCount++;
22: }
23:
24: int List::is_present( int value ) const
25: {
26:     if ( head == 0 ) return 0;
27:     if ( head->val == value || tail->val == value )
```

```
28:         return 1;
29:
30:      ListCell *pt = head->next;
31:      for (; pt != tail; pt = pt->next)
32:          if ( pt->val == value )
33:              return 1;
34:
35:      return 0;
36: }
```

3. 下面是一种答案：

```
 0: // Exercise 19.3
 1: template <class Type>
 2: List<Type>::~List()
 3: {
 4:     ListCell *pt = head;
 5:
 6:     while ( pt )
 7:     {
 8:         ListCell *tmp = pt;
 9:         pt = pt->next;
10:         delete tmp;
11:     }
12:     head = tail = 0;
13: }
14:
15: template <class Type>
16: void List<Type>::insert(Type value)
17: {
18:     ListCell *pt = new ListCell( value, head );
19:     assert (pt != 0);
20:
21:     // this line added to handle tail
22:     if ( head == 0 ) tail = pt;
23:
24:     head = pt;
25:     theCount++;
26: }
27:
28: template <class Type>
29: void List<Type>::append( Type value )
30: {
31:     ListCell *pt = new ListCell( value );
32:     if ( head == 0 )
33:         head = pt;
34:     else
35:         tail->next = pt;
36:
37:     tail = pt;
38:     theCount++;
39: }
40:
```

```
template <class Type>
int List<Type>::is_present( Type value ) const
{
    if ( head == 0 ) return 0;
    if ( head->val == value || tail->val == value )
        return 1;

    ListCell *pt = head->next;
    for (; pt != tail; pt = pt->next)
        if ( pt->val == value )
            return 1;

    return 0;
}
```

4. 下面的代码声明这3个对象：

```
List<String> string_list;
List<Cat> Cat_List;
List<int> int_List;
```

5. Cat没有定义operator==，所有对List节点中的值进行比较的操作（如is_present）都将导致编译错误。为减少这种出错机会，在模板定义前进行详细注释，指出为使实例化能够通过编译，必须定义哪些操作。

6. 下面是一种答案：

```
friend int operator==( const Type& lhs, const Type& rhs );
```

7. 下面是一种答案：

```
Exercise 19.7
template <class Type>
int List<Type>::operator==( const Type& lhs, const Type& rhs )
{
    // compare lengths first
    if ( lhs.theCount != rhs.theCount )
        return 0;    // lengths differ

    ListCell *lh = lhs.head;
    ListCell *rh = rhs.head;

    for(; lh != 0; lh = lh.next, rh = rh.next )
        if ( lh.value != rh.value )
            return 0;

    return 1;        // if they don't differ, they must match
}
```

8. 是的，由于数组涉及元素比较，因此也必须为元素定义operator!=。

9. 下面是一种答案：

```
// Exercise 19.9
// template swap:
// must have assignment and the copy constructor defined for the Type.
template <class Type>
void swap( Type& lhs, Type& rhs)
{
```

```
6:      Type temp( lhs );
7:      lhs = rhs;
8:      rhs = temp;
9:  }
```

第 20 章

测验

1. 异常是由于使用关键字 throw 而创建的对象，用于指示异常状态，沿调用栈向上传递到能够处理它的第一条 catch 语句。

2. try 块是一组可能导致异常的语句。

3. catch 语句是一个例程，其参数指出了它能处理的异常类型。它位于 try 块后面，用于捕获 try 块内可能引发的异常。

4. 异常是一个对象，可包含在用户创建的类中能定义的任何信息。

5. 程序调用关键字 throw 时将创建异常对象。

6. 通常应该按引用来传递。如果不打算修改异常对象的内容，应传递 const 引用。

7. 如果按引用传递异常就可以。

8. catch 语句按它们出现在源代码中的顺序被检查，将使用特征标与异常匹配的第一条 catch 语句。一般而言，最好按具体到通用的顺序捕获异常。

9. catch(...)捕获任何类型的异常。

10. 断点是代码中调试器停止执行的地方。

练习

1. 下面是一种答案：

```
 0:  #include <iostream>
 1:  using namespace std;
 2:  class OutOfMemory {};
 3:  int main()
 4:  {
 5:      try
 6:      {
 7:          int *myInt = new int;
 8:          if (myInt == 0)
 9:              throw OutOfMemory();
10:      }
11:      catch (OutOfMemory)
12:      {
13:          cout << "Unable to allocate memory!" << endl;
14:      }
15:      return 0;
16:  }
```

2. 下面是一种答案：

```
1:  #include <iostream>
2:  #include <stdio.h>
3:  #include <string.h>
4:  using namespace std;
5:  class OutOfMemory
```

```
{
  public:
    OutOfMemory(char *);
    char* GetString() { return itsString; }
  private:
    char* itsString;
};

OutOfMemory::OutOfMemory(char * theType)
{
    itsString = new char[80];
    char warning[] = "Out Of Memory! Cant't allocate room for: ";
    strncpy(itsString,warning,60);
    strncat(itsString,theType,19);
}

int main()
{
    try
    {
        int *myInt = new int;
        if (myInt == 0)
            throw OutOfMemory("int");
    }
    catch (OutOfMemory& theException)
    {
        cout << theException.GetString();
    }
    return 0;
}
```

3. 下面是一种答案：

```
// Exercise 20.3
#include <iostream>
using namespace std;
// Abstract exception data type
class Exception
{
  public:
    Exception(){};
    virtual ~Exception(){}
    virtual void PrintError() = 0;
};

// Derived class to handle memory problems.
// Note no allocation of memory in this class!
class OutOfMemory : public Exception
{
  public:
    OutOfMemory(){}
    ~OutOfMemory(){}
    virtual void PrintError();
```

```
20:    private:
21:  };
22:
23:  void OutOfMemory::PrintError()
24:  {
25:    cout << "Out of Memory!!" << endl;
26:  }
27:
28:  // Derived class to handle bad numbers
29:  class RangeError : public Exception
30:  {
31:   public:
32:     RangeError(unsigned long number){badNumber = number;}
33:     ~RangeError(){}
34:     virtual void PrintError();
35:     virtual unsigned long GetNumber() { return badNumber; }
36:     virtual void SetNumber(unsigned long number) {badNumber =
          number;}
37:   private:
38:     unsigned long badNumber;
39:  };
40:
41:  void RangeError::PrintError()
42:  {
43:    cout << "Number out of range. You used " ;
44:    cout << GetNumber() << "!!" << endl;
45:  }
46:
47:  void MyFunction(); // func. prototype
48:
49:  int main()
50:  {
51:    try
52:    {
53:      MyFunction();
54:    }
55:    // Only one catch required, use virtual functions to do the
56:    // right thing.
57:    catch (Exception& theException)
58:    {
59:      theException.PrintError();
60:    }
61:    return 0;
62:  }
63:
64:  void MyFunction()
65:  {
66:      unsigned int *myInt = new unsigned int;
67:      long testNumber;
68:
69:      if (myInt == 0)
70:        throw OutOfMemory();
```

```
71:
72:       cout << "Enter an int: ";
73:       cin >> testNumber;
74:
75:       // this weird test should be replaced by a series
76:       // of tests to complain about bad user input
77:
78:       if (testNumber > 3768 || testNumber < 0)
79:         throw RangeError(testNumber);
80:
81:       *myInt = testNumber;
82:       cout << "Ok. myInt: " << *myInt;
83:       delete myInt;
84:   }
```

4. 下面是一种答案：

```
 0:   // Exercise 20.4
 1:   #include <iostream>
 2:   using namespace std;
 3:   // Abstract exception data type
 4:   class Exception
 5:   {
 6:     public:
 7:       Exception(){}
 8:       virtual ~Exception(){}
 9:       virtual void PrintError() = 0;
10:   };
11:
12:   // Derived class to handle memory problems.
13:   // Note no allocation of memory in this class!
14:   class OutOfMemory : public Exception
15:   {
16:     public:
17:       OutOfMemory(){}
18:       ~OutOfMemory(){}
19:       virtual void PrintError();
20:     private:
21:   };
22:
23:   void OutOfMemory::PrintError()
24:   {
25:     cout << "Out of Memory!!\n";
26:   }
27:
28:   // Derived class to handle bad numbers
29:   class RangeError : public Exception
30:   {
31:     public:
32:       RangeError(unsigned long number){badNumber = number;}
33:       ~RangeError(){}
34:       virtual void PrintError();
35:       virtual unsigned long GetNumber() { return badNumber; }
```

```
36:       virtual void SetNumber(unsigned long number) {badNumber =
              number;}
37:     private:
38:       unsigned long badNumber;
39:   };
40:
41:   void RangeError::PrintError()
42:   {
43:     cout << "Number out of range. You used ";
44:     cout << GetNumber() << "!!" << endl;
45:   }
46:
47:   // func. prototypes
48:   void MyFunction();
49:   unsigned int * FunctionTwo();
50:   void FunctionThree(unsigned int *);
51:
52:   int main()
53:   {
54:     try
55:     {
56:       MyFunction();
57:     }
58:     // Only one catch required, use virtual functions to do the
59:     // right thing.
60:     catch (Exception& theException)
61:     {
62:       theException.PrintError();
63:     }
64:     return 0;
65:    }
66:
67:   unsigned int * FunctionTwo()
68:   {
69:     unsigned int *myInt = new unsigned int;
70:     if (myInt == 0)
71:       throw OutOfMemory();
72:     return myInt;
73:   }
74:
75:   void MyFunction()
76:   {
77:       unsigned int *myInt = FunctionTwo();
78:       FunctionThree(myInt);
79:       cout << "Ok. myInt: " << *myInt;
80:       delete myInt;
81:   }
82:
83:   void FunctionThree(unsigned int *ptr)
84:   {
85:       long testNumber;
86:       cout << "Enter an int: ";
```

```
87:         cin >> testNumber;
88:         // this weird test should be replaced by a series
89:         // of tests to complain about bad user input
90:         if (testNumber > 3768    testNumber < 0)
91:           throw RangeError(testNumber);
92:         *ptr = testNumber;
93:   }
```

5. 在处理“内存耗尽”状态的过程中，xOutOfMemory 的构造函数创建了一个 string 对象。这种异常仅在程序耗尽内存时才会出现，因此该内存分配操作将失败。

试图创建该 string 对象可能引发相同的异常，这将形成无限循环，直到程序崩溃。如果确实需要该 string 对象，可在程序开始前在静态缓冲区中分配空间，然后在该异常被引发时使用它。

可将 if(var==0)修改为 if(0)（这将引发内存耗尽异常）以测试该程序。

第 21 章

测验

1. 多重包含防范用于防止头文件被多次包含到程序中。
2. 这个问题只能由读者来回答，因为这取决于使用的编译器。
3. 前者将 debug 定义为等于 0，每当出现 debug 时，都将用字符 0 来替换它；后者撤销对 debug 的定义，文件中的 debug 将保留不变。
4. 结果为 4/2，即 2。
5. 结果为 10 + 10 / 2，即 15。这显然不是所需的结果。
6. 应该添加括号，如下所示：

```
HALVE (x) ((x)/2)
```

7. 两个字节为 16 位，因此最多可存储 16 位值。
8. 5 位能够存储 32 个不同的值（0～31）。
9. 结果为 1111 1111。
10. 结果为 0011 1100。

练习

1. 头文件 STRING.H 的多重包含防范语句如下：

```
#ifndef STRING_H
#define STRING_H
...
#endif
```

2. 下面是一种答案：

```
0:   #include <iostream>
1:
2:   using namespace std;
3:   #ifndef DEBUG
4:   #define ASSERT(x)
5:   #elif DEBUG == 1
6:   #define ASSERT(x) \
7:             if (! (x)) \
8:             { \
9:        cout << "ERROR!! Assert " << #x << " failed" << endl; \
```

```
10:             }
11:    #elif DEBUG == 2
12:    #define ASSERT(x) \
13:             if (! (x) ) \
14:             { \
15:        cout << "ERROR!! Assert " << #x << " failed" << endl; \
16:        cout << " on line " << __LINE__  << endl; \
17:        cout << " in file " << __FILE__ << endl;  \
18:             }
19:    #endif
```

3. 下面是一种答案：

```
#ifndef DEBUG
#define DPRINT(string)
#else
#define DPRINT(STRING) cout << #STRING ;
#endif
```

4. 下面是一种答案：

```
class myDate
{
   public:
      // stuff here...
   private:
      unsigned int Month : 4;
      unsigned int Day   : 8;
      unsigned int Year  : 12;
}
```