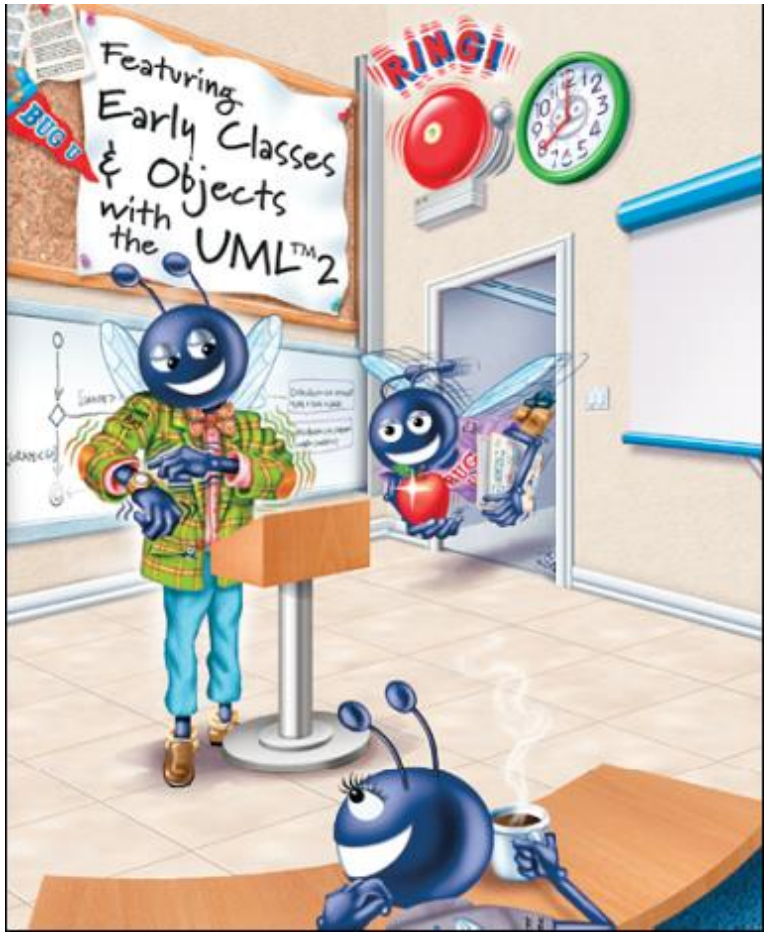
C++程序设计



上节课内容回顾

1. if、if...else、while
2. 计数器控制的循环和标记控制的循环
3. 使用 for 循环语句和 do..while 循环
4. 使用 switch 选择语句
5. 使用 break 和 continue 改变控制流程

第五讲 函数和递归

学习目标：

- 理解函数间的信息传递机制
- 能利用随机数生成机制实现模拟技术
- 理解如何使标识符的可见性限定于特定的程序区域中
- 理解如何编写和使用自我调用的函数
- 理解引用、函数重载、函数模板的概念

1. 函数原型和参数强制类型转换

- Function prototype (函数原型)
  - 也称为 function declaration (函数声明)
  - 告知编译器
    - 函数名称
    - 函数返回类型
    - 函数所需要的参数
      - 参数个数、参数类型、参数顺序

1. 函数原型和参数强制类型转换

- Function signature (函数签名)
  - 函数名、参数个数、参数类型
    - 注意：不包括函数返回类型
- Argument Coercion (参数强制转换)
  - 例如，以整型参数调用一个函数，即使这个函数原型声明的参数为double类型，这个函数仍能正确工作

1. 函数原型和参数强制类型转换

- C++ Promotion Rules (提升规则)
  - 如何转换类型且不丢失数据
  - 应用于有多种数据类型参与的表达式
    - ◈混合类型表达式
    - ◈提升至表达式中的“最高”类型
      - ◈创建临时值进行计算
        - ◇原始值保持不变

1. 函数原型和参数强制类型转换

- **C++ Promotion Rules (提升规则)**

Data types	
`long double`	
`double`	
`float`	
`unsigned long long int`	(synonymous with `unsigned long long`; in the new standard)
`long long int`	(synonymous with `long long`; in the new standard)
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char`	
`bool`	

1. 函数原型和参数强制类型转换

- C++ Promotion Rules (提升规则)
  - 当实参与形参类型不匹配时也将产生提升
  - 转换为更低的数据类型时
    - 数据丢失
    - 只能显示转换
      - 编译器警告
      - 使用类型转换运算符

2. Case Study: 随机数的生成

- rand function (<cstdlib>)
  - i = rand();
  - Generates unsigned integer between 0 and RAND_MAX (usually 32767)

2. Case Study: 随机数的生成

- 产生特定范围的随机数可以使用取模运算 (%)
  - Example: rand() % 6;
    - 产生 0 到 5 的整数
  - This is called scaling, 6 is the scaling factor
  - Shifting can move the range to 1 to 6

    1 + rand() % 6;

2. Case Study: 随机数的生成

- Function rand
  - 产生伪随机数
  - 每次程序执行得到相同序列的随机数
- C++ Standard Library function srand
  - 参数： unsigned int
  - rand 的种子，用来产生不同序列的随机数

2. Case Study: 随机数的生成

- 无需每次输入一个种子值
  - srand(time(0));
    - 读取系统始终获得种子值
  - Function time (with the argument 0)
    - 返回格林威治标准时间1970年1月1日0时起到现在的秒数
    - 函数原型在 <ctime> 中

2. Case Study: 随机数的生成

- **Scaling and shifting random numbers**
  - 要得到指定范围的随机数

    number = *shiftingValue* + rand() % *scalingFactor*;

    - *shiftingValue* 为指定范围内的第一个数
    - *scalingFactor* 指定范围的宽度

3. Game of Chance and Introducing enum

- Enumeration (枚举)
  - 一组整型常量
    - 开始于0，以1递增
  - 定义枚举
    - Example

```
enum Months { JAN = 1, FEB, MAR, APR };
```

3. Game of Chance and Introducing enum

- Craps(双骰儿赌博) simulator rules
  - Roll two dice(骰子)
    - ✓ 7 or 11 on first throw, player wins
    - ✓ 2, 3, or 12 on first throw, player loses
    - ✓ 4, 5, 6, 8, 9, 10
      - value becomes player's "point"
      - player must roll his point before rolling 7 to win

```
.....
int main()
 {
   // enumeration with constants that represent the game status
   enum Status { CONTINUE, WON, LOST }; // all caps in constants

   int myPoint; // point if no win or loss on first roll
   Status gameStatus; // can contain CONTINUE, WON or LOST

   // randomize random number generator using current time
   srand( time( 0 ) );

   int sumOfDice = rollDice(); // first roll of the dice
```

```
switch ( sumOfDice )
{
   case 7: // win with 7 on first roll
   case 11: // win with 11 on first roll
      gameStatus = WON;
      break;
   case 2: // lose with 2 on first roll
   case 3: // lose with 3 on first roll
   case 12: // lose with 12 on first roll
      gameStatus = LOST;
      break;
   default: // did not win or lose, so remember point
      gameStatus = CONTINUE; // game is not over
      myPoint = sumOfDice; // remember the point
      cout << "Point is " << myPoint << endl;
      break; // optional at end of switch
} // end switch
```

```
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again
      if ( sumOfDice == myPoint ) // win by making point
         gameStatus = WON;
      else
         if ( sumOfDice == 7 ) // lose by rolling 7 before point
            gameStatus = LOST;
   } // end while

   // display won or lost message
   if ( gameStatus == WON )
      cout << "Player wins" << endl;
   else
      cout << "Player loses" << endl;

   return 0; // indicates successful termination
} // end main
```

```
int rollDice()
{
   // pick random die values
   int die1 = 1 + rand() % 6; // first die roll
   int die2 = 1 + rand() % 6; // second die roll

   int sum = die1 + die2; // compute sum of die values
   // display results of this roll
   cout << "Player rolled " << die1 << " + " << die2
      << " = " << sum << endl;
   return sum; // end function rollDice
} // end function rollDice
```

3. Game of Chance and Introducing enum

良好编程习惯：使用用户自定义类型名称的标识符，其首字母应该大写。

良好编程习惯：在枚举常量中，只采用大写字母。这样可以使这些常量在程序中更能引起程序员的注意，提醒程序员枚举并不是变量。

良好编程习惯：在程序中使用枚举常量代替整数常量能使程序更清晰。

4. Storage Classes （存储类别）

- 标识符的属性
  - 名字，类型，大小和值
  - storage class, scope and linkage
- C++ provides five storage-class specifiers
  - auto, register, extern, mutable and static

4. Storage Classes （存储类别）

- **Automatic storage class**
  - Declared with keywords auto and register
  - Automatic variables
    - ◈在进入定义变量的语句块时被创建，在该语句块内有效，退出该语句块后被销毁
  - 通常局部变量和函数参数为 automatic storage class

4. Storage Classes （存储类别）

- **Static storage class**
  - Declared with keywords extern and static
  - Static-storage-class variables
    - 程序开始执行时存在，遇到声明进行一次初始化，在程序运行期间有效
  - Static-storage-class functions
    - 和其他函数一样，在程序开始运行时函数名存在
  - 并不意味着这些标识符在整个程序中都能使用

5. Scope Rules（作用域规则）

- Scope (作用域)
  - 标识符在哪里可以使用
  - 四种作用域
    - Function scope
    - File scope
    - Block scope
    - Function-prototype scope

6. Function Call Stack（函数调用堆栈）

- **Function Call Stack**
  - Sometimes called the program execution stack
  - Supports the function call/return mechanism
    - 一个函数调用另一个函数时，一个堆栈项（stack frame）或活动记录（ active record ）被压入堆栈
      - 维护被调用函数返回到其调用者所需的返回地址
      - 维护被调用函数的自动变量（函数参数和局部变量）

7. Inline Functions（内联函数）

- inline functions
  - 减少函数调用的开销
  - 编译器将函数代码拷贝到程序中，而不是使用函数调用
  - 编译器可以忽略 inline 限定符
  - 应该将小函数，经常使用的函数作为内联函数

7. Inline Functions

```
inline double cube( const double side )
{
   return side * side * side; // calculate cube
} // end function cube
```

8. References and Reference Parameters

- 两种方式向函数传递参数
  - Pass-by-value
    - 参数值的拷贝传递给被调用函数
    - 对拷贝的修改不影响调用者原始值
  - Pass-by-reference
    - 被调用函数可以直接修改调用者的参数

8. References and Reference Parameters

- **Pass by reference uses a reference parameter**
  - 引用参数是函数调用中参数的别名
  - 在函数原型的参数类型前加"&"，说明这是引用参数

8. References and Reference Parameters

- Pass by reference uses a reference parameter
  - 函数原型：void myFunction(int &)
  - 函数定义：
    - void myFunction(int &data1)
    - The function returns an int named data1
    - Read “data1 is a reference to an int”

```
int squareByValue( int ); // function prototype (value pass)
void squareByReference( int & ); // function prototype (reference pass)
int main()
 {
   int x = 2; // value to square using squareByValue
   int z = 4; // value to square using squareByReference

   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
     << squareByValue( x ) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;

   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
   return 0; // indicates successful termination
} // end main
```

```
int squareByValue( int number )
{
   return number *= number; // caller's argument not modified
} // end function squareByValue

void squareByReference( int &numberRef )
{
   numberRef *= numberRef; // caller's argument modified
} // end function squareByReference
```

8. References and Reference Parameters

```
int FibonElem( int pos )
{
   int elem = 1;
   int n1 = 1, n2 = 1;
   for( int i = 3; i <= pos; ++i)
   {
      elem = n1 + n2;
      n1 = n2;
      n2 = elem;
   }
   return elem;
}
```

8. References and Reference Parameters

```
bool FibonElem( int pos, int &elem )
{
   if(pos <= 0 || pos > 1024)
   {
       elem = 0; return false;
   }
   elem = 1;
   int n1 = 1, n2 = 1;
   for( int i = 3; i <= pos; ++i)
   {
       elem = n1 + n2;
       n1 = n2;
       n2 = elem;
   }
   return true;
}
```

8. References and Reference Parameters

- 引用作为其他变量的别名
  - 引用相同的变量
  - 可以在函数内使用

```
int count = 1;   //declare integer variable count
int &cRef = count; // create cRef as an alias for count
++cRef;          // increment count (using its alias)
```

- 引用在声明时必须进行初始化

8. References and Reference Parameters

软件工程知识：综合考虑到程序的清晰性和高性能，许多C++程序员喜欢使用指针将可修改的参数传递给函数，小型的非修改性参数可以按值传递，大型的非修改性参数则可以利用常量引用传递给函数。

9. Default Arguments

- **Default argument**
  - 参数的默认值，当函数调用没有提供相应参数时
  - 基于参数列表的最右侧原则
  - 通常默认参数在函数原型中说明

9. Default Arguments

- Set defaults in function prototype

```
int myFunction( int x = 1, int y = 2, int z = 3 );
```

  - myFunction(3)

    x = 3, y and z get defaults (rightmost)

  - myFunction(3, 5)

    x = 3, y = 5 and z gets default

10. Unary Scope Resolution Operator

- 一元解析运算符 (::)
  - 如果局部变量与全局变量同名，访问全局变量
  - ::variable
    - y = ::x + 3;
  - 应避免全局变量与局部变量同名

11. Function Overloading（函数重载）

- **Function overloading**
  - 相同的函数名不同的参数列表
  - 完成相似的任务
    - int square(int x) {return x * x;}
    - float square(float x) { return x * x; }

11. Function Overloading （函数重载）

- Function overloading
  - 编译器通过签名来选择函数
    - ◈签名由函数名和参数类型决定
  - 可以具有相同的返回类型

11. Function Overloading

- **Function overloading**

分析以下两种情况，是否构成函数的重载

第一种情况：　(1) void output();

(2) int output();

第二种情况：　(1) void output(int a,int b=5);

(2) void output(int a);

12. Function Templates（函数模板）

- Compact way to make overloaded functions
  - 函数模板自动为不同的数据类型产生不同的函数
- Format
  - usage: template <class *type*>

```
template < class T >  // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
   T maximumValue = value1; // assume value1 is maximum

   // determine whether value2 is greater than maximumValue
   if ( value2 > maximumValue )
      maximumValue = value2;

   // determine whether value3 is greater than maximumValue
   if ( value3 > maximumValue )
      maximumValue = value3;

   return maximumValue;
} // end function template maximum
```

13. Recursion

- Recursive function
  - 调用自身的函数
- Recursion
  - Base case(s)
    - 函数知道如何处理的最简单情况

13. Recursion

- **Factorial**
  - The factorial of a nonnegative integer *n*, written *n*! (and pronounced "*n* factorial"), is the product
    - $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
  - Recursive definition of the factorial function
    - $n! = n \cdot (n-1)!$
    - Example: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

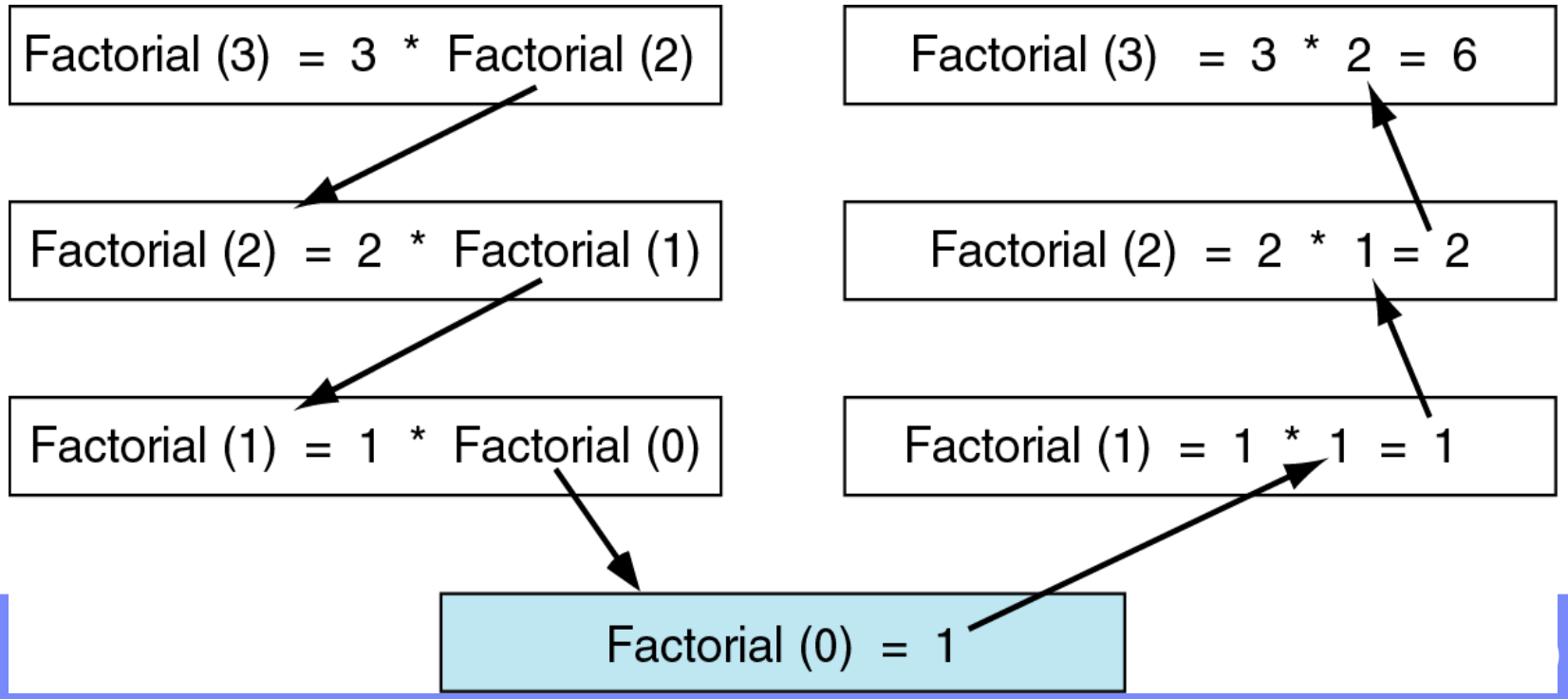
      $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

      $5! = 5 \cdot (4!)$

13. Recursion

```
int factorial (int n)
{
/* Statements */
if (n == 0)
   return 1;
else
   return
     (n*factorial(n-1);

} /* factorial */
```



14. Recursion vs. Iteration

```
// iterative function factorial
unsigned long factorial( unsigned long number )
{
        unsigned long result = 1;

        // iterative factorial calculation
        for ( unsigned long i = number; i >= 1; --i )
                result *= i;
        return result;
} // end function factorial
```

14. Recursion vs. Iteration

- 均基于控制结构
  - Iteration – 循环
  - Recursion – 选择
- 均包含循环
  - Iteration – 显式使用循环结构
  - Recursion – 重复函数调用
- 均包含终止测试
  - Iteration – 循环终止测试
  - Recursion – base case

14. Recursion vs. Iteration

- 均逐渐达到终止条件
  - Iteration 修改计数器直至测试条件失败
  - Recursion 产生问题的简单版本进行计算
- 均有可能出现无限循环
  - Iteration – 如果控制循环的条件始终为真
  - Recursion – 如果没有能够解决问题的简单情况

14. Recursion vs. Iteration

- Negatives of recursion
  - 重复函数调用的开销
    - 占用处理器时间和内存空间
- Iteration
  - 通常出现在一个函数中
  - 开销较小

思考题：

- P241 6.54 修改双骰游戏程序，允许玩家下赌注
  - 把掷骰子部分打包为一个函数
  - 初始化变量bankBalance为1000美元
  - 提示玩家输入赌注：wager，检查其是否<=bankBalance，如果不是，提示用户重新输入，直到获得一个合法值
  - 如果玩家获胜，bankBalance增加wager；如果玩家失利，bankBalance减少wager
  - 打印新的bankBalance值，并检查其值是否为0，如果是，则打印“Sorry, you busted!”
  - 在游戏过程中可以添加聊天效果