
6

# Functions and an Introduction to Recursion

东南大学软件学院

# OBJECTIVES

In this chapter you'll learn:

- To use common math functions available in the C++ Standard Library.
- To create functions with multiple parameters.
- The mechanisms for passing information between functions and returning results.
- How the function call/return mechanism is supported by the function call stack and activation records.
- To use random number generation to implement game-playing applications.
- How the visibility of identifiers is limited to specific regions of programs.
- To write and use recursive functions, i.e., functions that call themselves.

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

int main() {
  int threeExpFour = 1;
  for (int i = 0; i < 4; i = i + 1) {
    threeExpFour = threeExpFour * 3;
  }
  cout << "3^4 is " << threeExpFour << endl;
  int sixExpFive = 1;
  for (int i = 0; i < 5; i = i + 1) {
    sixExpFive = sixExpFive * 6;
  }
  cout << "6^5 is " << sixExpFive << endl;
  int twelveExpTen = 1;
  for (int i = 0; i < 10; i = i + 1) {
    twelveExpTen = twelveExpTen * 12;
  }
  cout << "12^10 is " << twelveExpTen << endl;
  return 0;
}
```

# 6.1 Introduction

- **Divide and conquer technique 分而治之**
  - **Construct a large program from small, simple pieces (e.g., components)**
- **Functions 函数 (<cmath>中的pow(x,y)函数)**
  - **Allow programmers to modularize a program by separating its tasks into self-contained units 模块化**
    - **Statements in function bodies are written only once**
      - **Reused from perhaps several locations in a program**
      - **Hidden from other functions**
      - **Avoid repeating code**
    - **Enable the divide-and-conquer approach**
    - **Reusable in other programs**

# Error-Prevention Tip 6.1

**A small function that performs one task is easier to test and debug than a larger function that performs many tasks.** 功能单一的小程序更容易测试和调试。

# 6.2 Program Components in C++ (cont.)

- **Function (Cont.)**
  - **A function is invoked by a function call**
    函数通过函数调用来执行
    - **Called function（被调函数） either returns a result or simply returns to the caller function（主调函数）**
    - **Function calls form hierarchical relationships** 函数调用采用层次结构

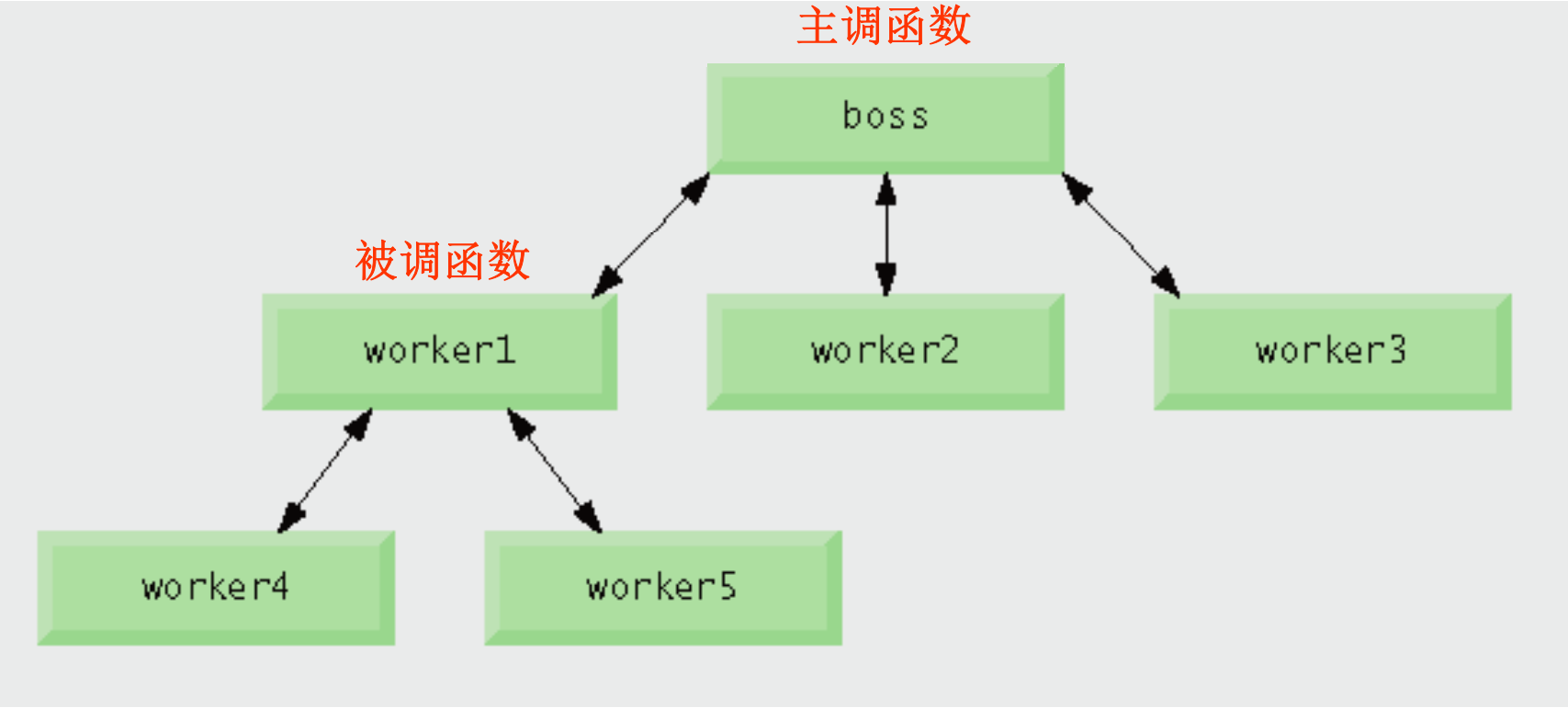
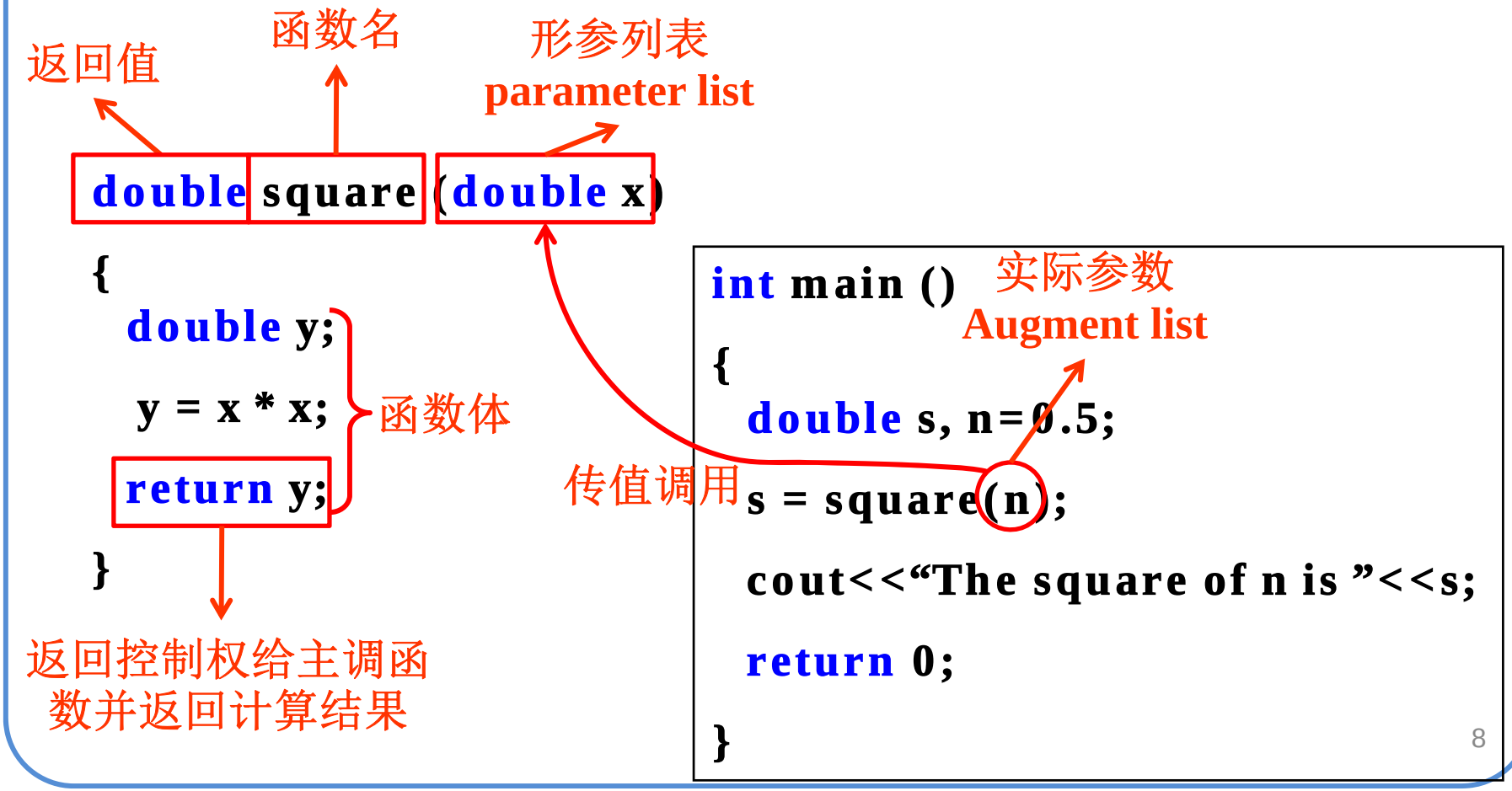



Fig. 6.1 | Hierarchical boss function/worker function relationship.

# Function Definitions with Single Parameter

- Single Parameter 单形参函数
  - Define a function to calculate the square of x



# 6.4 Function Definitions with Multiple Parameters (Cont.)

- **Three ways to return control to the calling statement:**
  - **If the function does not return a result:**
    - **Program flow reaches the function-ending right brace or**
    - **Program executes the statement `return;`**
  - **If the function does return a result:**
    - **Program executes the statement `return` *expression*;**
      - ***expression* is evaluated and its value is returned to the caller**

# 6.12 Functions with Empty Parameter Lists

- **Empty parameter list**
  - **Specified by writing either `void` or nothing at all in parentheses**
  - **For example,**

```
void print();
```

**specifies that function `print` does not take arguments and does not return a value**

# 6.4 Function Definitions with Multiple Parameters

- **Multiple parameters 多形参函数**
  - Functions often require more than one piece of information to perform their tasks
  - Specified in the function header as a comma-separated list of parameters

```
int mypow( double x, int n)
{
  double result = 1;

  for (int i=0; i<n; i++)
     result *= x;

  return result;
}
```

以逗号分割的
形参列表

以逗号分割的
实参列表

```
int main()
{
  int a = 3;
  double r, b = 2.8;
  r = mypow(b, a);

  return 0;
}
```

函数调用原则：

1. 函数调用名称与函数定义名称一致
2. 实参类型要与形参相容
3. 实参数量和顺序要与形参一致
4. 实参数量要与形参一致（缺省参数时例外）

# Software Engineering Observation 6.4

The commas used in line 58 of Fig. 6.4 to separate the arguments to function `maximum` are not comma operators as discussed in Section 5.3. The comma operator guarantees that its operands are evaluated left to right. The order of evaluation of a function's arguments, however, is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders. The C++ standard *does* guarantee that all arguments in a function call are evaluated *before* the called function executes.

```
r = mypow(b, a); //b^a 正确

r = mypow(b, a+c); //b^(a+c) 正确

r = mypow(x++, x+y); X 错误！
```

```
//mysquare.cpp
#include <iostream>
using namespace std;
double square (double);
int main ()
{
  double s, n=0.5;
  s = square(n);
  cout<<"The square of n is "<<s;
  return 0;
}
double square (double x)
{
   double y;
   y = x * x;
   return y;
}
```

函数原型
function prototype

Compiler

当函数调用在函数定义之前时，编译器如何知道该函数的调用符合规定？

mysquare.cpp(7) : error C3861: "square": 找不到标识符

# 6.4 Function Definitions with Multiple Parameters (Cont.)

- Compiler uses a function prototype (函数原型) to: `double square (double);`
  - Check that calls to the function contain the correct number and types of arguments in the correct order
  - Ensure that the value returned by the function is used correctly in the expression that called the function
- Each argument must be consistent（相容） with the type of the corresponding parameter

# 6.5 Function Prototypes and Argument Coercion

- **Function prototype 函数原型**
  - **Also called a function declaration 函数声明**
  - **Indicates to the compiler:**
    1. **Name of the function**
    2. **Type of data returned by the function**
    3. **Parameters the function expects to receive**
       - ① **Number of parameters**
       - ② **Types of those parameters**
       - ③ **Order of those parameters**

```
int func ( int x, int y, int z );
```

## Software Engineering Observation 6.6

**Function prototypes are required in C++. Use `#include` preprocessor directives to obtain function prototypes for the C++ Standard Library functions from the header files for the appropriate libraries (e.g., the prototype for math function `sqrt` is in header file `<cmath>`). C++标准库中的头文件中包含了相应函数的原型。**

# 6.5 Function Prototypes and Argument Coercion (Cont.)

- **Function signature 函数签名**
  - **The portion of a function prototype that includes the name of the function and the types of its arguments**
    - **Does not specify the function's return type**
  - **Functions in the same scope must have unique signatures**
    - **The scope of a function is the region of a program in which the function is known and accessible**

```
int func ( int x, int y, int z );
```

# 6.5 Function Prototypes and Argument Coercion (Cont.)

- **Argument Coercion 实参类型强制转换**
  - **Forcing arguments to the appropriate types specified by the corresponding parameters**
    - **For example, calling a function with an integer argument, even though the function prototype specifies a double parameter**
      - **The function will still work correctly**

# 6.5 Function Prototypes and Argument Coercion (Cont.)

- **C++ Promotion Rules 升级规则**
  - **Indicate how to convert between types without losing data**
  - **Apply to expressions containing values of two or more data types 表达式中包含两种以上的数据类型**

| Data types | |
|---|---|
| `long double` | |
| `double` | |
| `float` | |
| `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` | |

| Data types | |
|---|---|
| long double | |
| double | |
| float | |
| 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 | |

Fig. 6.6 | Promotion hierarchy for fundamental data types.

# 6.5 Function Prototypes and Argument Coercion (Cont.)

- **C++ Promotion Rules (Cont.)**
  - **Promotion also occurs when the type of a function argument does not match the specified parameter type** 实参与形参类型不一致
    - **Promotion is as if the argument value were being assigned directly to the parameter variable**
  - **Converting a value to a lower fundamental type**
    - **Will likely result in the loss of data or incorrect values**
    - **Can only be performed explicitly**
      - **By assigning the value to a variable of lower type (some compilers will issue a warning in this case) or**
      - **By using a cast operator (static_cast< *type* >)**

```
warning C4244: “参数”：从“double”转换到“int”，可能丢失数据
```

# 6.6 C++ Standard Library Header Files

- C++ Standard Library header files
  - Each contains a portion of the Standard Library
    - Function prototypes for the related functions
    - Definitions of various class types and functions
    - Constants needed by those functions
  - "Instruct" the compiler on how to interface with library and user-written components
  - Header file names ending in .h
    - Are "old-style" header files
    - Superseded by the C++ Standard Library header files (被C++标准库头文件取代)

| C++ Standard Library header file | Explanation |
|---|---|
| `<iostream>` | Contains function prototypes for the C++ standard input and standard output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iostream.h>`. |
| `<iomanip>` | Contains function prototypes for stream manipulators that format streams of data. This header file is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output. This header file replaces header file `<iomanip.h>`. |
| `<cmath>` | Contains function prototypes for math library functions (discussed in Section 6.3). This header file replaces header file `<math.h>`. |
| `<cstdlib>` | Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header file are covered in Section 6.7; Chapter 11, Operator Overloading; String and Array Objects; Chapter 16, Exception Handling; Chapter 19, Web Programming; Chapter 22, Bits, Characters, C-Strings and `structs`; and Appendix E, C Legacy Code Topics. This header file replaces header file `<stdlib.h>`. |

| C++ Standard Library header file | Explanation |
|---|---|
| <ctime> | **Contains function prototypes and types for manipulating the time and date. This header file replaces header file <time.h>. This header file is used in Section 6.7.** |
| <vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset> | **These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The <vector> header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these header files in Chapter 23, Standard Template Library (STL).** |
| <cctype> | **Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <ctype.h>. These topics are discussed in Chapter 8, Pointers and Pointer-Based Strings, and Chapter 22, Bits, Characters, C-Strings and structs.** |
| <cstring> | **Contains function prototypes for C-style string-processing functions. This header file replaces header file <string.h>. This header file is used in Chapter 11, Operator Overloading; String and Array Objects.** |

**2 of 4)**

| C++ Standard Library header file | Explanation |
|---|---|
| `<typeinfo>` | Contains classes for runtime type identification (determining data types at execution time). This header file is discussed in Section 13.8. |
| `<exception>`, `<stdexcept>` | These header files contain classes that are used for exception handling (discussed in Chapter 16). |
| `<memory>` | Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling. |
| `<fstream>` | Contains function prototypes for functions that perform input from files on disk and output to files on disk (discussed in Chapter 17, File Processing). This header file replaces header file `<fstream.h>`. |
| `<string>` | Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18). |
| `<sstream>` | Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing). |
| `<functional>` | Contains classes and functions used by C++ Standard Library algorithms. This header file is used in Chapter 23. |

| C++ Standard Library header file | Explanation |
|---|---|
| <iterator> | Contains classes for accessing data in the C++ Standard Library containers. This header file is used in Chapter 23, Standard Template Library (STL). |
| <algorithm> | Contains functions for manipulating data in C++ Standard Library containers. This header file is used in Chapter 23. |
| <cassert> | Contains macros for adding diagnostics that aid program debugging. This replaces header file <assert.h> from pre-standard C++. This header file is used in Appendix F, Preprocessor. |
| <cfloat> | Contains the floating-point size limits of the system. This header file replaces header file <float.h>. |
| <climits> | Contains the integral size limits of the system. This header file replaces header file <limits.h>. |
| <cstdio> | Contains function prototypes for the C-style standard input/output library functions and information used by them. This header file replaces header file <stdio.h>. |
| <locale> | Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.). |
| <limits> | Contains classes for defining the numerical data type limits on each computer platform. |
| <utility> | Contains classes and functions that are used by many C++ Standard Library header files. |

# Math Library Functions

- **Global functions 全局函数**
  - **Do not belong to a particular class**
  - **Have function prototypes placed in header files**
    - **Can be reused in any program that includes the header file and that can link to the function's object code**
  - **Example: `sqrt` in `<cmath>` header file**
    - `sqrt( 900.0 )`
    - **All functions in `<cmath>` are global functions**

| Function | Description | Example |
|---|---|---|
| `ceil( x )` | rounds *x* to the smallest integer not less than *x* | `ceil( 9.2 )` is 10.0<br>`ceil( -9.8 )` is -9.0 |
| `cos( x )` | trigonometric cosine of *x* (*x* in radians) | `cos( 0.0 )` is 1.0 |
| `exp( x )` | exponential function $e^x$ | `exp( 1.0 )` is 2.71828<br>`exp( 2.0 )` is 7.38906 |
| `fabs( x )` | absolute value of *x* | `fabs( 5.1 )` is 5.1<br>`fabs( 0.0 )` is 0.0<br>`fabs( -8.76 )` is 8.76 |
| `floor( x )` | rounds *x* to the largest integer not greater than *x* | `floor( 9.2 )` is 9.0<br>`floor( -9.8 )` is -10.0 |
| `fmod( x, y )` | remainder of *x*/*y* as a floating-point number | `fmod( 2.6, 1.2 )` is 0.2 |
| `log( x )` | natural logarithm of *x* (base *e*) | `log( 2.718282 )` is 1.0<br>`log( 7.389056 )` is 2.0 |
| `log10( x )` | logarithm of *x* (base 10) | `log10( 10.0 )` is 1.0<br>`log10( 100.0 )` is 2.0 |
| `pow( x, y )` | *x* raised to power *y* ($x^y$) | `pow( 2, 7 )` is 128<br>`pow( 9, .5 )` is 3 |
| `sin( x )` | trigonometric sine of *x* (*x* in radians) | `sin( 0.0 )` is 0 |
| `sqrt( x )` | square root of *x* (where *x* is a nonnegative value) | `sqrt( 9.0 )` is 3.0 |
| `tan( x )` | trigonometric tangent of *x* (*x* in radians) | `tan( 0.0 )` is 0 |

# 6.7 Case Study: Random Number Generation

- C++ Standard Library function `rand` 随机数
  - Introduces the element of chance into computer applications
  - Example
    - `i = rand();`
    - Generates an unsigned integer between 0 and RAND_MAX (a symbolic constant defined in header file `<cstdlib>`)
  - Function prototype for the `rand` function is in `<cstdlib>`

# 6.7 Case Study: Random Number Generation (Cont.)

- To produce integers in a specific range, use the modulus operator (%) with `rand`
  - `number = shiftingValue + rand() % scalingFactor;`
  - Example

```
rand() % 6;
```

    - Produces numbers in the range 0 to 5

```
1 + rand() % 6;
```

    - Shifting can move the range to 1 to 6

```
10 #include <cstdlib> // contains function prototype for rand
11 using std::rand;
12
13 int main()
14 {
15   // loop 20 times
16   for ( int counter = 1; counter <= 20; counter++ )
17   {
18     // pick random number from 1 to 6 and output it
19     cout << setw( 10 ) << ( 1 + rand() % 6 );
21     // if counter is divisible by 5, start a new line of output
22     if ( counter % 5 == 0 )
23        cout << endl;
24   } // end for
26   return 0;
27 }
```

```
6     6     5     5     6
5     1     1     5     3
6     6     2     4     2
6     2     3     4     1
```

当次数足够多时，Fig 6.9程序表明骰子每个面出现次数基本相等

# 6.7 Case Study: Random Number Generation (Cont.)

- **Function `rand`**
  - Generates pseudorandom numbers 伪随机数
  - The same sequence of numbers repeats itself each time the program executes （程序6.8、6.9皆是）
- **Randomizing**
  - Conditioning a program to produce a different sequence of random numbers for each execution
- **C++ Standard Library function `srand`**
  - Takes an unsigned integer argument
  - Seeds the `rand` function to produce a different sequence of random numbers

```
// Fig. 6.10: fig06_10.cpp
11 #include <cstdlib>
12 using std::rand;
13 using std::srand;
15 int main()
16 {
17   unsigned seed;
19   cout << "Enter seed: ";
20   cin >> seed;
21   srand( seed );
23   // loop 10 times
24   for ( int counter = 1; counter <= 10; counter++ )
25   {
27     cout << setw( 10 ) << ( 1 + rand() % 6 );
30     if ( counter % 5 == 0 )
31        cout << endl;
32   }
34   return 0;
35}
```

生成一系列随机数只需要在开始时调用一次srand函数

**Common Programming Error6.7**

```
Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4

Enter seed: 432
          4          6          3          1          6
          3          1          5          4          2

Enter seed: 67
          6          1          4          6          2
          1          6          1          6          4
```

# 6.7 Case Study: Random Number Generation (Cont.)

- **To randomize without having to enter a seed each time**

```
srand( time( 0 ) );
```

  - **This causes the computer to read its clock to obtain the seed value**
  - **Function `time` (with the argument 0)**
    - **Returns the current time as the number of seconds since January 1, 1970 at midnight Greenwich Mean Time (GMT)**
      返回距离格林威治时间1970年1月1日零时的秒数
    - **Function prototype for `time` is in `<ctime>`**

## 6.8 Case Study: Game of Chance and Introducing enum

- 程序中周一至周日如何表示？（1, 2, 3, .., 7）
- 足球比赛程序中输、赢、平局比赛结果如何表示？（1代表输，2代表赢，3代表平局）
- 不直观，易出错
  - int day = 8 ?
  - int result = 0?
- 有没有更好的表示方法？
- MON, TUE, WED, THU, FRI, SAT, SUN
- LOST, WON, DRAW

# 6.8 Case Study: Game of Chance and Introducing enum

- **Enumeration 枚举**

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

C++关键词　　类型名　　花括号内定义逗号分割的标识符列表

  - **A set of integer constants represented by identifiers**
    - **The values of enumeration constants start at 0, unless specified otherwise, and increment by 1**

```
enum Day {SUN, MON, TUE, WED, THU, FRI, SAT};
enum Day {SUN=7, MON=1, TUE, WED, THU, FRI, SAT};
```

    - **The identifiers in an enum must be unique, but separate enumeration constants can have the same integer value**

```
enum Test {AA=2, BB=1, CC, DD};
```

参看6.8节游戏规则

```
1// Fig. 6.11: fig06_11.cpp
11 #include <ctime> // contains prototype for function time
12 using std::time;
13
14 int rollDice(); // rolls dice, calculates amd displays sum
15
16 int main()
17 {
18   // enumeration with constants that represent the game status
19   enum Status { CONTINUE, WON, LOST };
20
21   int myPoint;
22   Status gameStatus; // can contain CONTINUE, WON or LOST
25   srand( time( 0 ) );
26
27   int sumOfDice = rollDice(); // first roll of the dice
30   switch ( sumOfDice )
31   {
32     case 7: // win with 7 on first roll
33     case 11: // win with 11 on first roll
34       gameStatus = WON;
35       break;
```

```
49 while ( gameStatus == CONTINUE ) // not WON or LOST
50 {
51    sumOfDice = rollDice(); // roll dice again
53    // determine game status
54    if ( sumOfDice == myPoint ) // win by making point
55       gameStatus = WON;
56    else
57       if ( sumOfDice == 7 ) // lose by rolling 7 before point
58          gameStatus = LOST;
59 } // end while
```

使用枚举而不是常整形可以使得程序
更加清晰和易于维护。

# 6.9 Storage Classes 存储类别

- **Each identifier has several attributes（属性）**
  - **Name, type, size and value**
  - **Also storage class(存储类别), scope(作用域) and linkage(连接)**
- **C++ provides five storage-class specifiers:**
  - `auto`, `register`, `extern`, `mutable` and `static`
- **Identifier's storage class（标识符存储类别）**
  - **Determines the period during which that identifier exists in memory（内存中存在的时间）**
- **Identifier's scope （标识符作用域）**
  - **Determines where the identifier can be referenced in a program（可以被引用的范围）**

# 6.9 Storage Classes (Cont.)

- **Identifier's linkage（标识符的连接）**
  - **Determines whether an identifier is known only in the source file where it is declared or across multiple files that are compiled, then linked together**（决定了标识符是仅在声明的源文件中可以被识别还是在编译后连接在一起的多个文件中可以被识别）
- **An identifier's storage-class specifier helps determine its storage class and linkage**
  - **Static (静态存储类型) and Automatic (自动存储类型)**

# 6.9 Storage Classes (Cont.)

- **Automatic storage class自动存储类型**
  - **Declared with keywords `auto` and `register`**
  - **Automatic variables 自动变量**
    - **Created when program execution enters block in which they are defined**
    - **Exist while the block is active**
    - **Destroyed when the program exits the block**
  - **Only local variables and parameters can be of automatic storage class 仅有函数的局部变量和形参能成为自动存储类型**
    - **Such variables normally are of automatic storage class**

# 6.9 Storage Classes (Cont.)

- **Static storage class静态存储类型**
  - **Declared with keywords `extern` and `static`**
  - **Static-storage-class variables 静态存储类型变量**
    - **Exist from the point at which the program begins execution 程序开始的时候分配空间**
    - **Initialized once when their declarations are encountered 只初始化一次**
    - **Last for the duration of the program 程序运行时都存在**
  - **Static-storage-class functions静态存储类型函数**
  - **However, even though the variables and the function names exist from the start of program execution, this does not mean that these identifiers can be used throughout the program. 但并不表示总能使用**

# 6.9 Storage Classes (Cont.)

- Two types of identifiers with static storage class
  - ① External identifiers 外部标识符
    - Such as global variables 全局变量 and global function 全局函数
  - ② Local variables declared with the storage class specifier `static` 用`static`声明的局部变量
- Global variables 全局变量
  - Created by placing variable declarations **outside** any class or function definition 在类和函数之外定义的变量
  - Retain their values throughout the execution of the program 整个程序运行期间都保留其数值
  - Can be referenced by any function that follows their declarations or definitions in the source file 能够在定义的源文件中的任意函数中使用

# 6.9 Storage Classes (Cont.)

- **Local variables declared with keyword `static`**
  **用`static`声明的局部变量（重点）**
  - **Known only in the function in which they are declared**
    **仅在它所定义的函数中可见**
  - **Retain their values when the function returns to its caller**
    **当函数调用结束返回时保留其中的数值**
    - **Next time the function is called, the `static` local variables contain the values they had when the function last completed**
  - **If numeric variables of the static storage class are not explicitly initialized by the programmer**
    - **They are initialized to zero 缺省初始化为零**

# 6.10 Scope Rules 作用域规则

- **Scope**
  - **Portion of the program where an identifier can be used**
  - **Four scopes for an identifier**
    - **File scope**文件作用域 → **global namespace scope** 全局名字空间作用域
    - **Block scope**块作用域 →**local scope** 局部作用域
    - ~~**Function scope**函数作用域~~
    - ~~**Function-prototype scope**函数原型作用域~~

# 6.10 Scope Rules (Cont.)

- **File scope or Global namespace scope文件作用域 或全局作用域**
  - **For an identifier declared outside any function or class**
    - **Such an identifier is "known" in all functions from the point at which it is declared until the end of the file 从标识符声明的位置开始直至文件末尾**
  - **Global variables, function definitions and function prototypes placed outside a function all have file scope**

# 6.10 Scope Rules (Cont.)

- **Block scope or Local scope块作用域 或局部作用域（重点）**
  - **Identifiers declared inside a block have block scope**
    - **Block scope begins at the identifier's declaration**
    - **Block scope ends at the terminating right brace (}) of the block in which the identifier is declared**
  - **Local variables and function parameters have block scope**
    **局部变量和函数形参具有块作用域**
    - **The function body is their block**
  - **Any block can contain variable declarations**
  - **Identifiers in an outer block can be "hidden" when a nested block has a local identifier with the same name 当内层块中的标识符与外层块中的同名时，外层块中的同名标识符被隐藏**
  - **Local variables declared `static` still have block scope, even though they exist from the time the program begins execution**
    - **Storage duration does not affect the scope of an identifier**

# 6.10 Scope Rules (Cont.)

- File scope文件作用域
  - For an identifier declared outside any function or class
    - Such an identifier is “known” in all functions from the point at which it is declared until the end of the file
  - Global variables, function definitions and function prototypes placed outside a function all have file scope
- Function scope函数作用域
  - Labels (identifiers followed by a colon such as `start:`) are the only identifiers with function scope
    - Can be used anywhere in the function in which they appear
    - Cannot be referenced outside the function body
    - Labels are implementation details that functions hide from one another

# 6.10 Scope Rules (Cont.)

- **Function-prototype scope函数原型作用域**
  - **Only identifiers used in the parameter list of a function prototype have function-prototype scope**
  - **Parameter names appearing in a function prototype are ignored by the compiler**
    - **Identifiers used in a function prototype can be reused elsewhere in the program without ambiguity**
    - **However, in a single prototype, a particular identifier can be used only once**
- **Example**

  **int f(int a, int b, float c);**

```
1 // Fig. 6.12: fig06_12.cpp
2 // A scoping example.
7 void useLocal( void ); // function prototype
8 void useStaticLocal( void ); // function prototype
9 void useGlobal( void ); // function prototype
10
11int x = 1; // global variable
12
13int main()
14{
15   int x = 5; // local variable to main
17   cout << "local x in main's outer scope is " << x <<
endl;
18
19   { // start new scope
20      int x = 7; // hides x in outer scope
22      cout << "local x in main's inner scope is " << x
<< endl;
23   } // end new scope
24
25   cout << "local x in main's outer scope is " << x <<
endl;
```

```
26
27    useLocal(); // useLocal has local x
28    useStaticLocal(); // useStaticLocal has static local x
29    useGlobal(); // useGlobal uses global x
30    useLocal(); // useLocal reinitializes its local x
31    useStaticLocal(); // static local x retains its prior value
32    useGlobal(); // global x also retains its value
33
34    cout << "\nlocal x in main is " << x << endl;
35    return 0; // indicates successful termination
36} // end main
37
38// useLocal reinitializes local variable x during each call
39void useLocal( void )
40{
41    int x = 25; // initialized each time useLocal is called
42
43    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44    x++;
45    cout << "local x is " << x << " on exiting useLocal" << endl;
46} // end function useLocal
```

```
47
48// useStaticLocal initializes static local variable x only the
49// first time the function is called; value of x is saved
50// between calls to this function
51void useStaticLocal( void )
52{
53   static int x = 50; // initialized first time useStaticLocal is
called
54
55   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56      << endl;
57   x++;
58   cout << "local static x is " << x << " on exiting useStaticLocal"
59      << endl;
60} // end function useStaticLocal
61
62// useGlobal modifies global variable x during each call
63void useGlobal( void )
64{
65   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66   x *= 10;
67   cout << "global x is " << x << " on exiting useGlobal" << endl;
68} // end function useGlobal
```

```
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

# 6.16 Unary Scope Resolution Operator
# 一元作用域分辨运算符

- **Unary scope resolution operator (::)**
  - **Used to access a global variable when a local variable of the same name is in scope**
  - **Cannot be used to access a local variable of the same name in an outer block 不能用来访问外层语句块中同名变量**
- **Always using the unary scope resolution operator (::) to refer to global variables makes programs easier to modify by reducing the risk of name collisions with nonglobal variables.**

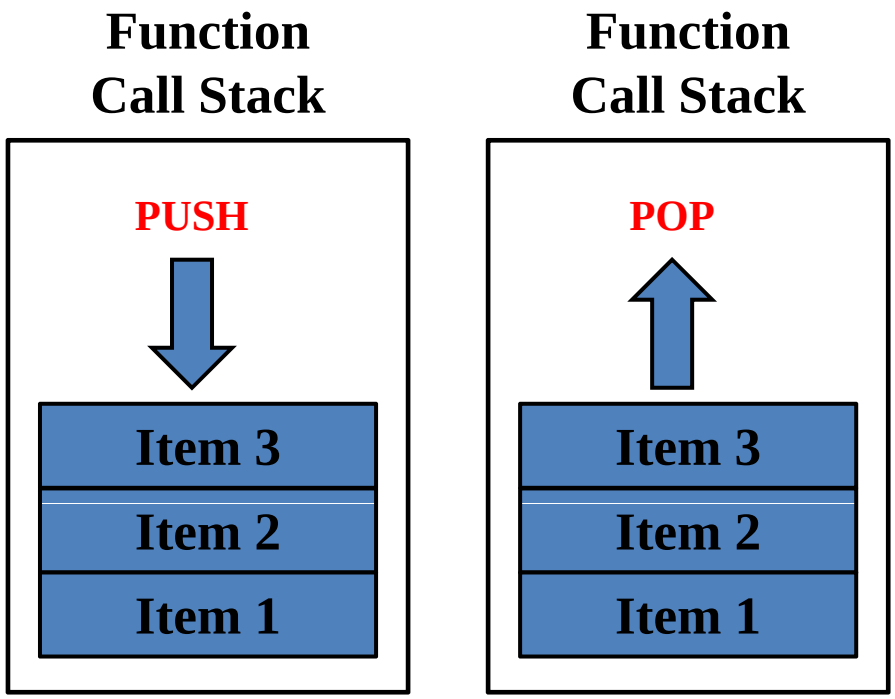
```cpp
// Fig. 6.23: fig06_23.cpp
// Using the unary scope resolution operator.
#include <iostream>
using std::cout;
using std::endl;

int number = 7; // global variable named number

int main()
{
   double number = 10.5; // local variable named number

   // display values of local and global variables
   cout << "Local double value of number = " << number
      << "\nGlobal int value of number = " << ::number << endl;
   return 0; // indicates successful termination
} // end main
```

```
Local double value of number = 10.5
Global int value of number = 7
```

# 6.11 Function Call Stack and Activation Records 函数调用堆栈及活动记录

- **Data structure:**相互之间存在一种或多种特定关系的数据元素的集合
- **Stack data structure** 堆栈数据结构
  - **Analogous to a pile of dishes**
  - **When a dish is placed on the pile, it is normally placed at the top**
    - **Referred to as pushing the dish (item) onto the stack**
  - **Similarly, when a dish is removed from the pile, it is normally removed from the top**
    - **Referred to as popping the dish (item) off the stack**
  - **A last-in, first-out (LIFO) data structure**
    - **The last item pushed (inserted) on the stack is the first item popped (removed) from the stack**

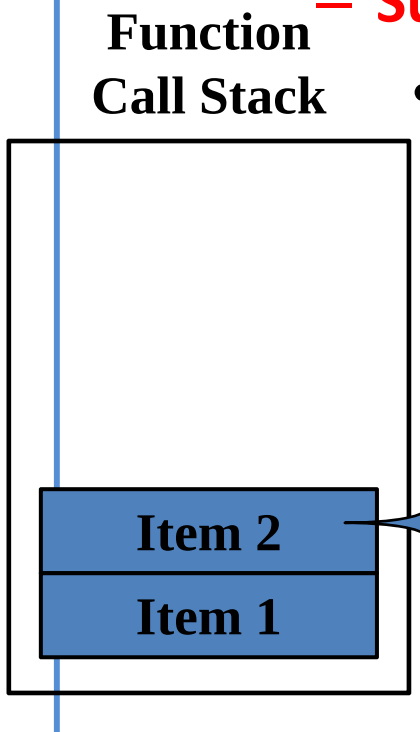
# 6.11 Function Call Stack and Activation Records




LIFO - Last In First Out

# 6.11 Function Call Stack and Activation Records (Cont.)

- **Function Call Stack 函数调用堆栈**
  - **Sometimes called the program execution stack**
  - **Supports the function call/return mechanism**

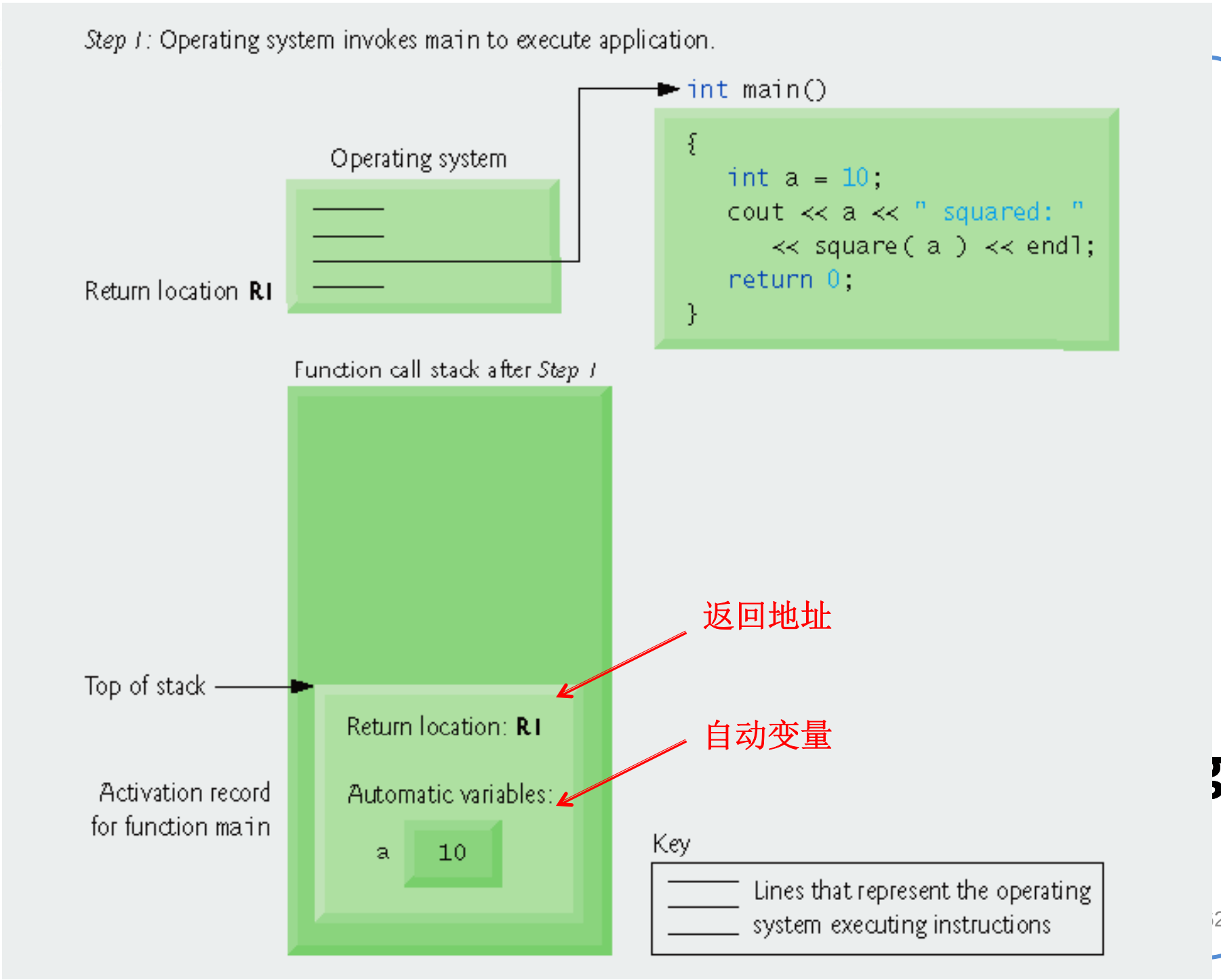


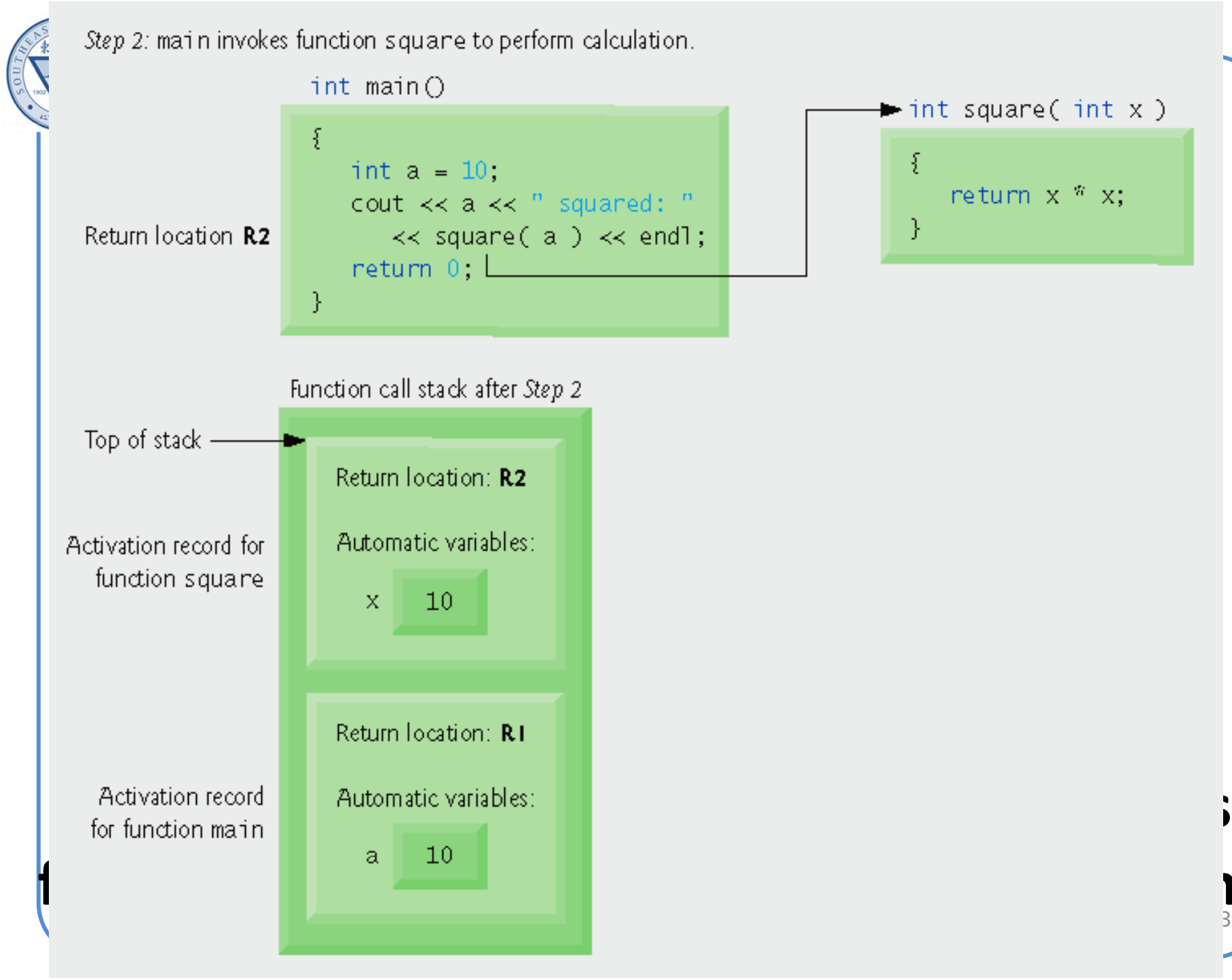

- **Each time a function calls another function, a stack frame (activation record 活动记录) is pushed onto the stack. It includes:**
  - **① Maintains the return address that the called function needs to return to the calling function（主调函数）**
  - **② Contains automatic variables—parameters and any local variables the function declares**

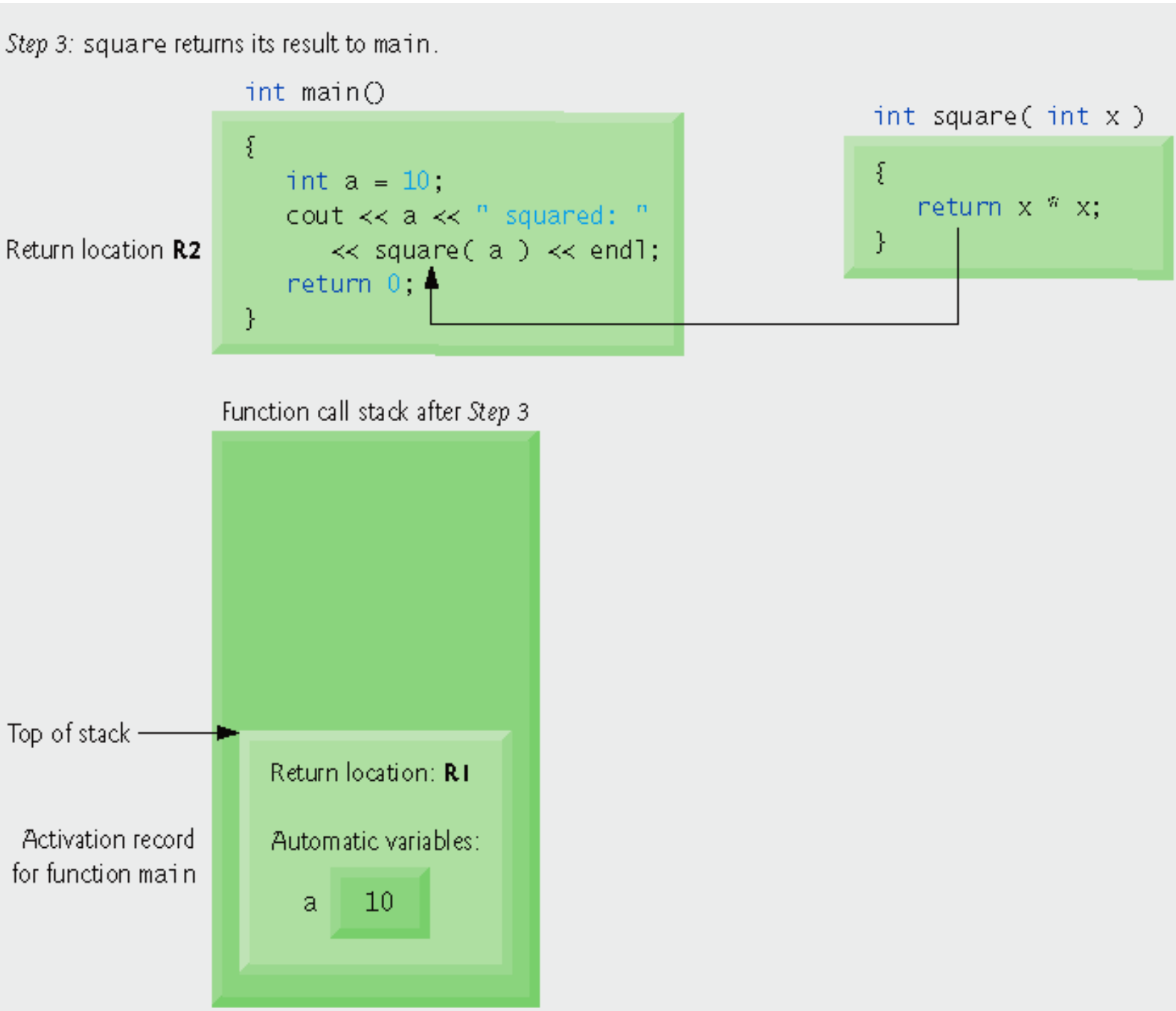
# 6.11 Function Call Stack and Activation Records (Cont.)

- Function Call Stack (Cont.)
  - If a function makes a call to another function => PUSH 调用
    - Stack frame for the new function call is simply pushed onto the call stack
    - Return address required by the newly called function to return to its caller is now located at the top of the stack.
  - When the called function returns => POP 调用结束返回
    - Stack frame for the function call is popped
    - Control transfers to the return address in the popped stack frame
- Stack overflow 堆栈溢出错误

```
1// Fig. 6.13: fig06_13.cpp
9  int square( int ); // prototype for function square
10
11int main()
12{
13   int a = 10;
14
15   cout << a << " squared: " << square( a ) << endl;
16   return 0;
17}
18
19// returns the square of an integer
20int square( int x )
21{
22    return x * x;
23}
```


Step 1: Operating system invokes main to execute application.
int main()
{
int a = 10;
cout << a << " squared: "
<< square( a ) << endl;
return 0;
}
Operating system
Return location R1
Function call stack after Step 1
返回地址
Top of stack
Return location: R1
自动变量
Activation record
for function main
Automatic variables:
a
10
Key
Lines that represent the operating
system executing instructions


Step 2: main invokes function square to perform calculation.
int main()
{
int a = 10;
cout << a << " squared: "
<< square( a ) << endl;
return 0;
}
Return location R2
int square( int x )
{
return x * x;
}
Function call stack after Step 2
Top of stack
Return location: R2
Automatic variables:
x 10
Activation record for function square
Return location: R1
Automatic variables:
a 10
Activation record for function main


Step 3: square returns its result to main.
int main()
{
int a = 10;
cout << a << " squared: "
<< square( a ) << endl;
return 0;
}
Return location R2
int square( int x )
{
return x * x;
}
Function call stack after Step 3
Top of stack
Return location: R1
Automatic variables:
a 10
Activation record for function main

# 6.13 Inline Functions内联函数

– **Reduce function call overhead—especially for small functions** 减少函数调用的开销

请大家自学！

# 6.13 Inline Functions内联函数

- Reduce function call overhead—especially for small functions 减少函数调用的开销
- Qualifier `inline` before a function's return type in the function definition 在函数返回值之前使用限定符inline
  - "Advises" the compiler to generate a copy of the function's code in place (when appropriate) to avoid a function call
- Trade-off of inline functions
  - Multiple copies of the function code are inserted in the program (often making the program larger)
- The compiler can ignore the `inline` qualifier and typically does so for all but the smallest functions

# 6.14 References and Reference Parameters 引用与引用形参

- Two ways to pass arguments to functions
  - Pass-by-value 传值调用
    - A copy of the argument's value is passed to the called function
    - Changes to the copy do not affect the original variable's value in the caller 改变形参的值不会影响实际参数
      - Prevents accidental side effects of functions
  - Pass-by-reference 传引用调用
    - Gives called function the ability to access and modify the caller's argument data directly

# Performance Tip 6.5

One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.

# 6.14 References and Reference Parameters (Cont.)

- **References 引用**
  - **Can also be used as aliases (别名) for other variables within a function**
    - **All operations supposedly performed on the alias (i.e., the reference) are actually performed on the original variable**
    - **An alias is simply another name for the original variable**
    - **Must be initialized in their declarations 必须在声明时初始化**
      - **Cannot be reassigned afterward 定义后不能重新被初始化**
  - **Example**

```
int count = 1;
int &cRef = count;
cRef++;
```

```
7  int main()
8  {
9    int x = 3;
10   int &y = x;
11
12   cout << "x = " << x << endl << "y = " << y << endl;
13   y = 7;
14   cout << "x = " << x << endl << "y = " << y << endl;
15   return 0;
16 }
```

int &y;

```
x = 3
y = 3
x = 7
y = 7
```

C:\cpphtp6_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' : references must be initialized

# 6.14 References and Reference Parameters (Cont.)

- **Reference Parameter 引用形参**
  - **An alias (别名) for its corresponding argument in a function call**
  - **& placed after the parameter type in the function prototype and function header**
  - **Example**

```
int f(int &count)
```

  - **Parameter name in the body of the called function actually refers to the original variable in the calling function 在被调用函数体中对引用形参的操作就是对主调函数中实参本身的操作**

# Performance Tip 6.6

**Pass-by-reference is good for performance reasons, because it can eliminate the pass-by-value overhead of copying large amounts of data.**

```
int squareByValue( int number )
{
    return number *= number;
}
void squareByReference( int &numberRef )
{
    numberRef *= numberRef;
}

int main()
{
  int x = 2, z=4, r;
  r = squareByValue(x);
  squareByReference(z);
}
```

# Common Programming Error 6.14

**Because reference parameters are mentioned only by name in the body of the called function, the programmer might inadvertently treat reference parameters as pass-by-value parameters. This can cause unexpected side effects if the original copies of the variables are changed by the function.**

程序员可能在不经意间把引用形参当做按值传递的形参。这样，如果函数改变了原始的变量副本，就可能产生不可预期的副作用。

## Software Engineering Observation 6.14

Keyword `const` would protect only a copy of the original argument, not the original argument itself, which when passed by value is safe from modification by the called function. 使用const定义为常量。

```
const int a = 5;
void f (const int a);
void f (const int& b);
```

# Software Engineering Observation 6.15

**For the combined reasons of clarity and performance, many C++ programmers prefer that modifiable arguments be passed to functions by using pointers (which we study in Chapter 8), small nonmodifiable arguments be passed by value and large nonmodifiable arguments be passed to functions by using references to constants.**

会改变值的实参——用指针
不会改变值的小型实参——用传值
不会改变值的大型实参——用常数型的引用

# 6.14 References and Reference Parameters (Cont.)

- **Returning a reference from a function**
  - Functions can return references to variables
    - Should only be used when the variable is `static`
      作为返回值的变量必须是`static`类型
  - Dangling reference
    - Returning a reference to an automatic variable
      - That variable no longer exists after the function ends

# 6.15 Default Arguments 默认实参

- **Default argument**
  - **A default value to be passed to a parameter**
    - **Used when the function call does not specify an argument for that parameter**
  - **Must be the rightmost argument(s) in a function's parameter list** 形参列表中最右边的形参
  - **Should be specified with the first occurrence of the function name** 必须在函数名第一次出现时指定
    - **Typically the function prototype** 通常在函数原型中指定

```
int boxVolume( int length = 1, int width = 1, int height = 1 );
```

# 6.17 Function Overloading 函数重载

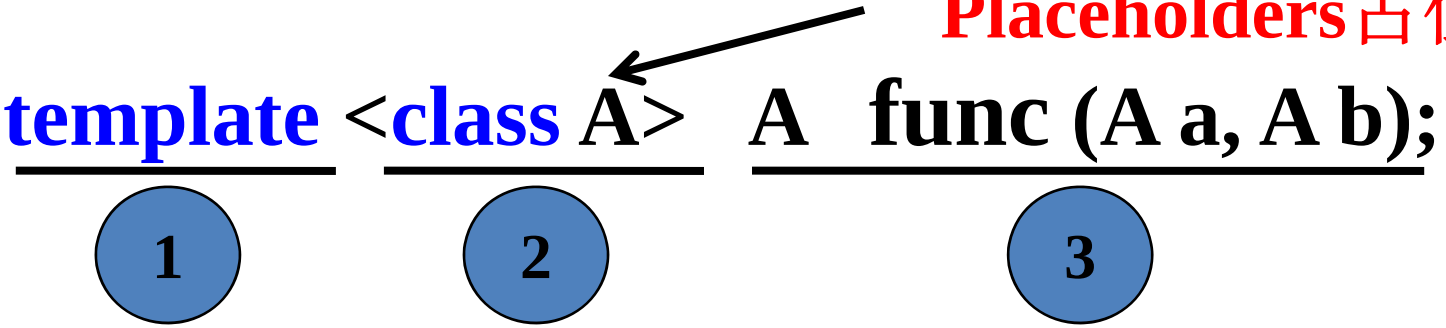
- **Overloaded functions**
  - **Overloaded functions have**
    - **Same name** 函数名相同
    - **Different sets of parameters** 形参不同（函数签名不同）
  - **Compiler selects proper function to execute based on number, types and order of arguments in the function call** 编译器根据实参类型自动选择合适的函数调用，如没有严格匹配的，按隐式升级规则选择函数

```
7  // function square for int values
8  int square( int x )
9  {
10         cout << "square of integer " << x << " is ";
11   return x * x;
12}
14// function square for double values
15double square( double y )
16{
17   cout << "square of double " << y << " is ";
18   return y * y;
19} // end function square with double argument

21int main()
22 {
23   cout << square( 7 ); // calls int version
24   cout << endl;
25   cout << square( 7.5 ); // calls double version
26   cout << endl;
27   return 0;
28}
```

函数名相同，
形参不同

# 6.18 Function Templates 函数模板

- More compact and convenient form of overloading




① Begins with the `template` keyword

② template parameter list of formal type parameters ( 形式类型参数)

③ Function prototype ( Placeholders for fundamental types or user-defined types)

# Common Programming Error 6.23

**template <class A, class B>  A  func (A a, B b)**

```
template <class S, T>...
```

X

```
template <class S, class T>...
```

√

```
4 template < class T >  // or template< typename T >
5 T maximum( T value1, T value2, T value3 )
6 {
7   T maximumValue = value1;
8
10  if ( value2 > maximumValue )
11        maximumValue = value2;
12
14  if ( value3 > maximumValue )
15        maximumValue = value3;
16
17  return maximumValue;
18}
```

# 6.18 Function Templates (Cont.)

- **Function-template specializations 特化**
  - Generated automatically by the compiler to handle each type of call to the function template
  - Example for function template `max` with type parameter `T` called with `int` arguments
    - Compiler detects a `max` invocation（调用） in the program code
    - `int` is substituted （代替）for `T` throughout the template definition
    - This produces function-template specialization （特化）`max< int >`

```
int int1=2, int2=3, int3=1;
int int_max = maximum(int1, int2, int3);

double double1=1.2, double2=2.3, double3=4.3;
double double_max = maximum(double1,
   double2, double3);

char char1='A', char2='B', char3='D';
char char_max = maximum(char1, char2, char3);
```

# 6.19 Recursion 递归

- **Recursive function**
  - **A function that calls itself, either directly, or indirectly (through another function)** 一个函数直接或间接地调用自己的函数

```
void f()
{
  …
  f();
  …
}
```

直接

```
void f() {
  …
  h();
  …
}
void h()
{
  …
  f();
  …
}
```

间接

# 6.19 Recursion 递归

- 利用递归函数计算阶乘

**n! = n(n-1)!= n(n-1)(n-2)! = n(n-1)(n-2)(n-3)!**

**…= n(n-1)(n-2)…1!**

1、问题的难度（规模）在逐步降低（简化）

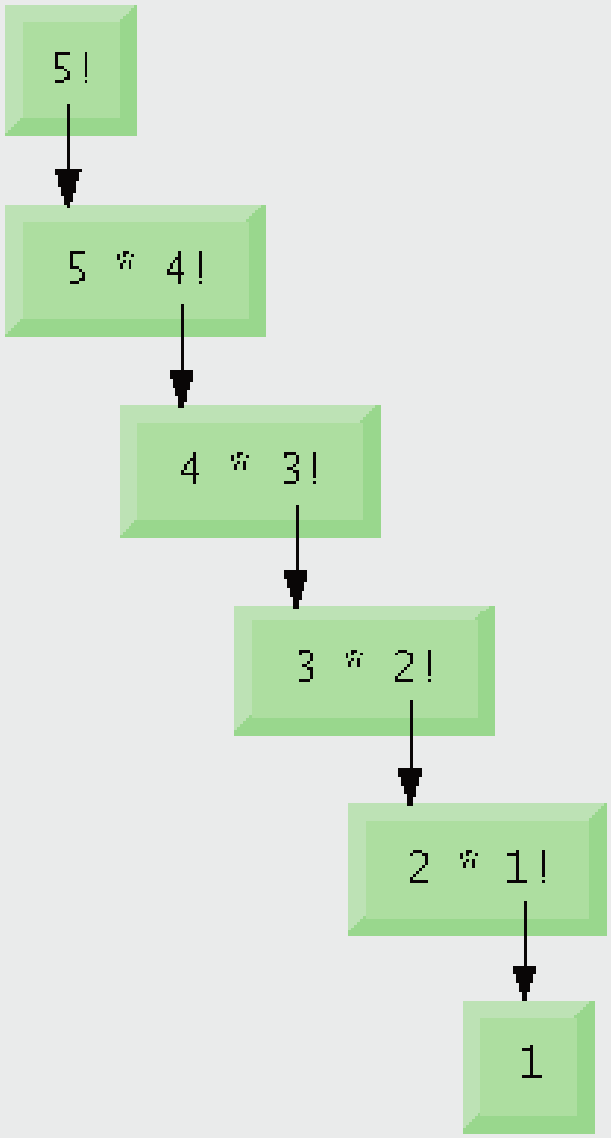
2、最终简化到一个易于求解的基本问题

```
unsigned long factorial( unsigned long number )
{
    if ( number <= 1 )
      return 1;
    else
      return number * factorial( number - 1 );
}
```

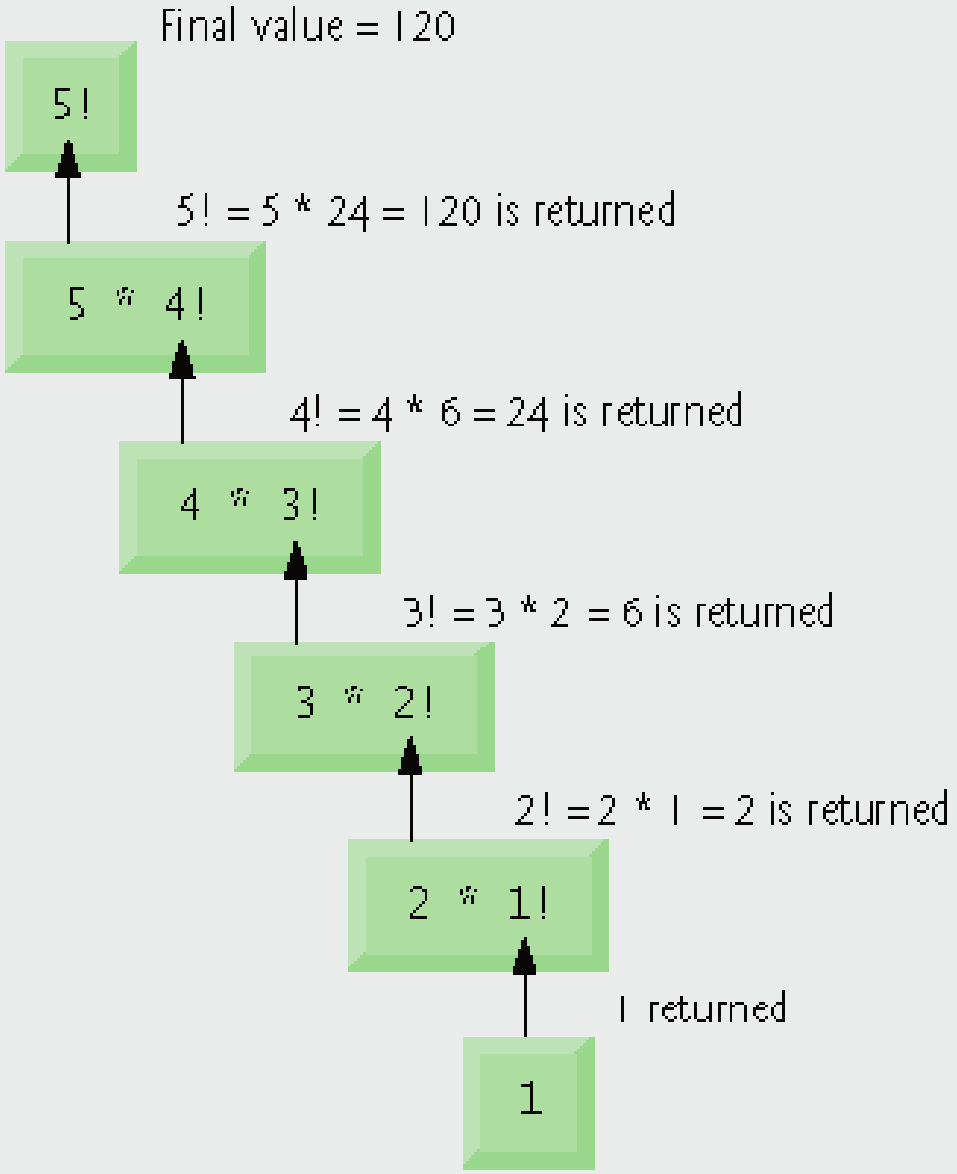
# 6.19 Recursion 递归

- Recursion递归算法的核心思想
  - Base case(s) 基本情况 1!=1
    - The simplest case(s), which the function knows how to handle
  - For all other cases, the function typically divides the problem into two conceptual pieces n!=n·(n-1)!
    - A piece that the function knows how to do
    - A piece that it does not know how to do

因为递归函数中需要判断当前是否是基本情况，所以通常递归函数都是采用if…else…结构




(a) Procession of recursive calls.



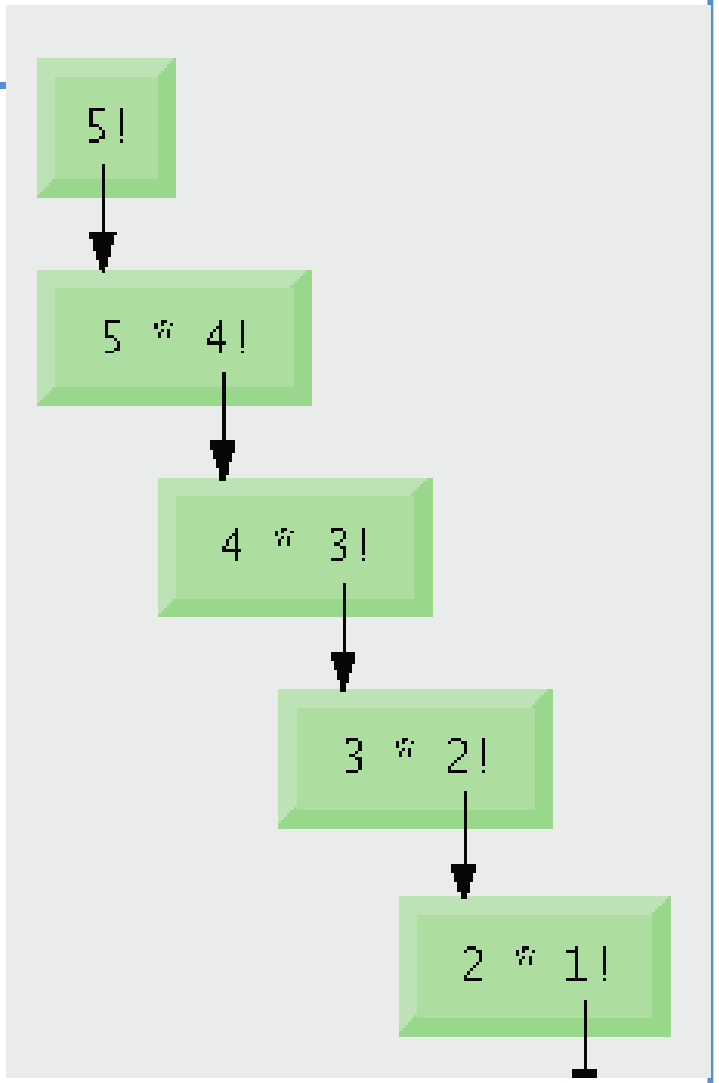

(b) Values returned from each recursive call.

# 利用VC调用堆栈来观察递归函数的调用

上课演示.exe!factorial(unsigned long number=5) 行27

上课演示.exe!factorial(unsigned long number=4) 行27
上课演示.exe!factorial(unsigned long number=5) 行30 + 0xc 字节

上课演示.exe!factorial(unsigned long number=1) 行27
上课演示.exe!factorial(unsigned long number=2) 行30 + 0xc 字节
上课演示.exe!factorial(unsigned long number=3) 行30 + 0xc 字节
上课演示.exe!factorial(unsigned long number=4) 行30 + 0xc 字节
上课演示.exe!factorial(unsigned long number=5) 行30 + 0xc 字节



# 6.20 Example Using Recursion: Fibonacci Series

- The Fibonacci series斐波那契数列
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, …
  - can be defined recursively as follows:
    - fibonacci(0) = 0
    - fibonacci(1) = 1
    - fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)

如何定义计算斐波那契数列的递归函数

```
unsigned long fibonacci( unsigned long number )
{
  if (________1_________) // base cases
    _______2_________;
  else // recursion step
    ________________3_________________;
}
```

```
unsigned long fibonacci( unsigned long number )
{
  if ( ( number == 0 ) || ( number == 1 ) ) // base cases
    return number;
  else // recursion step
    return fibonacci( number - 1 ) + fibonacci( number - 2 );
}
```

**unsigned long:**无符号长整型: 没有符号位

表示范围与机器、编译器相关，通常情况下

表示范围：**0 ~ 4294967295(2^32-1)**

**int:**

表示范围：**-2147483648 ~ 2147483647**

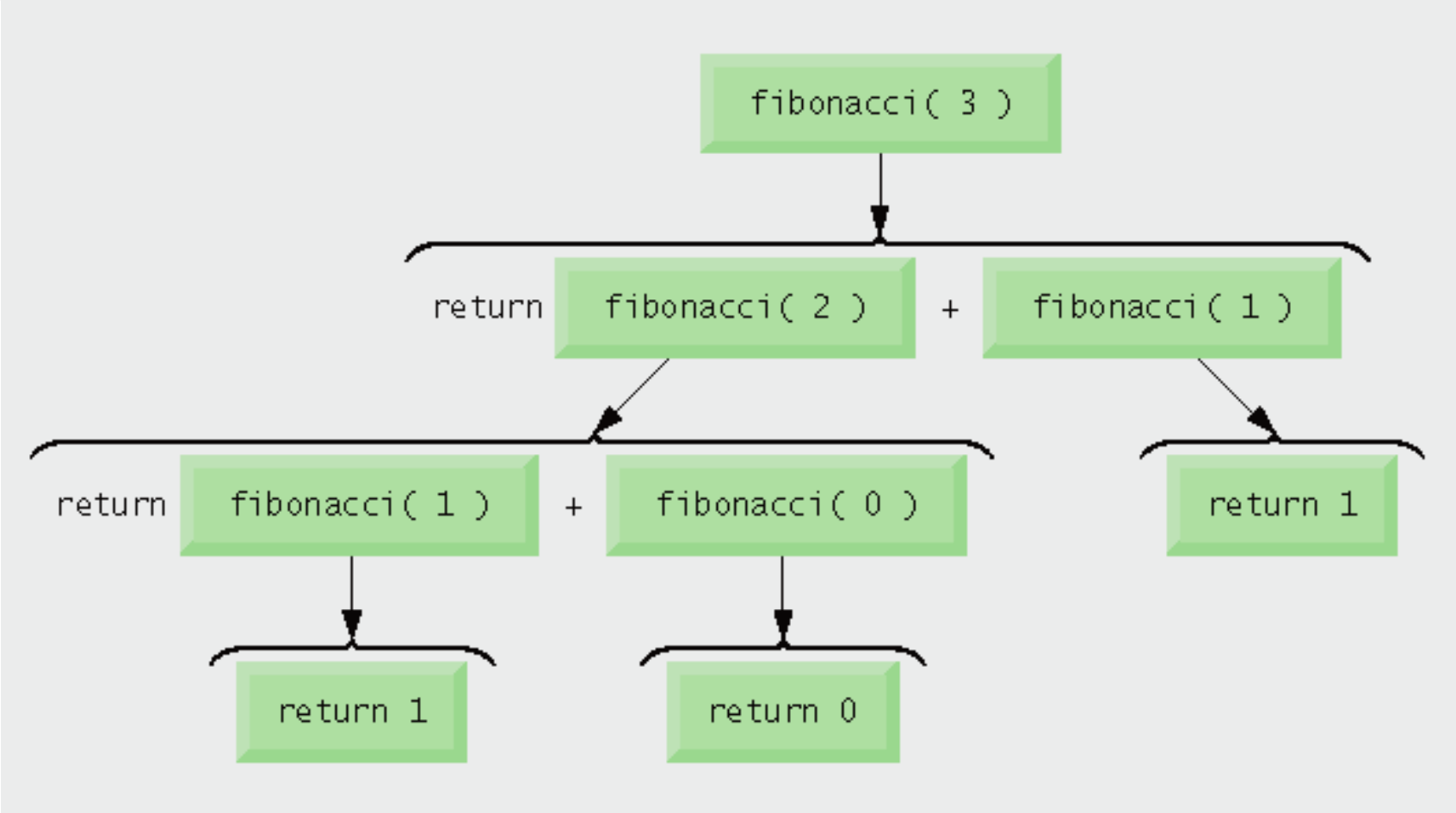
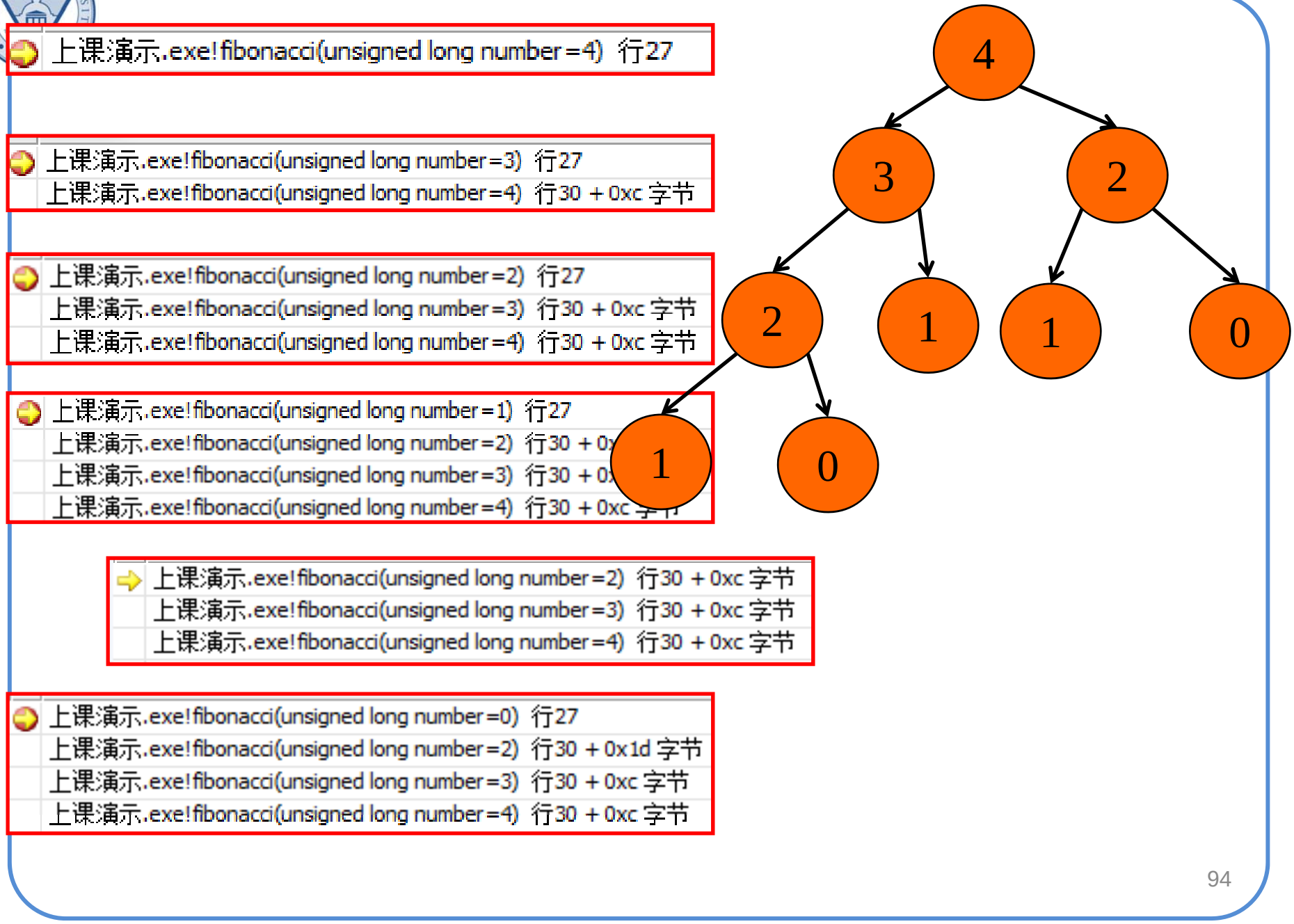



Fig. 6.31 | Set of recursive calls to function `fibonacci`.


上课演示.exe!fibonacci(unsigned long number=4) 行27
上课演示.exe!fibonacci(unsigned long number=3) 行27
上课演示.exe!fibonacci(unsigned long number=4) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=2) 行27
上课演示.exe!fibonacci(unsigned long number=3) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=4) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=1) 行27
上课演示.exe!fibonacci(unsigned long number=2) 行30 + 0x
上课演示.exe!fibonacci(unsigned long number=3) 行30 + 0x
上课演示.exe!fibonacci(unsigned long number=4) 行30 + 0xc
上课演示.exe!fibonacci(unsigned long number=2) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=3) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=4) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=0) 行27
上课演示.exe!fibonacci(unsigned long number=2) 行30 + 0x1d 字节
上课演示.exe!fibonacci(unsigned long number=3) 行30 + 0xc 字节
上课演示.exe!fibonacci(unsigned long number=4) 行30 + 0xc 字节
4
3
2
2
1
1
0
1
0

# 6.20 Example Using Recursion: Fibonacci Series (Cont.)

- **Caution about recursive programs**
  - **Each level of recursion in function `fibonacci` has a doubling effect on the number of function calls**
    - **i.e., the number of recursive calls that are required to calculate the *n*th Fibonacci number is on the order of $2^n$**
    - **20th Fibonacci number would require on the order of $2^{20}$ or about a million calls**
    - **30th Fibonacci number would require on the order of $2^{30}$ or about a billion calls.**
  - **Exponential complexity 指数型复杂度**
    - **Can humble even the world's most powerful computers**

# 6.21 Recursion vs. Iteration

- **Both are based on a control statement**
  - **Iteration – repetition structure**
  - **Recursion – selection structure**
- **Both involve repetition**
  - **Iteration – explicitly uses repetition structure**
  - **Recursion – repeated function calls**
- **Both involve a termination test**
  - **Iteration – loop-termination test**
  - **Recursion – base case**

# 6.21 Recursion vs. Iteration (Cont.)

- Both gradually approach termination
  - Iteration modifies counter until loop-termination test fails
  - Recursion produces progressively simpler versions of problem
- Both can occur infinitely
  - Iteration – if loop-continuation condition never fails
  - Recursion – if recursion step does not simplify the problem

# 6.21 Recursion vs. Iteration (Cont.)

- **Negatives of recursion 递归的缺点**
  - **Overhead of repeated function calls**
    - **Can be expensive in both processor time and memory space**
  - **Each recursive call causes another copy of the function (actually only the function's variables) to be created**
    - **Can consume considerable memory**
- **Iteration**
  - **Normally occurs within a function**
  - **Overhead of repeated function calls and extra memory assignment is omitted**

## Software Engineering Observation 6.18

**Any problem that can be solved recursively can also be solved iteratively (non-recursively).** A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution is not apparent.

| Location in Text | Recursion Examples and Exercises |
|---|---|
| *Chapter 6* | |
| Section 6.19, Fig. 6.29 | Factorial function |
| Section 6.19, Fig. 6.30 | Fibonacci function |
| Exercise 6.7 | Sum of two integers |
| Exercise 6.40 | Raising an integer to an integer power |
| Exercise 6.42 | Towers of Hanoi |
| Exercise 6.44 | Visualizing recursion |
| Exercise 6.45 | Greatest common divisor |
| Exercise 6.50, Exercise 6.51 | Mystery “What does this program do?” exercise |

Fig. 6.33 | Summary of recursion examples and exercises in the text. (Part 1 of 3)

| Location in Text | Recursion Examples and Exercises |
|---|---|
| *Chapter 7* | |
| Exercise 7.18 | Mystery “What does this program do?” exercise |
| Exercise 7.21 | Mystery “What does this program do?” exercise |
| Exercise 7.31 | Selection sort |
| Exercise 7.32 | Determine whether a string is a palindrome |
| Exercise 7.33 | Linear search |
| Exercise 7.34 | Binary search |
| Exercise 7.35 | Eight Queens |
| Exercise 7.36 | Print an array |
| Exercise 7.37 | Print a string backward |
| Exercise 7.38 | Minimum value in an array |
| *Chapter 8* | |
| Exercise 8.24 | Quicksort |
| Exercise 8.25 | Maze traversal |
| Exercise 8.26 | Generating Mazes Randomly |
| Exercise 8.27 | Mazes of Any Size |

# 汉诺塔 Hanoi

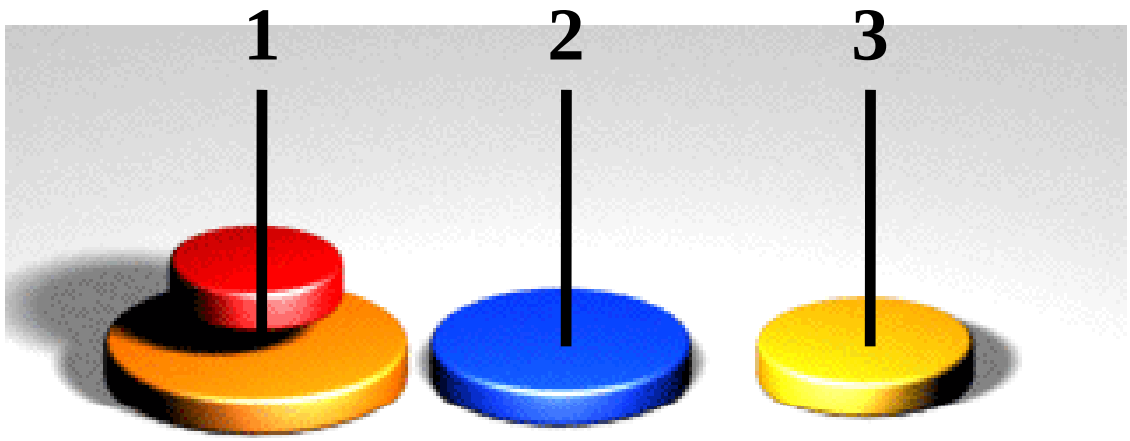
有三根杆子A，B，C。A杆上有N个(N>1)穿孔圆盘，盘的尺寸由下到上依次变小。

# 游戏规则

- 要求按下列规则将所有圆盘移至C杆：
  - 每次只能移动一个圆盘；
  - 大盘不能叠在小盘上面；
- 提示：可将圆盘临时置于B杆，也可将从A杆移出的圆盘重新移回A杆，但都必须尊循上述两条规则。

# 问题

- 最少要移动多少次？如何移？
  - $2^N-1$次
  - N=64，每秒移动一个盘子，需要5845.54亿年
  - N=20，超过一百万次



# 如何以递归形式实现？

- 步骤：
  1. 移动N-1个盘子从A杆到B杆
  2. 移动最后一个盘子（最大的）从A杆到C杆
  3. 移动N-1个盘子从B杆到C杆
- 当N=1的时候是最简单的情况，即直接将盘子从当前所在的杆移至目标杆

```
// towers recursively prints instructions for moving disks from
// start peg to end peg using temp peg for temporary storage
void towers( int disks, int start, int end, int temp )
{
  if ( disks == 1 ) // base case
    ____________________________;
  else // recursion step
  {
    // move disks - 1 disks from start to temp
    ____________________________;
    // move last disk from start to end
    ____________________________;
    // move disks - 1 disks from temp to end
    ____________________________;
  } // end else
} // end function towers
```