Chapter 6
Functions and an Introduction to Recursion

- □ To construct programs modularly from functions.
- □ 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.

Topics

- **6.1 Introduction**
- 6.2 Program Components in C++
- 6.3 Math Library Functions
- 6.4 Function Definitions with Multiple Parameters
- 6.5 Function Prototypes and Argument Coercion
- 6.6 C++ Standard Library Header Files
- 6.7 Case Study: Random Number Generation
- 6.8 Case Study: Game of Chance and Introducing enum
- 6.9 Storage Classes
- 6.10 Scope Rules

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.1 Introduction

- ❑ How to develop and maintain large programs?
- ❑ divide and conquer(分治法), 模块化编程
  - ❖ 使程序由小到大构建，易于管理
  - ❖ software reusability
  - ❖ avoid repeating code
  - ❖ makes the program easier to debug and maintain.
- ❑ Module: Classes + Functions

6.1 Introduction

- □ GradeBook::dispalyMessage()

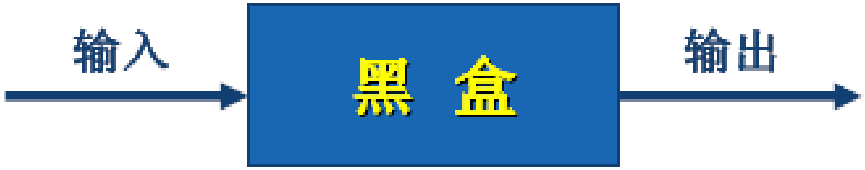
  类成员函数

- □ int main()

  Global Function(全局函数), 不是任何类的成员的函数. 如：<cmath>中的pow()函数

6.1 Introduction

- ❑Function (函数)
- ❑• 完成一项特定任务的代码行串, 由一个或多个语句块组成.
- ❑• 定义函数的语句通常只写一次, 这些语句对其它函数是不可见的



6.1 Introduction

❑ Function is invoked by a function call

① Provide function name and 实参arguments (data)

② Function performs its task

③ Function returns a result or simply returns control to the caller

Topics

- ❑ 6.1 Introduction
- ❑ **6.2 Program Components in C++**
- ❑ 6.3 Math Library Functions
- ❑ 6.4 Function Definitions with Multiple Parameters
- ❑ 6.5 Function Prototypes and Argument Coercion
- ❑ 6.6 C++ Standard Library Header Files
- ❑ 6.7 Case Study: Random Number Generation
- ❑ 6.8 Case Study: Game of Chance and Introducing enum
- ❑ 6.9 Storage Classes
- ❑ 6.10 Scope Rules

6.2 Program Components in C++

- C++ programs are typically written by combining
- ① new functions and classes
- ② with “prepackaged” functions and classes available in the C++ Standard Library.

Topics

- □ 6.1 Introduction
- □ 6.2 Program Components in C++
- □ **6.3 Math Library Functions**
- □ 6.4 Function Definitions with Multiple Parameters
- □ 6.5 Function Prototypes and Argument Coercion
- □ 6.6 C++ Standard Library Header Files
- □ 6.7 Case Study: Random Number Generation
- □ 6.8 Case Study: Game of Chance and Introducing enum
- □ 6.9 Storage Classes
- □ 6.10 Scope Rules

6.3 Math Library Functions

- **double ceil(double x);**
    - **rounds x to the smallest integer not less than x,** 不小于x的最小整数
    - **ceil(9.2) is 10.0, ceil(-9.8) is -9.0, ceil(2.0) is 2.0**
- **double floor(double x);**
    - **rounds x to the largest integer not greater than x,** 不大于x的最大整数
    - **floor(9.2) is 9.0, floor(-9.8) is -10.0, floor(2.0) is 2.0**

6.3 Math Library Functions

- □ double exp(double x);
  - ❖ exponential function ex, 指数函数
  - ❖ exp(1.0) is 2.71828, exp(2.0) is 7.38906

- □ double pow(double x, double y);
  - ❖ x raised to power y (x^y), 幂函数
  - ❖ pow(2, 7) is 128, pow(9, 0.5) is 3

6.3 Math Library Functions

- double fmod(double x, double y);
  - remainder of x/y as a floating-point number，x除以y的余数(%, 整数)
  - fmod(2.6, 1.2) is 0.2

- double sqrt(double x);
  - square root of x (where x is a nonnegative value)，非负数x的平方根
  - sqrt(9.0) is 3.0

6.3 Math Library Functions

- ❑ double log(double x);
  - ❖natural logarithm of x (base e), x的自然对数
  - ❖log(2.718282) is 1.0, log(7.389056) is 2.0

- ❑ double log10(double x);
  - ❖logarithm of x (base 10), x的以10为底的对数值
  - ❖log10(10.0) is 1.0, log10(100.0) is 2.0

6.3 Math Library Functions

- double sin(double x);
  - trigonometric sine of x (x in radians), 弧度x的正弦值
  - sin(0.0) is 0

- double cos(double x);
  - trigonometric cosine of x (x in radians), 弧度x的余弦值
  - cos(0.0) is 1.0

6.3 Math Library Functions

- ❑ double tan(double x);
  - ❖ trigonometric tangent of x (x in radians), x的正切值
  - ❖ tan(0.0) is 0

- ❑ double fabs(double x);
  - ❖ absolute value of x, x的绝对值
  - ❖ fabs(5.1) is 5.1, fabs(0.0) is 0.0, fabs(-8.76) is 8.76

Topics

- ❑ 6.1 Introduction
- ❑ 6.2 Program Components in C++
- ❑ 6.3 Math Library Functions
- ❑ **6.4 Function Definitions with Multiple Parameters**
- ❑ 6.5 Function Prototypes and Argument Coercion
- ❑ 6.6 C++ Standard Library Header Files
- ❑ 6.7 Case Study: Random Number Generation
- ❑ 6.8 Case Study: Game of Chance and Introducing enum
- ❑ 6.9 Storage Classes
- ❑ 6.10 Scope Rules

6.4 Function Definitions with Multiple Parameters

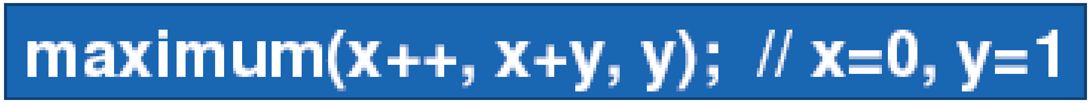
int maximum(int x, int y, int z) ;

- ❑ **1.** 函数定义：
- ❑ 形式参数列表, 每个参数都必须指定类型, 参数之间逗号分隔
- ❑ 接口声明时, 函数参数名可以不指定
- ❑ 函数体定义时, 应指定参数名

6.4 Function Definitions with Multiple Parameters

maximum (10, a, b*c);

- □ 2. 函数调用：
- □ 实际参数列表, 参数不能包含类型说明, 逗号隔开
- □ 实参可以是常量、变量或者表达式
- □ 非逗号表达式, 取值顺序因编译器而异



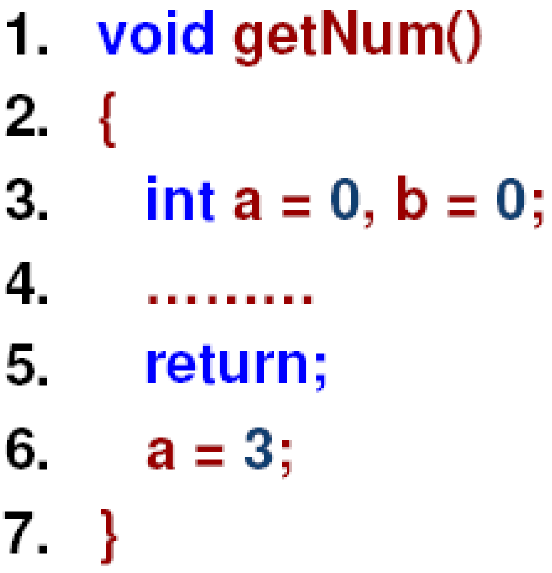
```
maximum(x++, x+y, y);  // x=0, y=1
```

NO！

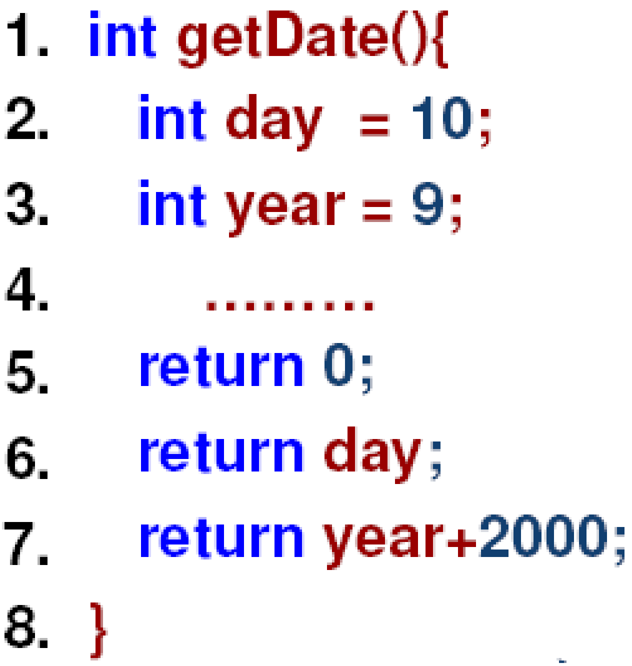
6.4 Function Definitions with Multiple Parameters

- ❑3.函数调用返回
- ❑如果没有返回数据, void
  - ❖则可以return; 直接返回
  - ❖没有return语句时, 至函数体右括号返回
- ❑如果有返回数据，则
  - ❖return expression; 返回
  - ❖expression可为常量、变量或者表达式

6.4 Function Definitions with Multiple Parameters



```
void getNum()
{
    int a = 0, b = 0;
    .........
    return;
    a = 3;
}
```



```
int getDate(){
    int day  = 10;
    int year = 9;
        .........
    return 0;
    return day;
    return year+2000;
}
```

Topics

- 6.1 Introduction
- 6.2 Program Components in C++
- 6.3 Math Library Functions
- 6.4 Function Definitions with Multiple Parameters
- **6.5 Function Prototypes and Argument Coercion**
- 6.6 C++ Standard Library Header Files
- 6.7 Case Study: Random Number Generation
- 6.8 Case Study: Game of Chance and Introducing enum
- 6.9 Storage Classes
- 6.10 Scope Rules

6.5 Function Prototypes and Argument Coercion

- ❑ 1.Function Prototypes(函数原型)

  作用: 告诉编译器和用户函数名、输入参数(个数、类型、顺序) 、返回类型

  返回类型 函数名(形式参数列表);

- ❑ 若无返回数据, 则void
- ❑ 函数名必须为标识符(identifier)
- ❑ 形参是需要被初始化的内部变量, 生命周期和可见性仅限于该函数内部
- ❑ 若无参数, 则()内为void或空

```
int maximum( int, int, int );
```

6.5 Function Prototypes and Argument Coercion

- ❑ 函数原型的应用
- ❑ 原则: 函数必须先声明后使用

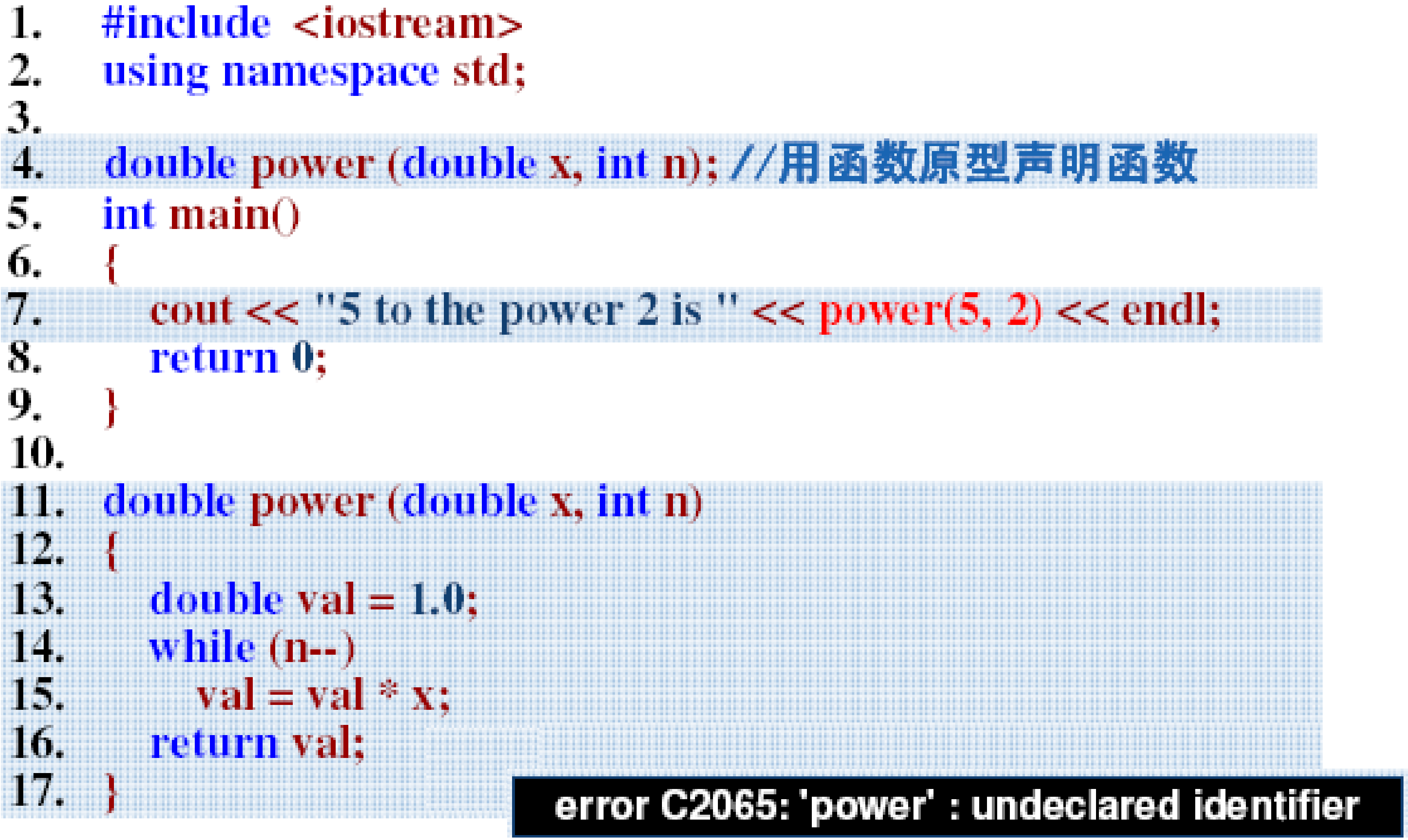
① 函数定义(实现代码)在前, 调用在后时, OK;

② 函数调用在前, 定义在后时?
必须在调用函数前给出函数原型

6.5 Function Prototypes and Argument Coercion

```
// 编写一个求x的n次方的函数
#include  <iostream>
using namespace std;

double power (double x, int n)
{
    double val = 1.0;
    while (n--)
        val = val*x;
    return val;
}
int main()
{
    cout << "5 to the power 2 is " << power(5, 2) << endl;
    return 0;
}
```

6.5 Function Prototypes and Argument Coercion



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

double power (double x, int n); //用函数原型声明函数
int main()
{
    cout << "5 to the power 2 is " << power(5, 2) << endl;
    return 0;
}

double power (double x, int n)
{
    double val = 1.0;
    while (n--)
        val = val * x;
    return val;
}
```

```
error C2065: 'power' : undeclared identifier
```

6.5 Function Prototypes and Argument Coercion

- □ **2.函数签名(function signature)**

  作用: 用于唯一确定所调用的函数!

- □ 函数原型中的函数名+参数类型
- □ 不包括函数返回类型
- □ 函数签名在同一个作用域内须唯一, 函数的作用域即程序中函数 “可见” (允许被访问)的范围

```
int printLargestNum (int num1, int num2);
void printLargestNum (int num1, int num2);
int main()
{
   printLargestNum (23, 76);
   return 0;
}
```

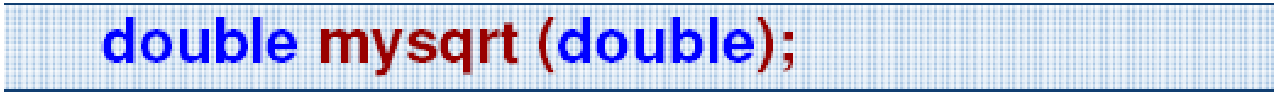
```
// GradeBook.h
#include <string>
using std::string;

// GradeBook class definition
class GradeBook {
public:
    GradeBook( string );
    void setCourseName( string );
    bool setCourseName( string );
    string getCourseName();
    void displayMessage();
    void determineClassAverage();
private:
    string courseName;
}; // end class GradeBook
```

error C2371: 'setCourseName' : redefinition; different basic types

6.5 Function Prototypes and Argument Coercion

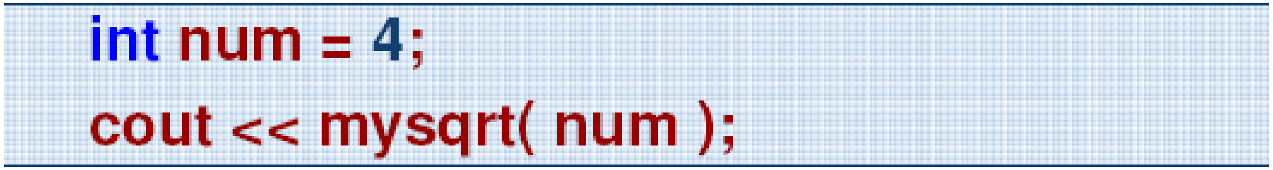
❑3.Argument Coercion(实参的强制类型转化)

函数原型:



```
double mysqrt (double);
```

函数调用:



```
int num = 4;
cout << mysqrt( num );
```

结果：能够正确的求值

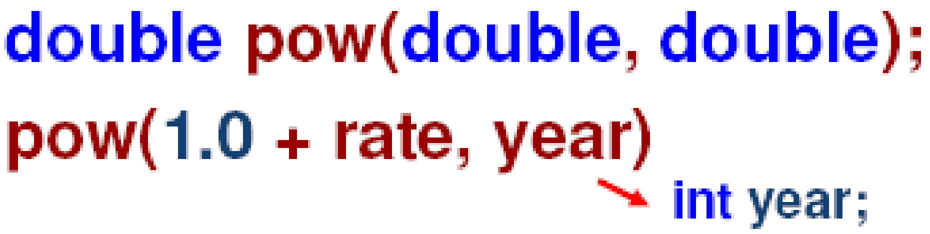
6.5 Function Prototypes and Argument Coercion

- ❑ 3-1 提升规则(promotion rule)- 隐性转化

  在不丢失数据的前提下, 将一种类型转换为另一种类型

① 函数的实参

- ❑ 函数实参类型不符合函数定义中指定的参数类型时, 自动将其转换为正确类型之后再进行函数调用



```
double pow(double, double);
pow(1.0 + rate, year)
                     ↘ int year;
```

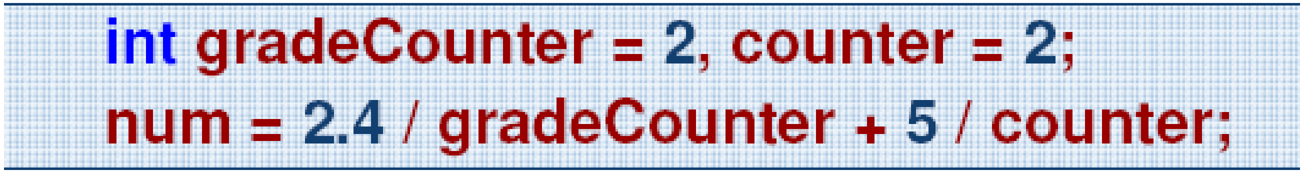
6.5 Function Prototypes and Argument Coercion

- ❑ **3-1 提升规则(promotion rule)- 隐性转化**

  在不丢失数据的前提下, 将一种类型转换为另一种类型

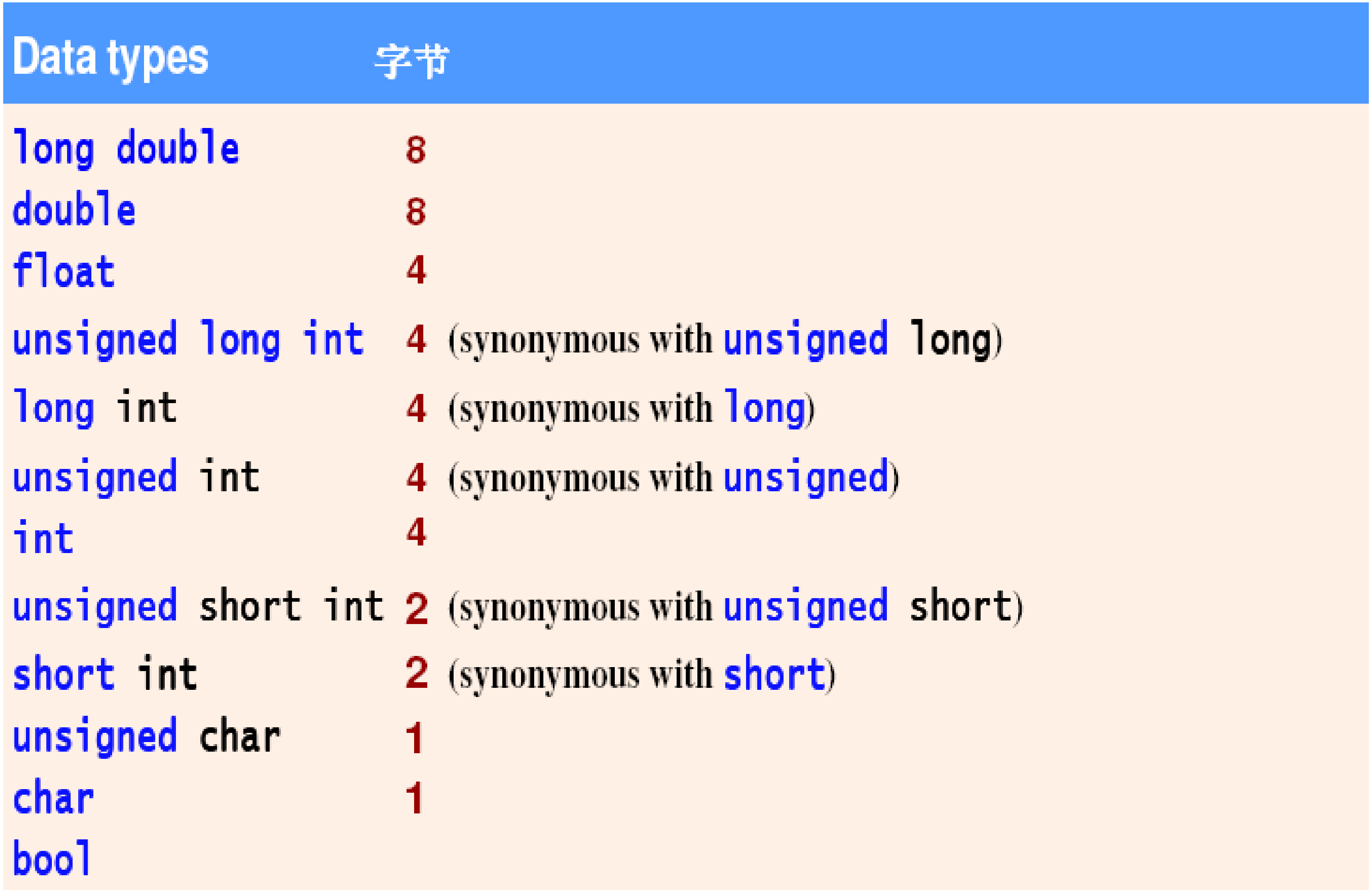
② **混合类型表达式(mixed-type expression)**

- ❑ 表达式中每个值的类型提升为所在子表达式中最高的类型
- ❑ 生成每个值的临时值并在表达式中使用, 原值保持不变



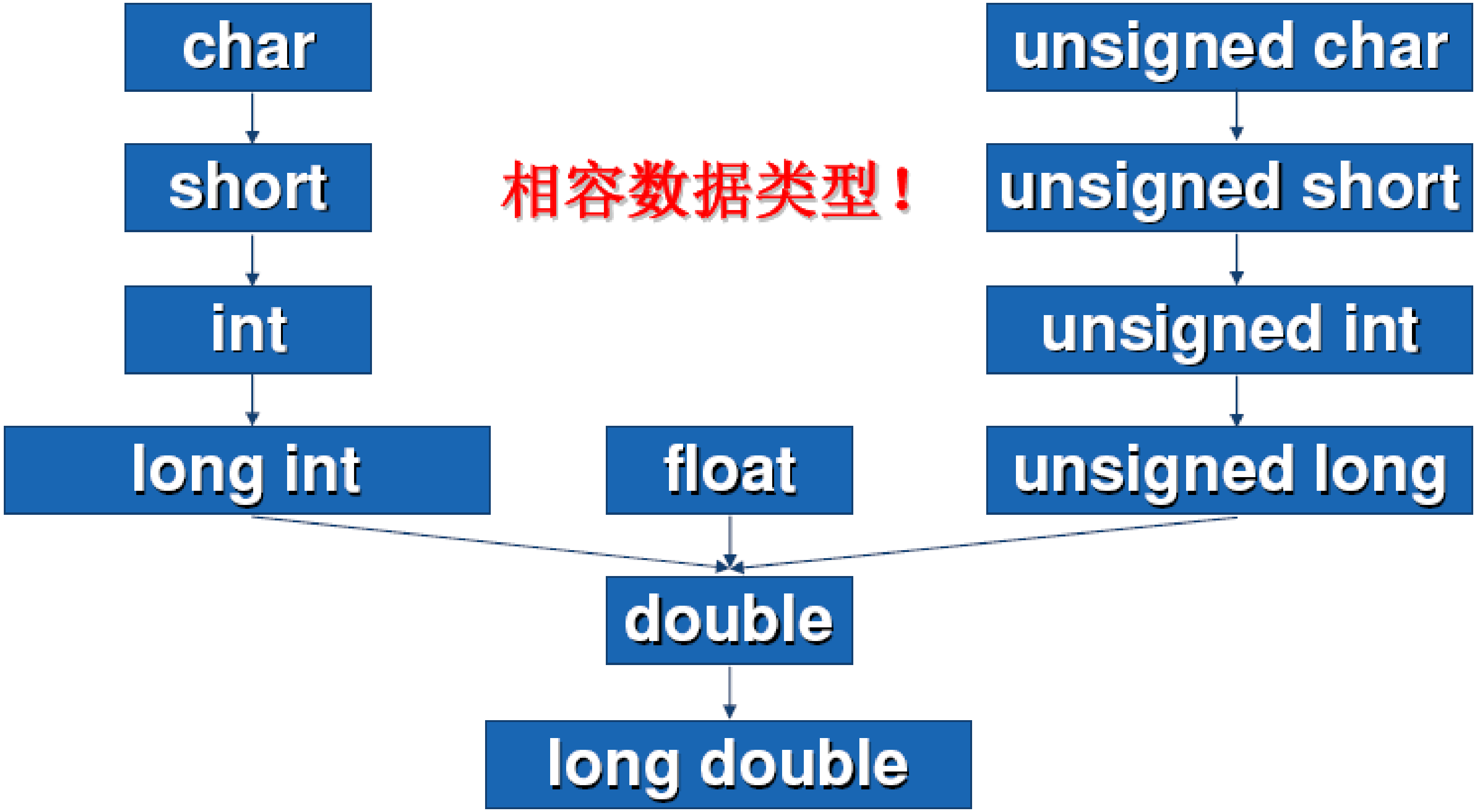
```
int gradeCounter = 2, counter = 2;
num = 2.4 / gradeCounter + 5 / counter;
```

基本数据类型的"高-低"顺序



Data types	字节	
long double	8	
double	8	
float	4	
unsigned long int	4	(synonymous with unsigned long)
long int	4	(synonymous with long)
unsigned int	4	(synonymous with unsigned)
int	4	
unsigned short int	2	(synonymous with unsigned short)
short int	2	(synonymous with short)
unsigned char	1	
char	1	
bool		

基本数据类型的“高-低”顺序



相容数据类型！

6.5 Function Prototypes and Argument Coercion

- 3-2 显式转化

  若将“高级别”基本数据类型数据转化为“低级别”数据, 则会造成数据丢失, 必须进行显示转化.

- ① 赋值

  1. int num = 4.3;

  2. int square (int num);

     square(4.3);

6.5 Function Prototypes and Argument Coercion

❑3-2 显式转化

若将“高级别”基本数据类型数据转化为“低级别”数据, 则会造成数据丢失, 必须进行显示转化.

② 类型转化操作符

```
double dnum = 4.3;
int num = static_cast<int>(dnum);
```

Topics

- □ 6.1 Introduction
- □ 6.2 Program Components in C++
- □ 6.3 Math Library Functions
- □ 6.4 Function Definitions with Multiple Parameters
- □ 6.5 Function Prototypes and Argument Coercion
- □ **6.6 C++ Standard Library Header Files**
- □ 6.7 Case Study: Random Number Generation
- □ 6.8 Case Study: Game of Chance and Introducing enum
- □ 6.9 Storage Classes
- □ 6.10 Scope Rules

6.6 C++ Standard Library Header Files

C++头文件：

① 标准C语言库函数的头文件, 带有.h后缀：

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

② 标准C++语言类库的头文件, 不带.h后缀：

```
#include <iostream>
```

③ 由标准C语言库函数头文件转换获得的标准C++的头文件, 把原有标准C语言库函数头文件去掉.h后缀而加上c前缀。

```
#include <cstring>
```

6.6 C++ Standard Library Header Files

- □ 每个标准库都有对应的头文件, 包含库中所有抽象类型接口定义、函数原型, 以及这些函数所需各种常量的定义

- □ C++标准库的内容在50个标准头文件中定义, 其中18个提供了C库的功能。

输入/输出类头文件

- ❑ <iostream> 支持标准流cin/cout/cerr/clog的输入和输出
- ❑ <iomanip> 提供流操作算子, 允许改变流的状态,从而改变输出的格式
- ❑<fstream> 支持文件的流输入输出
- ❑<cmath> C数学库, 并附加了重载函数以支持C++约定
- ❑ <string> 为字符串类型提供支持和定义, 包括单字节字符串(由char组成)和多字节字符串
- ❑<cstring> C类型字符串的处理函数
- ❑<ctime> 支持系统时钟函数

Topics

- 6.1 Introduction
- 6.2 Program Components in C++
- 6.3 Math Library Functions
- 6.4 Function Definitions with Multiple Parameters
- 6.5 Function Prototypes and Argument Coercion
- 6.6 C++ Standard Library Header Files
- **6.7 Case Study: Random Number Generation**
- 6.8 Case Study: Game of Chance and Introducing enum
- 6.9 Storage Classes
- 6.10 Scope Rules

6.7 Case Study: Random Number Generation

- ❑ 1.随机数的产生

  int rand();

- ❑ #include <cstdlib>
- ❑ 返回伪随机数(整型), 0 ~ RAND_MAX
- ❑ RAND_MAX, symbolic constant(符号常量)

  #define RAND_MAX 0x7fff   // 32767

6.7 Case Study: Random Number Generation

❑ 2.随机数种子的设置

```
void srand( unsigned int seed );
```

❑ 通过设置“种子”(SEED)来产生不同的随机数的序列, 程序中调用一次即可

❑ 只要种子不同, 那么通过rand()得到的随机数序列就不同; 反之, 如果种子一样, 那么通过rand()得到的随机数就是相同的

❑ unsigned int可简写为unsigned

6.7 Case Study: Random Number Generation

- 3.种子的选择

```
time_t  time( time_t *timer );
```

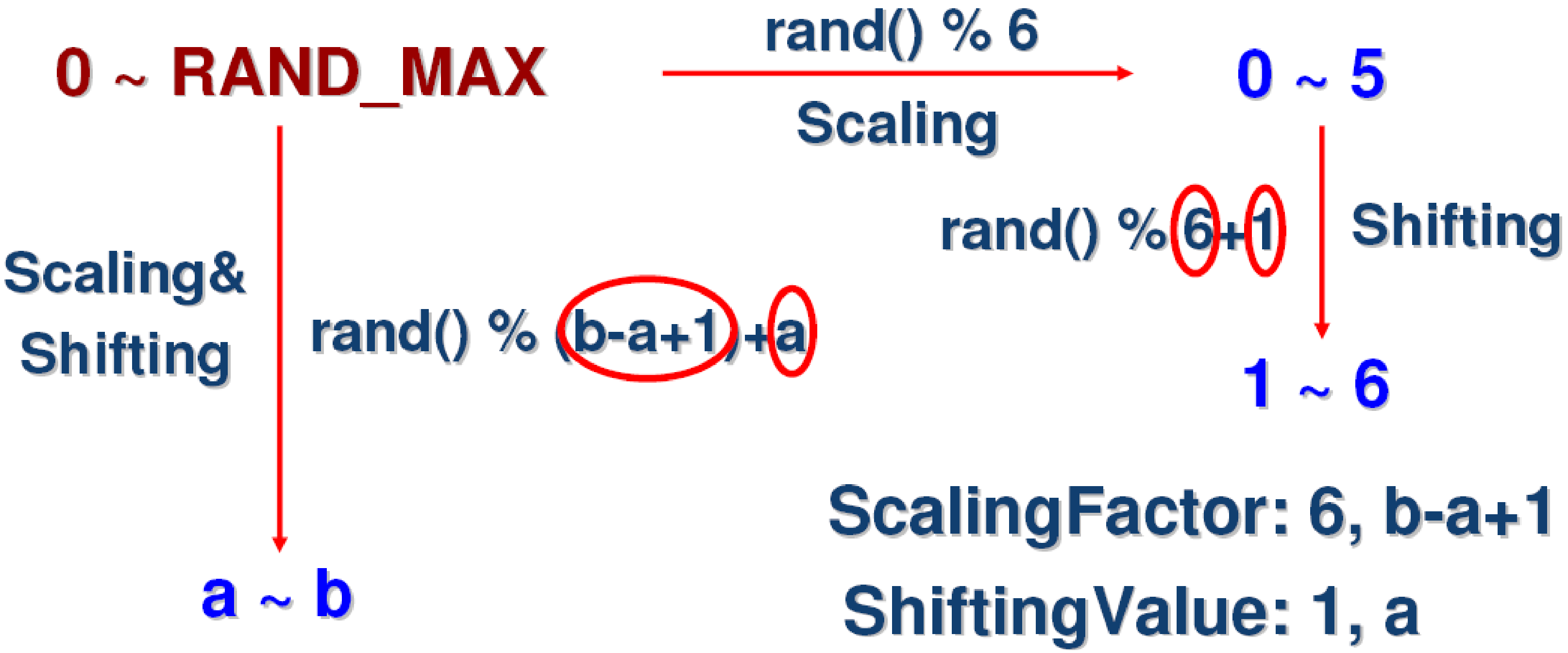
- #include <ctime>
- 返回当前时间, 自1970年1月1日0点至今过去的秒数, 其中time_t即long数据类型
- timer设置为NULL(==0)即可
- 即: srand (time (0));

6.7 Case Study: Random Number Generation

4.如何生成合适大小的随机数

❑ Scaling, 按比例缩放, 等概率

❑ Shifting, 平移



0 ~ RAND_MAX

rand() % 6

Scaling

0 ~ 5

rand() % 6+1

Shifting

1 ~ 6

Scaling&
Shifting

rand() % (b-a+1)+a

a ~ b

ScalingFactor: 6, b-a+1

ShiftingValue: 1, a

6.7 Case Study: Random Number Generation

❑ 5.结论

// 设置随机SEED

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

// 取[a, b]之间的随机数

```
int number = a + rand() % (b – a + 1);
```

程序解读（P158）

Q & A

□1. 简述函数签名的概念和意义.

```
void debit( int value );
int debit( int value );
```

□2. 如何产生随机数(a, b).

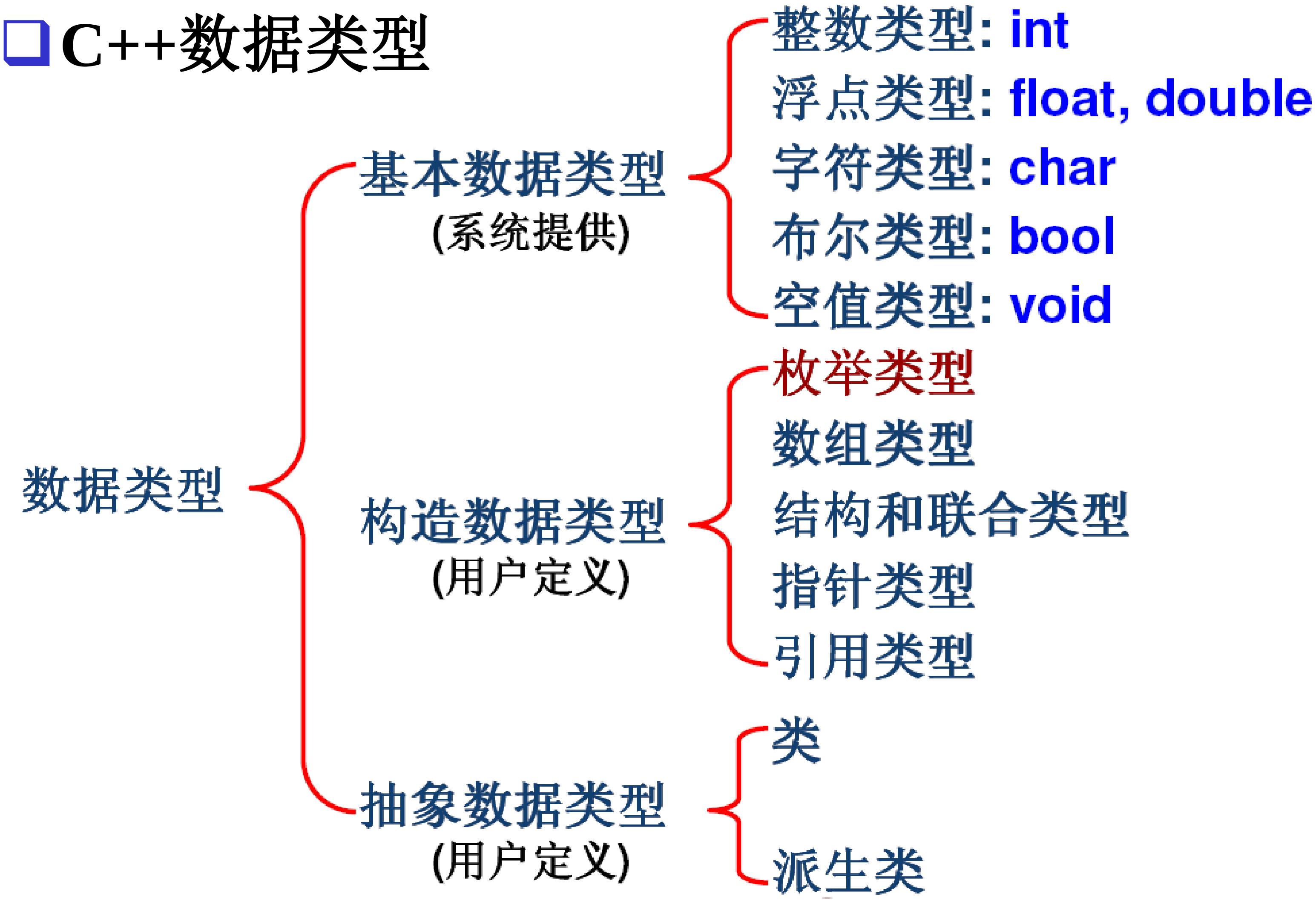
```
// <ctime> <cstdlib>
srand(time(0));
int n = rand() % (b-a-1) + a + 1;
```

Topics

- 6.1 Introduction
- 6.2 Program Components in C++
- 6.3 Math Library Functions
- 6.4 Function Definitions with Multiple Parameters
- 6.5 Function Prototypes and Argument Coercion
- 6.6 C++ Standard Library Header Files
- 6.7 Case Study: Random Number Generation
- **6.8 Case Study: Game of Chance and Introducing enum**
- 6.9 Storage Classes
- 6.10 Scope Rules

6.8 Case Study: Game of Chance and Introducing enum

- C++数据类型



- 数据类型
  - 基本数据类型 (系统提供)
    - 整数类型: int
    - 浮点类型: float, double
    - 字符类型: char
    - 布尔类型: bool
    - 空值类型: void
  - 构造数据类型 (用户定义)
    - 枚举类型
    - 数组类型
    - 结构和联合类型
    - 指针类型
    - 引用类型
  - 抽象数据类型 (用户定义)
    - 类
    - 派生类

6.8 Case Study: Game of Chance and Introducing enum

- 如何表示周一、周二…… 周日？(1-7)
- 如何表示一月、二月…… 十二月？(1-12)
- 如何表示RGB三原色, 即红、绿、蓝？(1-3)

- 不直观, 易出错!

  int day = 1表示周日? 周一?

  int day = 8? month=13?

6.8 Case Study: Game of Chance and Introducing enum

- ❑ **Enumeration type(枚举数据类型): 用户自定义的构造数据类型, 其值集由用户程序定义**

  enum 枚举类型名{枚举值表};

- ❑ 例如:

```
enum Day { SUN, MON, TUE, WED, THU, FRI,
               SAT };
enum Month { JAN, FEB, MAR, APR, MAY,
                 JUN, JUL, AUG, SEP, OCT, NOV,
                 DEC};
enum Color { RED, GREEN, BLUE };
```

6.8 Case Study: Game of Chance and Introducing enum

- **enum Day { SUN, MON, TUE, WED, THU, FRI, SAT };**

- 每个枚举值都应是标识符, 对应一个整数值,通常第一个枚举值对应常量值0, 第二个对应1, 依次递增类推

- **SUN为0, MON为1, TUE为2, WED为3, THU为4, FRI为5, SAT为6**

6.8 Case Study: Game of Chance and Introducing enum

- 在定义枚举类型时, 也可以指定枚举值对应的常量, 后续值依次递增至结束或者至下一个指定:
- **enum Day { SUN=7, MON=1, TUE, WED, THU, FRI, SAT };**
- **SUN**为**7, MON**为**1, TUE**为**2, WED**为**3, THU**为**4, FRI**为**5, SAT**为**6**
- 注意: 枚举值表中枚举值不能同名, 但是可以对应相同的整数常量值

6.8 Case Study: Game of Chance and Introducing enum

- enum Test { AA=3, BB, CC=3, DD=1, AA };

  error C2086: 'AA' : redefinition

- enum Test { AA=3, BB, CC=3, DD=1, EE };
- AA=3, BB=4, CC=3, DD=1, EE=2

6.8 Case Study: Game of Chance and Introducing enum

enum Test { AA=3, BB, CC=3, DD=1, EE };

❑ 两种变量定义方式

① enum Test tt;

② Test tt;

❑ 变量赋值

① tt = AA; // OK

② tt = 3; // ERROR, 用户构造/基本类型非相容类型

③ tt = static_cast<Test>(5); // OK, 但无意义！

6.8 Case Study: Game of Chance and Introducing enum

enum Test { AA=3, BB, CC=3, DD=1, EE };

□ 可根据对应的整数值进行关系运算 **(==, >, <, >=, <=)**

□ **switch判断, 比对**

1. enum Test tt;

2. tt = AA;

3. cout <<boolalpha << (tt == CC) << endl;

4. cout <<boolalpha << (tt == 3) << endl;

6.8 Case Study: Game of Chance and Introducing enum

Craps双色子赌博游戏

- ❑1. 第一把

① 掷出点数为7或11, 则玩家WON;

② 掷出点数为2、3或12, 则玩家LOST;

③ 否则记下该点数(Point), 进入2;

- ❑2. 继续掷色子, CONTINUE

① 掷出7, 则玩家LOST;

② 掷出点数==Point, 则玩家WON;

③ 否则重复2., CONTINUE

6.8 Case Study: Game of Chance and Introducing enum

Craps双色子赌博游戏

1. 掷色子: 1 – 6点

随机数生成srand() / rand()

2. 根据两个色子点数和(2–12), 判断游戏者状态

选择语句, 多选switch

3. 游戏状态WON/LOST/CONTINUE

枚举类型enum

程序解读（P161）

```
// Craps simulation.
#include <iostream>
using std::cout;
using std::endl;

#include <cstdlib> // contains prototypes for functions srand and rand
using std::rand;
using std::srand;

#include <ctime> // contains prototype for function time
using std::time;

int rollDice(); // rolls dice, calculates amd displays sum

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
```

```
   // determine game status and point (if needed) based on first roll
   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 game is not complete
   while ( gameStatus == CONTINUE ) // not WON or LOST
   {
      sumOfDice = rollDice(); // roll dice again

      // determine game status
      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

// roll dice, calculate sum and display results
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
```

```
Player rolled 2 + 5 = 7
Player wins
```

```
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

Topics

- ❑ 6.1 Introduction
- ❑ 6.2 Program Components in C++
- ❑ 6.3 Math Library Functions
- ❑ 6.4 Function Definitions with Multiple Parameters
- ❑ 6.5 Function Prototypes and Argument Coercion
- ❑ 6.6 C++ Standard Library Header Files
- ❑ 6.7 Case Study: Random Number Generation
- ❑ 6.8 Case Study: Game of Chance and Introducing enum
- ❑ **6.9 Storage Classes**
- ❑ 6.10 Scope Rules

6.9 Storage Classes

(1)标识符(Identifier)

- 用于变量、类型、函数与标签等的命名
- 由字母(a ~ z或A ~ Z) 、数字(0 ~ 9) 、下划线(_)组成
- 必须由字母或下划线开头
- 大小写敏感

6.9 Storage Classes

(2)变量(Variable)

- ❑ 在程序运行过程中从程序外部获得或在程序运行过程中通过计算产生的、其值可以变化的数据
- ❑ 在内存中占用一个逻辑存储空间
- ❑ 变量的基本属性：

① 名字Name

② 类型Type

③ 值Value

④ 内存空间(地址) Size

6.9 Storage Classes

(3)常量(Constant)

□ 在程序执行过程中保持不变的数据

□ 分为直接常量和符号常量两类:

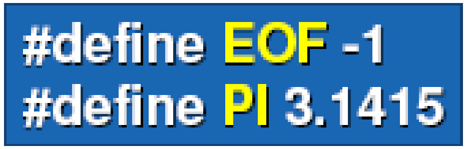
① 直接常量

.65, ‘a’, ‘\\’, “a”, “This is a string”

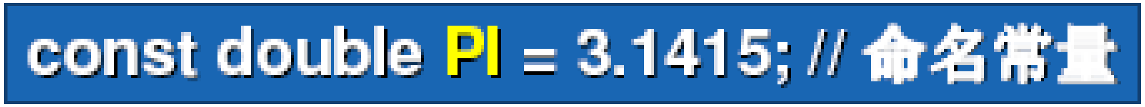
② 符号常量

#define <符号> <值>

const <类型名> <常量名>=<值>;

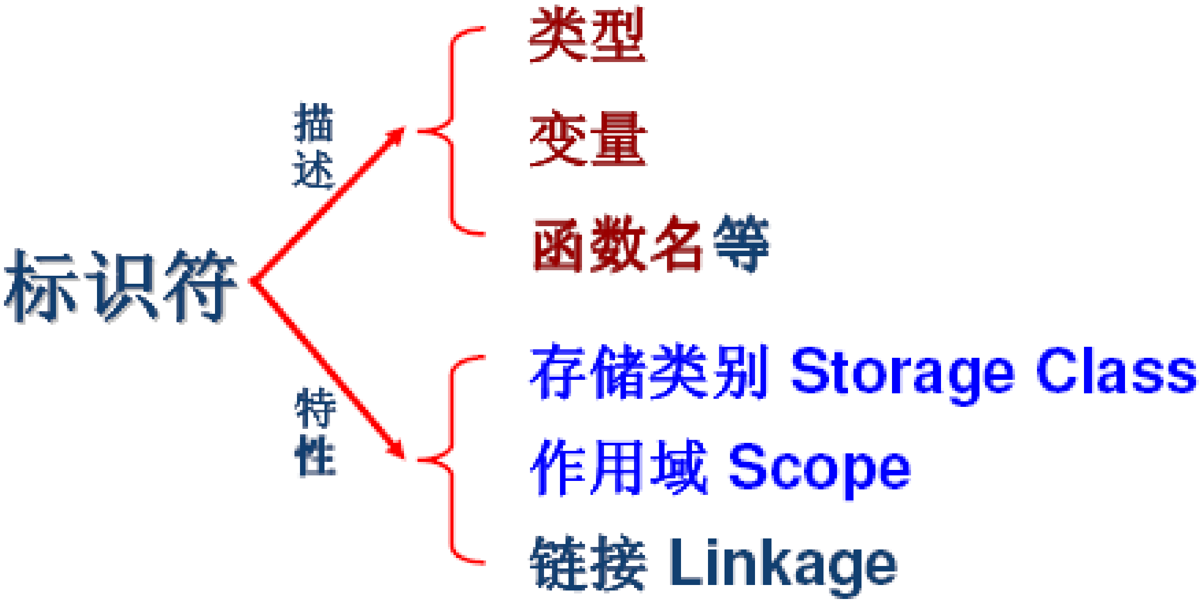


```
#define EOF -1
#define PI 3.1415
```



```
const double PI = 3.1415; // 命名常量
```

6.9 Storage Classes



标识符

描述

类型

变量

函数名等

特性

存储类别 Storage Class

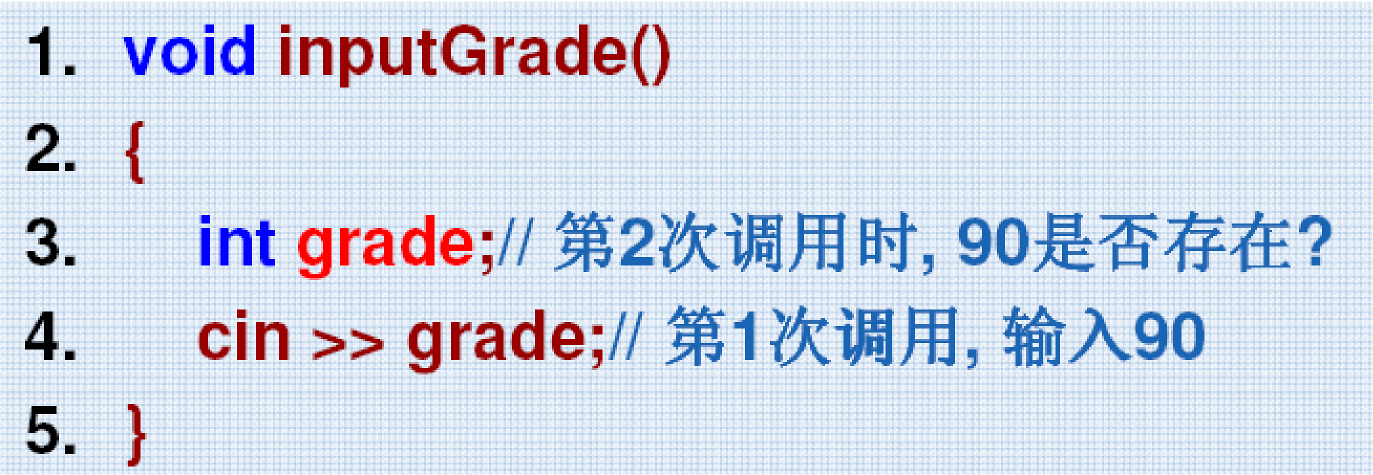
作用域 Scope

链接 Linkage

6.9 Storage Classes

- 存储类别(Storage Class)

  决定标识符在内存中的生存期: 有些标识符的存在时间很短, 有些则重复生成和删除, 有些存在于整个程序的执行期间



```
void inputGrade()
{
    int grade;// 第2次调用时, 90是否存在?
    cin >> grade;// 第1次调用, 输入90
}
```

6.9 Storage Classes

两种存储类别:

- ❑ **(1) Automatic Storage Class**

  (自动存储类别, 变量)

  auto, register说明符(specifier)

- ❑ **(2) Static Storage Class**

  (静态存储类别, 变量和函数)

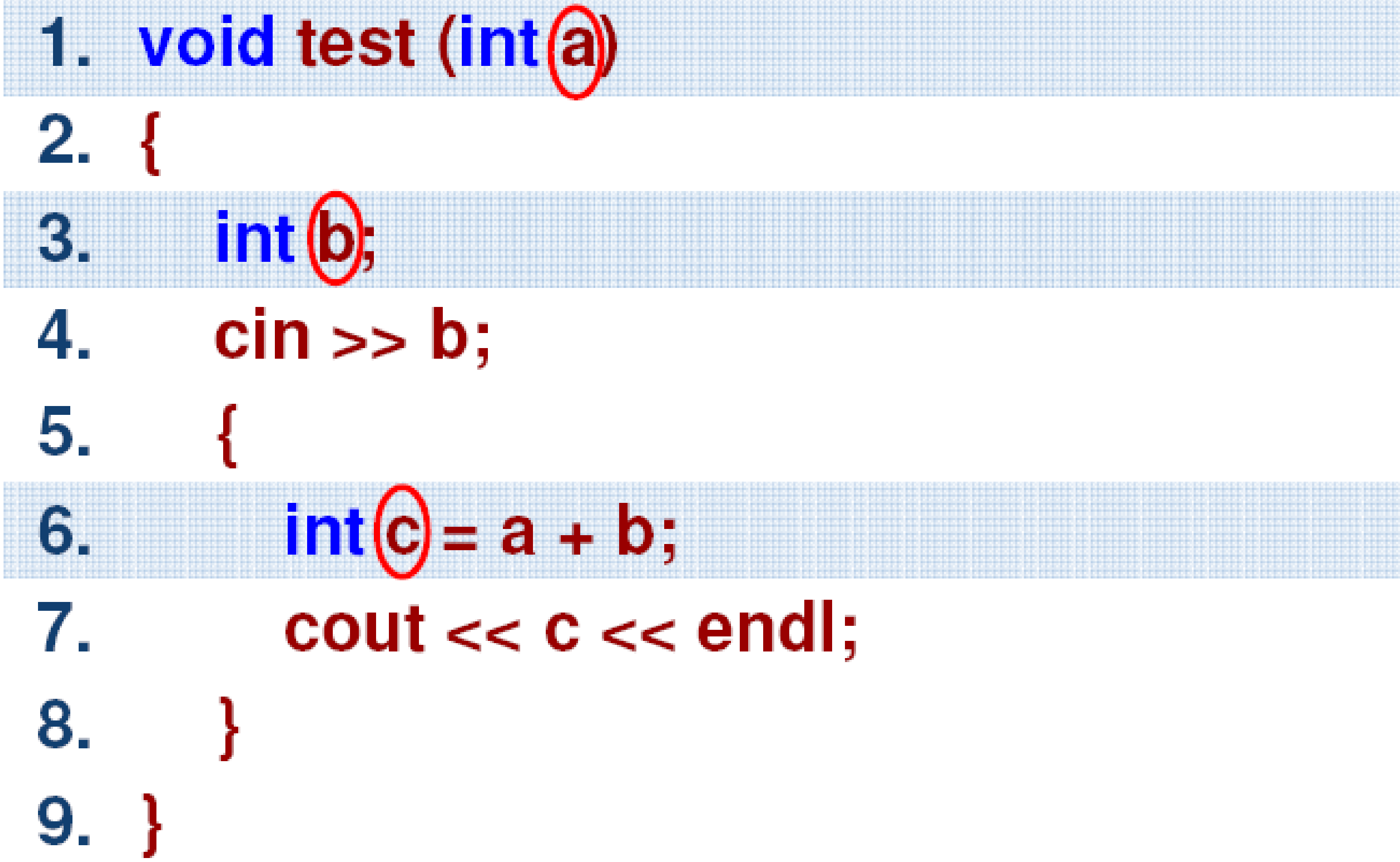
  static, extern说明符

6.9 Storage Classes

(1)自动存储类别(Automatic Storage Class)

- ❑ 在进入程序块执行时生成, 退出时销毁(释放内存)
- ❑ 函数的局部变量通常是自动存储类别, 包括函数内部定义的局部变量和形参, 简称自动变量

6.9 Storage Classes



```
void test (int a)
{
    int b;
    cin >> b;
    {
        int c = a + b;
        cout << c << endl;
    }
}
```

6.9 Storage Classes

- 说明符1: auto
- ① auto double x, y;

  显式声明x, y是自动存储类变量
- ② 函数局部变量和形参默认为auto变量, 关键词auto很少使用

6.9 Storage Classes

- 说明符2: register
- ① 自动变量用register修饰时, 建议编译将变量存入寄存器中进行运算, 称为寄存器变量(register variables)

  ```
  register int counter = 1;
  ```

- ② register声明通常是不需要的. 具有优化功能的编译器通常能自动识别经常使用的变量, 并决定将其放在寄存器中

6.9 Storage Classes

(2)静态存储类别(static storage class)

- □生存期为整个程序运行期间, 可适用于变量和函数

- □ 两种静态存储类别的标识符

  ① **External identifier:** 全局变量名和全局函数名

  ② **static**修饰的局部变量名

6.9 Storage Classes

□ 说明符1: static

① 用于修饰函数中的局部变量时*

- ❖该变量属于静态存储类别
- ❖仅在变量声明时进行一次初始化
- ❖函数结束时, 保留变量的数值; 在下次调用时,依然可以使用上次函数退出时的值
- ❖数值变量默认初始化为0

② 还可以修饰全局变量和全局函数以及类成员函数和成员变量

6.9 Storage Classes

- □ 说明符2:

  用于修饰全局函数/全局变量

- □ static和extern修饰全局函数和全局变量时,涉及多源文件程序中的Linkage等问题

6.9 Storage Classes

```
void test ()
{
    static int n = 10;
    cout << n++ << " ";
}
```

10 11 12

```
void test ()
{
    static int n;
    cout << n++ << " ";
}
```

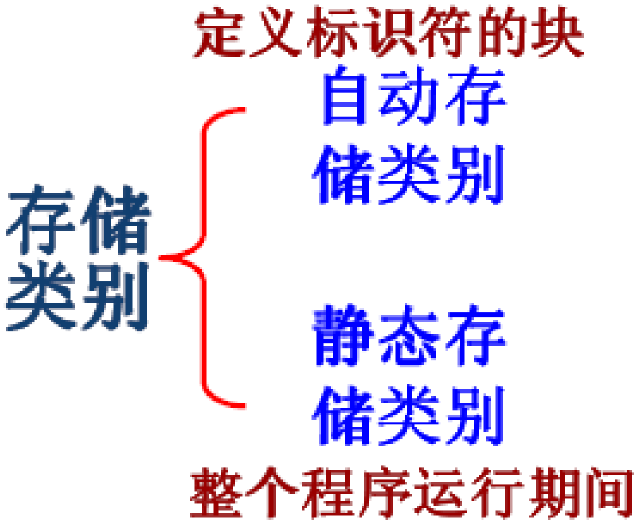
0 1 2

```
void test ()
{
    static int n;
    n = 10;
    cout << n++ << " ";
}
```

10 10 10

```
int main()
{
    test(); test(); test();
    return 0;
}
```

6.9 Storage Classes



定义标识符的块

存储类别

自动存储类别

静态存储类别

整个程序运行期间

- 函数局部变量和形参为自动存储类别
- 但是static修饰的局部变量为静态存储类别
- 全局函数和全局变量为静态存储类别

注意:一个标识符不能使用多个存储类说明符

```
register auto int x = 7; // ERROR
```

Topics

- 6.1 Introduction
- 6.2 Program Components in C++
- 6.3 Math Library Functions
- 6.4 Function Definitions with Multiple Parameters
- 6.5 Function Prototypes and Argument Coercion
- 6.6 C++ Standard Library Header Files
- 6.7 Case Study: Random Number Generation
- 6.8 Case Study: Game of Chance and Introducing enum
- 6.9 Storage Classes
- **6.10 Scope Rules**

6.10 Scope Rules

- scope(作用域):
- 程序中一个标识符可以被使用的范围

(1)函数作用域(function scope)

- 整个函数, 可以在整个函数体的任何地方使用该标识符
- 标签(label)是唯一具有函数作用域的标识符

6.10 Scope Rules

```
void func( int x )
{
    int a = 2, b = 5;
    goto start;
    b = 3;
start: a = b;
    cout << a << " " << b << endl;
}
```

```
5  5
```

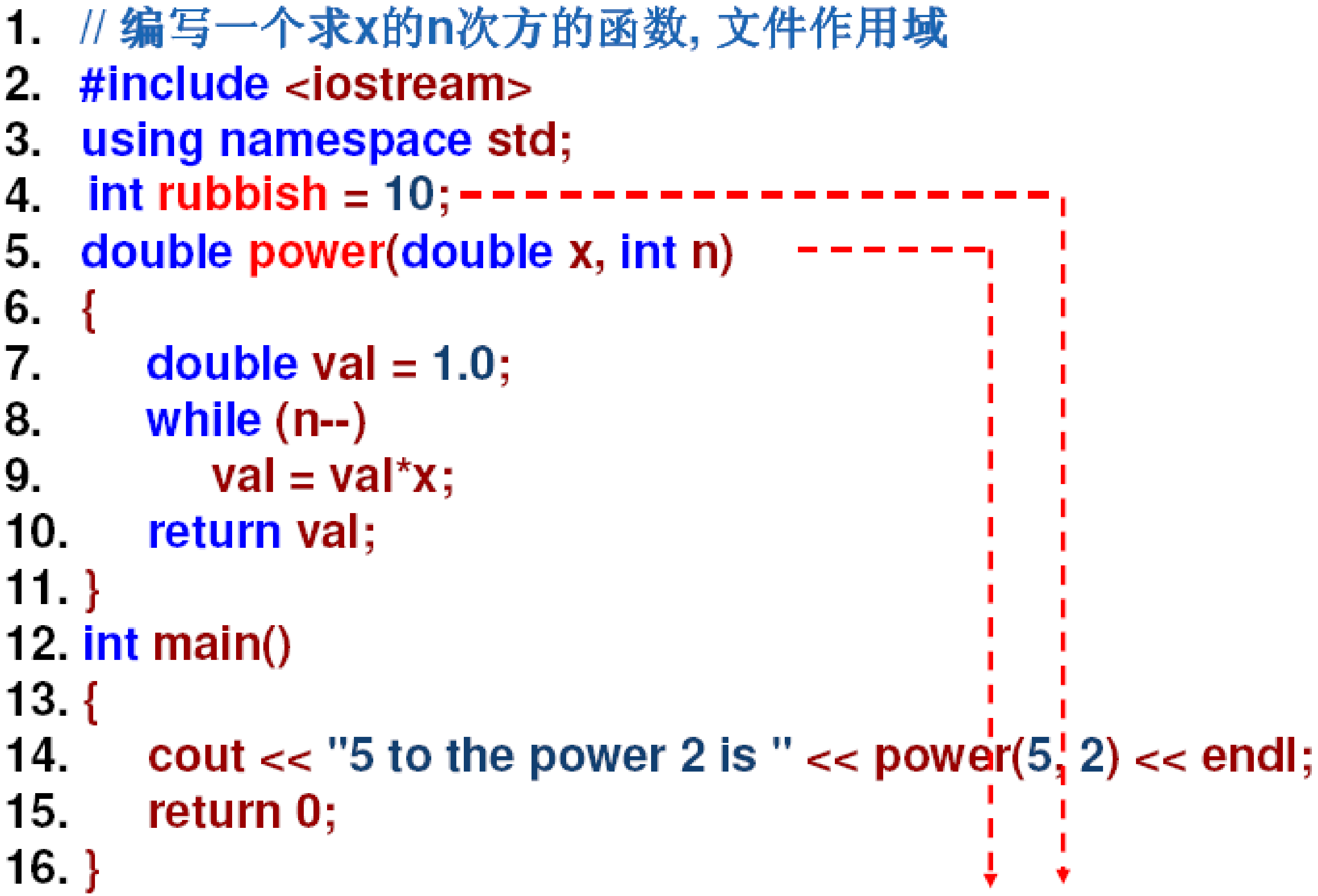
6.10 Scope Rules

(2)文件作用域(file scope)

- 声明→文件结束: 从该标识符声明处起到文件末尾的任何函数中都可以访问

- 在任何函数和类外声明的标识符, 如全局变量名、全局函数名

6.10 Scope Rules



```
// 编写一个求x的n次方的函数, 文件作用域
#include <iostream>
using namespace std;
 int rubbish = 10;
double power(double x, int n)
{
     double val = 1.0;
     while (n--)
          val = val*x;
     return val;
}
int main()
{
     cout << "5 to the power 2 is " << power(5, 2) << endl;
     return 0;
}
```

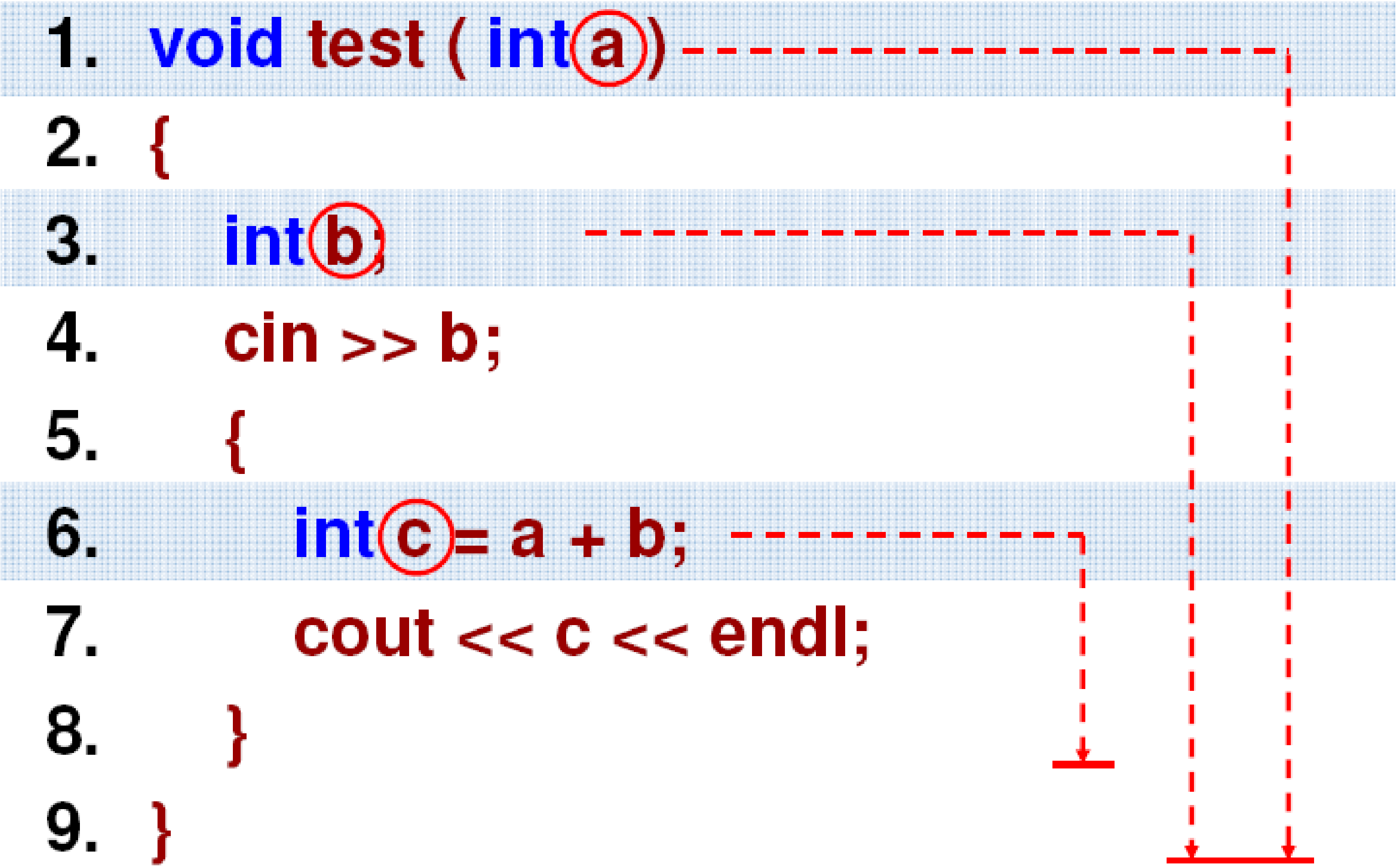
6.10 Scope Rules

(3)块作用域(block scope)

- ❑ 声明→块尾: 从标识符声明开始, 到所在块结束的右花括号处结束, 也称局部作用域(Local Scope)

- ❑ 块中声明的标识符

6.10 Scope Rules



```
void test ( int a)
{
    int b;
    cin >> b;
    {
        int c = a + b;
        cout << c << endl;
    }
}
```

6.10 Scope Rules

(4)函数原型作用域(function-prototype scope)

- ❑参数列表()中, 函数原型中使用的标识符可以在程序中的其它地方复用, 不会产生歧义
- ❑函数原型中的形参名是唯一具有该作用域的标识符

```
void useLocal( int a, int b ); // function prototype
void max( int a, int b ); // function prototype
```

6.10 Scope Rules

- (5)类作用域Class Scope, Ch9
- (6)名空间作用域Namespace Scope, Ch24

Topics

- **6.11 Function Call Stack and Activation Records**
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.11 Function Call Stack and Activation Records

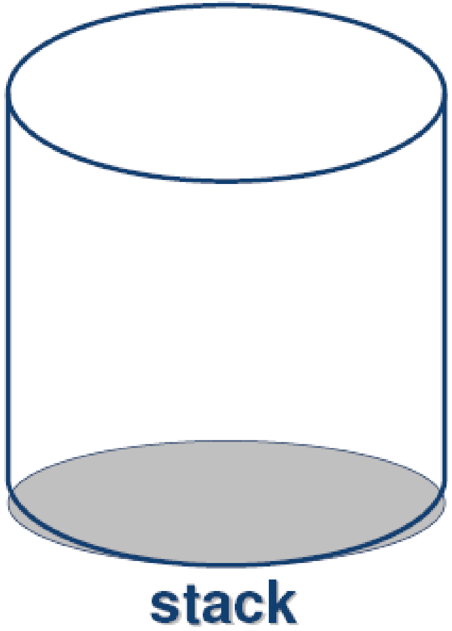
程序占用内存区域的分布:

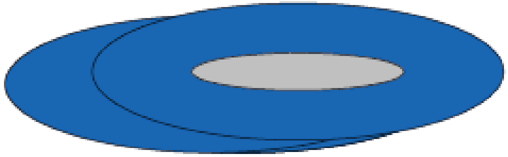
- ❑ 代码区(code area)
  - ❖ 被编译程序的执行代码部分
- ❑ 全局数据区(data area)
  - ❖ 常量、静态全局量和静态局部量等
- ❑ 堆区(heap)
  - ❖ 动态分配的内存, new / delete
- ❑ 栈区(stack)
  - ❖ 函数数据区, 函数形参和局部变量等自动变量所使用的内存区域

6.11 Function Call Stack and Activation Records

□栈: 后进先出(LIFO: last-in, first-out)

- 访问(top)
- 进栈(push)
- 出栈(pop)





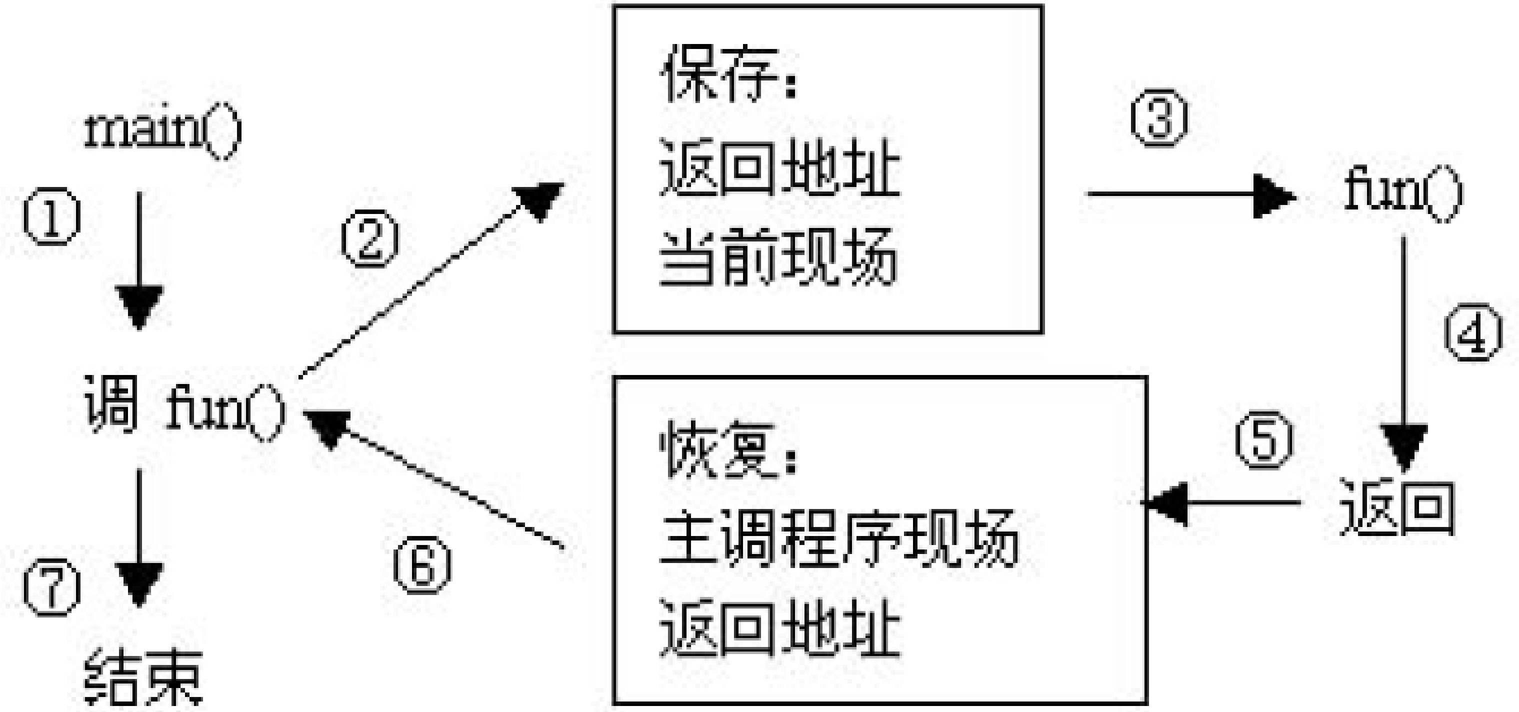
6.11 Function Call Stack and Activation Records

```
// 测试阶乘计算函数的主程序
#include <iostream>
using namespace std;
double fun(int);
int main()
{
    int n;
    cin >> n;
    cout << n << "! = " << fun(n) << endl;
    return 0;
}
```

6.11 Function C Activation

❑函数调用和返回的过程

```
// 测试阶乘计算函数的主程序
#include <iostream>
using namespace std;
double fun(int);
int main()
{
    int n;
    cin >> n;
    cout << n << "! = " << fun(n) << endl;
    return 0;
}
```



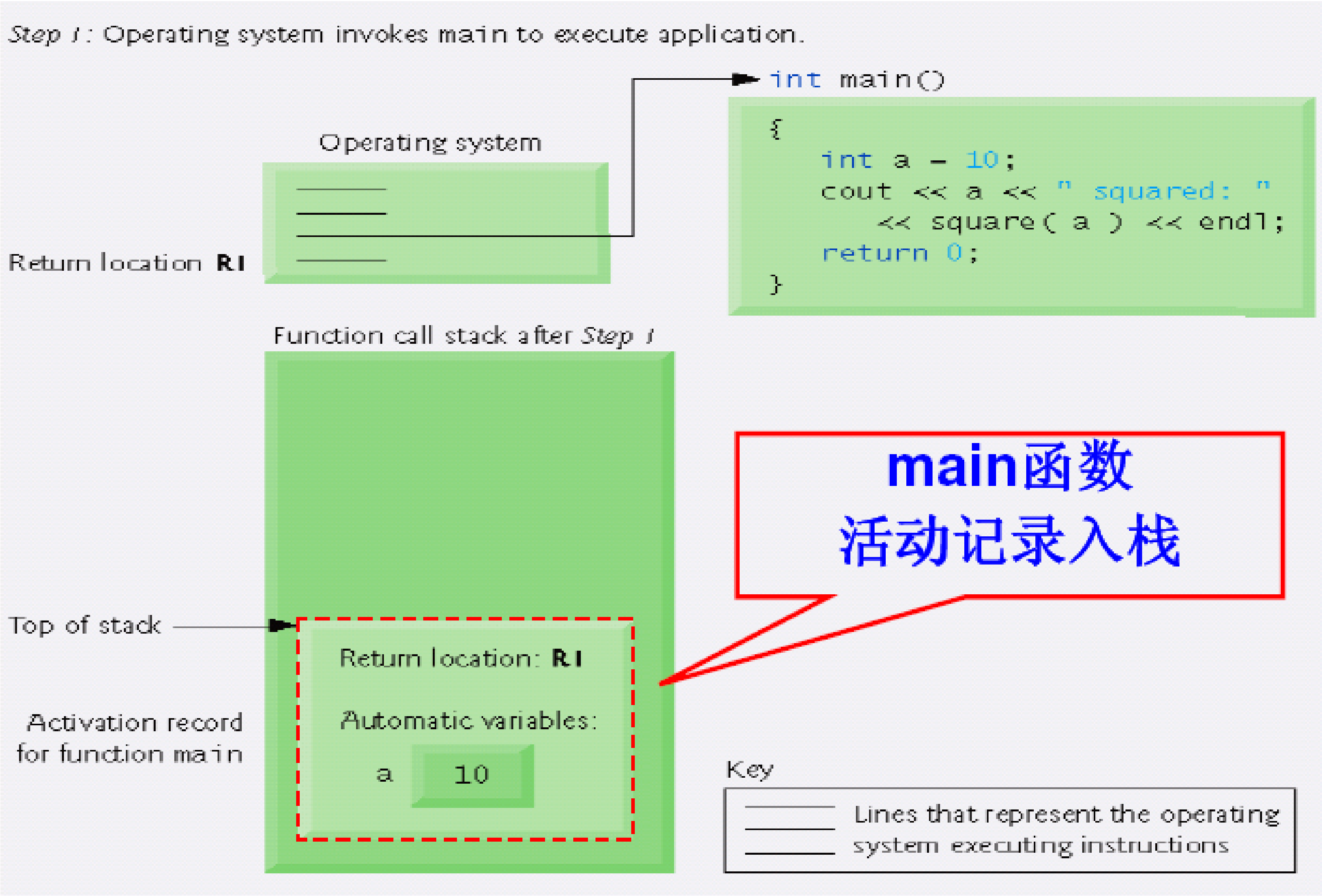
6.11 Function Call Stack and Activation Records

- 利用栈保存函数的活动记录

  当前被调用函数的活动记录总在栈顶

- 活动记录
  - 调用函数的返回地址(main)
  - 被调用函数(fun)的自动变量(函数形参和局部变量)
- 活动记录的作用:
  - 谁调用了我, 即函数执行完时返回哪儿
  - 如何维护本地的自动变量数据

Step 1：OS调用main函数



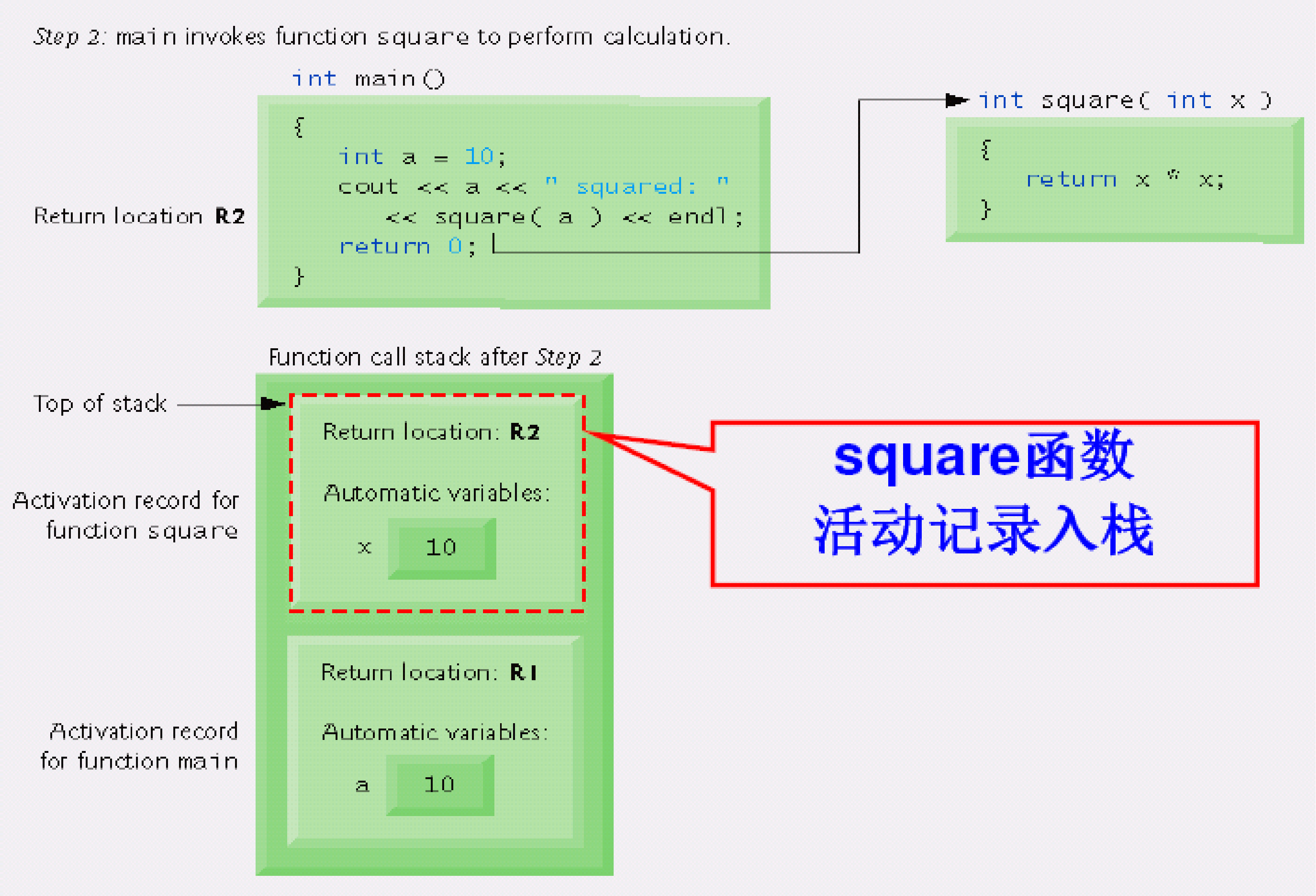
Step 1: Operating system invokes main to execute application.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

Operating system
Return location R1
Function call stack after *Step 1*
Top of stack
Return location: R1
Automatic variables:
a
10
Activation record for function main
Key
Lines that represent the operating system executing instructions

main函数
活动记录入栈

Step2：main函数调用square



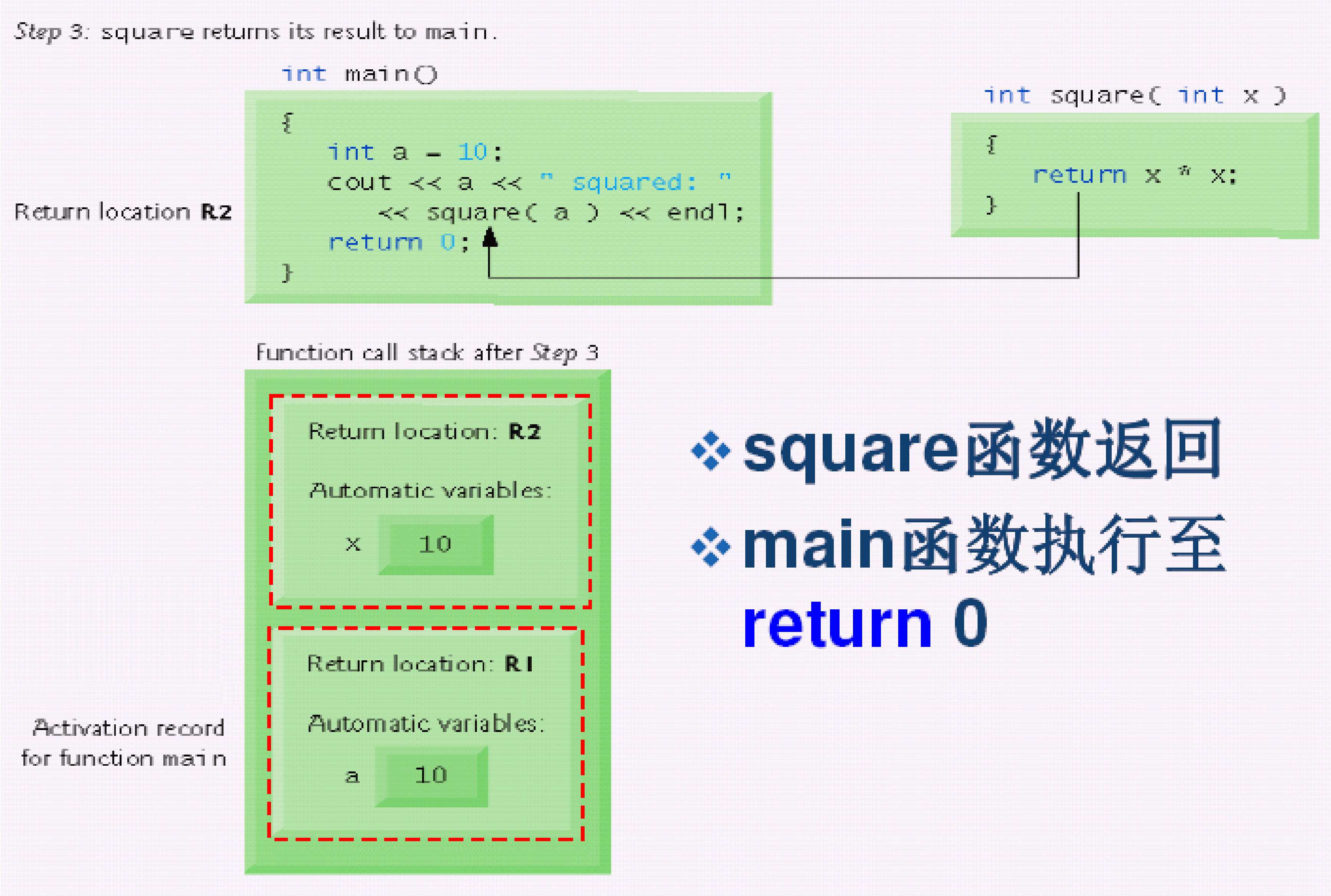
Step 2: main invokes function square to perform calculation.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

```
int square( int x )
{
   return x * x;
}
```

square函数
活动记录入栈

Step3：square结束后返回到main



Step 3: square returns its result to main.

```
int main()
{
   int a = 10;
   cout << a << " squared: "
      << square( a ) << endl;
   return 0;
}
```

```
int square( int x )
{
   return x * x;
}
```

Function call stack after *Step* 3

Return location: **R2**

Automatic variables:

x	10

Return location: **R1**

Automatic variables:

a	10

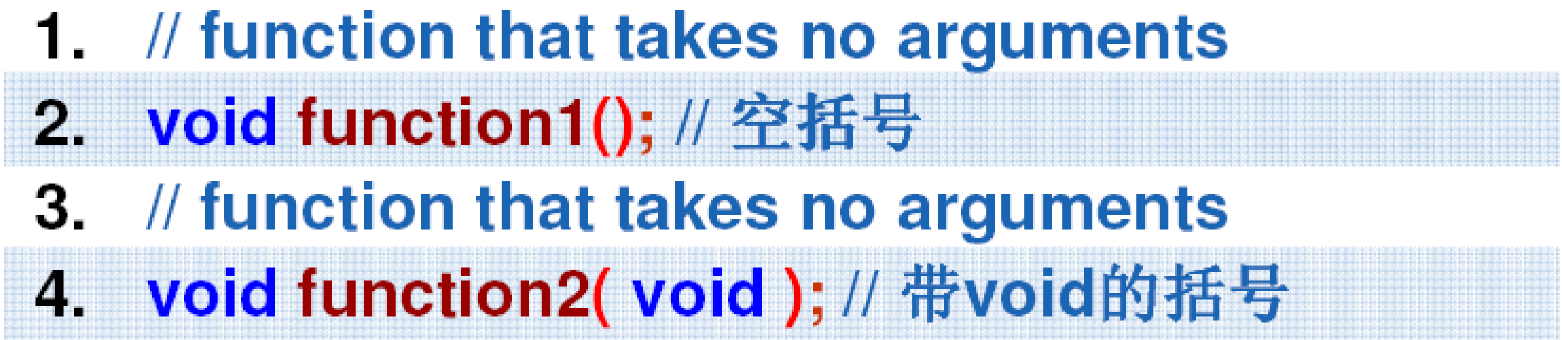
- square函数返回
- main函数执行至 return 0

Topics

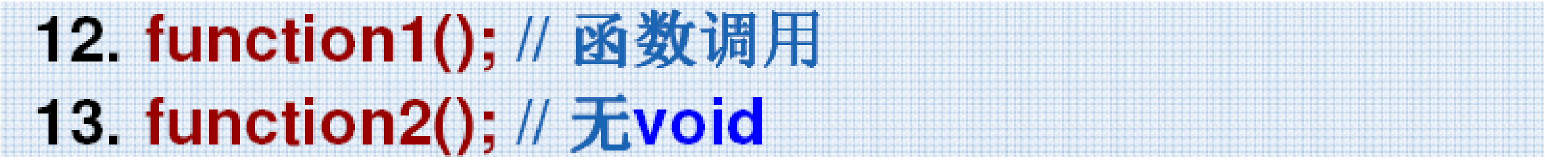
- □ 6.11 Function Call Stack and Activation Records
- □ **6.12 Functions with Empty Parameter Lists**
- □ 6.13 Inline Functions
- □ 6.14 References and Reference Parameters
- □ 6.15 Default Arguments
- □ 6.16 Unary Scope Resolution Operator
- □ 6.17 Function Overloading
- □ 6.18 Function Templates
- □ 6.19 Recursion
- □ 6.20 Example Using Recursion: Fibonacci Series
- □ 6.21 Recursion vs. Iteration

6.12 Functions with Empty Parameter Lists

- An empty parameter list (空参数列表) is specified by writing ① either void or ② nothing at all in parentheses



```
// function that takes no arguments
void function1(); // 空括号
// function that takes no arguments
void function2( void ); // 带void的括号
```

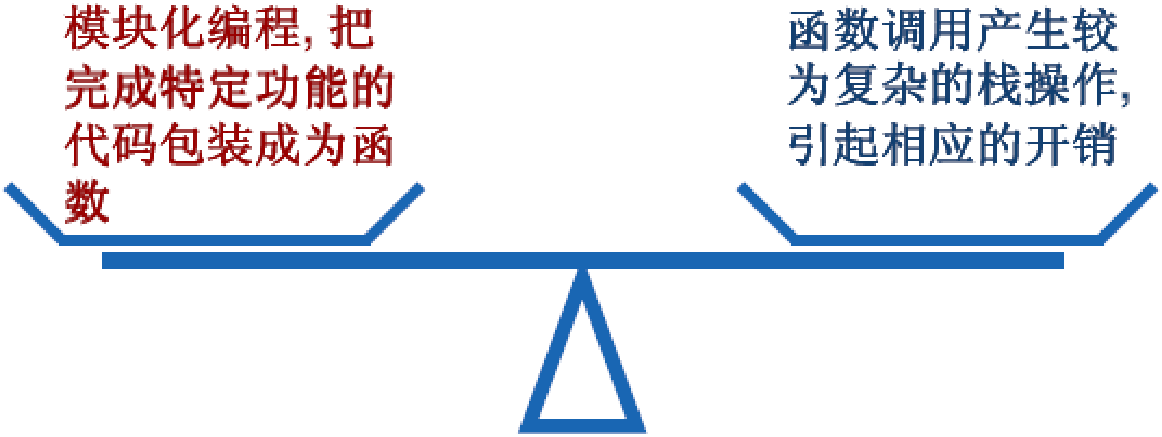


```
function1(); // 函数调用
function2(); // 无void
```

Topics

- ❑ 6.11 Function Call Stack and Activation Records
- ❑ 6.12 Functions with Empty Parameter Lists
- ❑ **6.13 Inline Functions**
- ❑ 6.14 References and Reference Parameters
- ❑ 6.15 Default Arguments
- ❑ 6.16 Unary Scope Resolution Operator
- ❑ 6.17 Function Overloading
- ❑ 6.18 Function Templates
- ❑ 6.19 Recursion
- ❑ 6.20 Example Using Recursion: Fibonacci Series
- ❑ 6.21 Recursion vs. Iteration

6.13 Inline Functions



模块化编程，把完成特定功能的代码包装成为函数

函数调用产生较为复杂的栈操作，引起相应的开销

- □ 需求：对于代码量较小的功能模块，如何既能进行包装以方便调用，又能避免函数调用引起的性能开销？

6.13 Inline Functions

- **inline function – 内联函数**

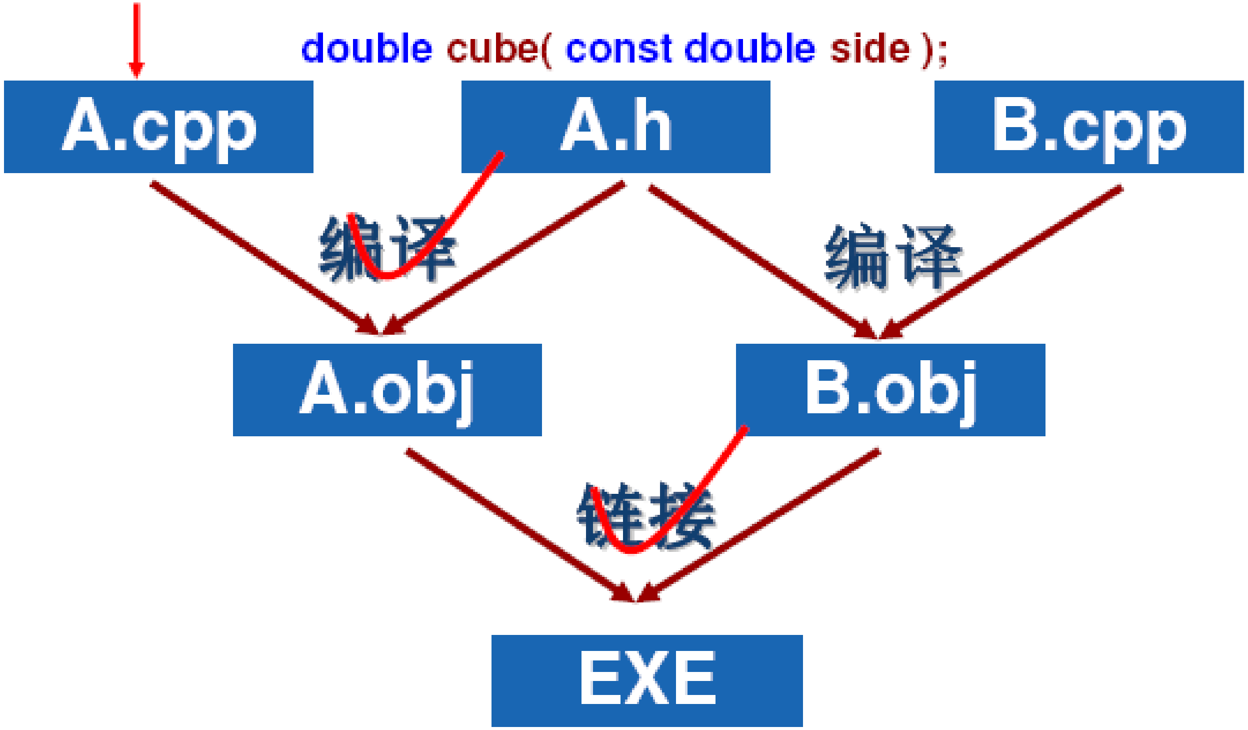
  在函数返回类型前加**inline**, 建议编译器在调用该函数的地方通过插入函数代码副本的方式来避免产生函数调用以及相应的开销

- **1. 建议**: 通常编译器不会按照**inline**的指示执行,除非该函数代码特别简单

- **2. 代码副本**:
  - 通过插入代码副本的方式避免函数调用, 会使程序体积变大;
  - 如果内联函数代码发生了改变, 那么所有应用该内联函数的源码文件都需要重新编译;
  - 直接在头文件中给出函数定义!

6.13 Inline Functions

```
double cube( const double side )
{
    return side * side * side;
}
```



double cube(const double side);

A.cpp

A.h

B.cpp

编译

编译

A.obj

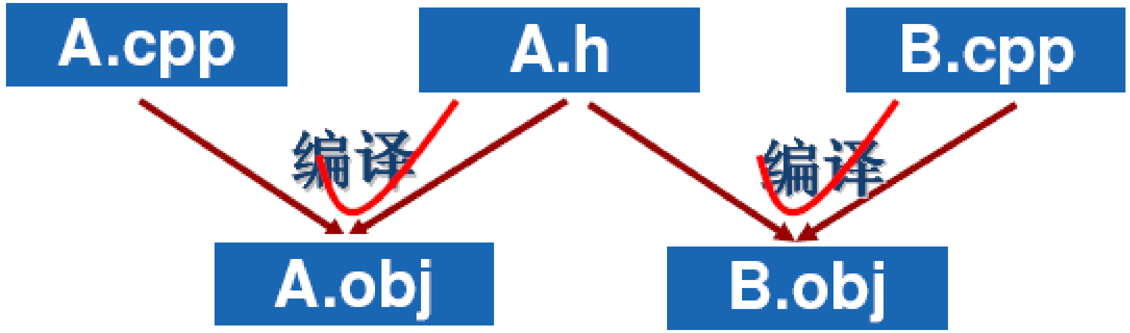
B.obj

链接

EXE

6.13 Inline Functions

```
inline double cube( const double side )
{
   return side * side * side;
}
```



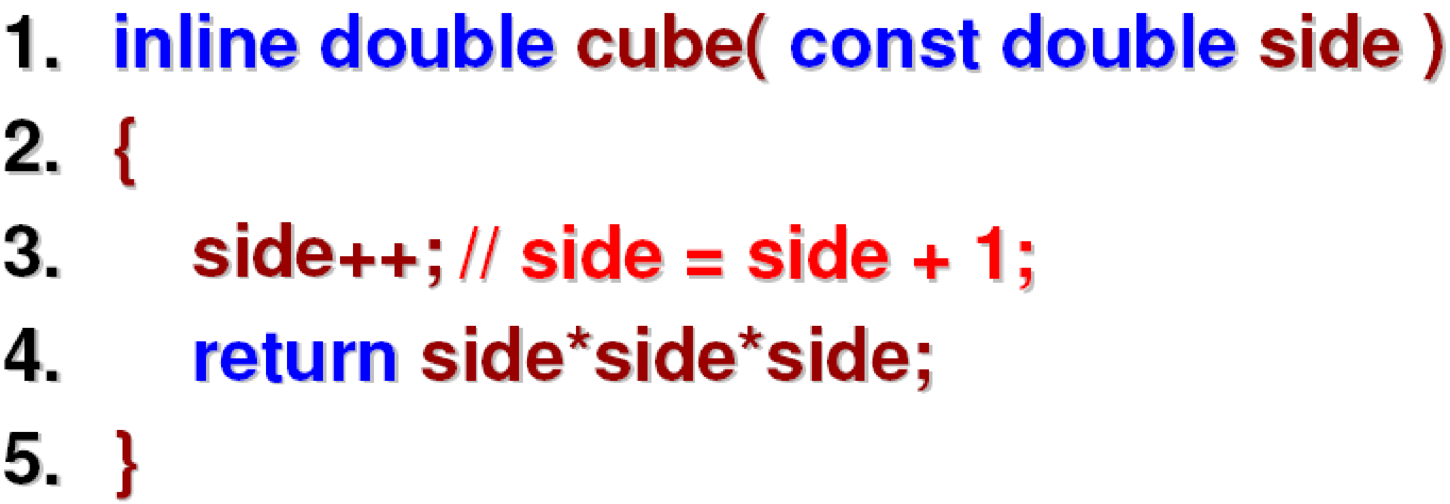
- ❑ 普通函数: 调用前必须给出声明
- ❑ 内联函数: 由于编译器需要进行代码复制, 必须在调用前给出实现代码, 因此内联函数通常直接在头文件中定义!

```
// Fig. 6.18: fig06_18.cpp
// Using an inline function to calculate the volume of a cube.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// Definition of inline function cube. Definition of function appears
// before function is called, so a function prototype is not required.
// First line of function definition acts as the prototype.
inline double cube( const double side )
{
   return side * side * side;
}

int main()
{
   double sideValue; // stores value entered by user
   cout << "Enter the side length of your cube: ";
   cin >> sideValue; // read value from user

   // calculate cube of sideValue and display result
   cout << "Volume of cube with side "
        << sideValue << " is " << cube( sideValue ) << endl;
   return 0; // indicates successful termination
}
```

const: side为常变量, 在函数体中不能被赋值和修改.

Principle of Least Privilege 最小授权原则, 仅赋予完成任务的最小权限!

6.13 Inline Functions



```
inline double cube( const double side )
{
    side++; // side = side + 1;
    return side*side*side;
}
```

❑ error C2166: l-value specifies const object

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- **6.14 References and Reference Parameters**
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.14 References and Reference Parameters

❑ 传值(pass-by-value)

```
int squareByValue( int x )
{
    return x *= x;
}
int main()
{
    int x = 2;
    cout << squareByValue(x) << “  ” << x;
    return 0;
}
```

① 将实参x的值2传递给形参，即squareByValue函数局部变量x

② squareByValue函数局部变量x初始化为2, int x = 2

③ 计算x*=x, 即x = x * x, 即x = 4

④ 返回乘法赋值表达式的值, 即4

```
4 2
```

6.14 References and Reference Parameters

❑ 需求

```
void inputGrade( int grade )
{
    cout << “请输入下一个成绩: ”;
    cin >> grade; // 输入90
}
int main()
{
    int grade = 0;
    inputGrade( grade );
    cout << grade << endl;
    return 0;
}
```

如何通过函数调用，来对实参**赋值**或者**修改**实参值？

6.14 References and Reference Parameters

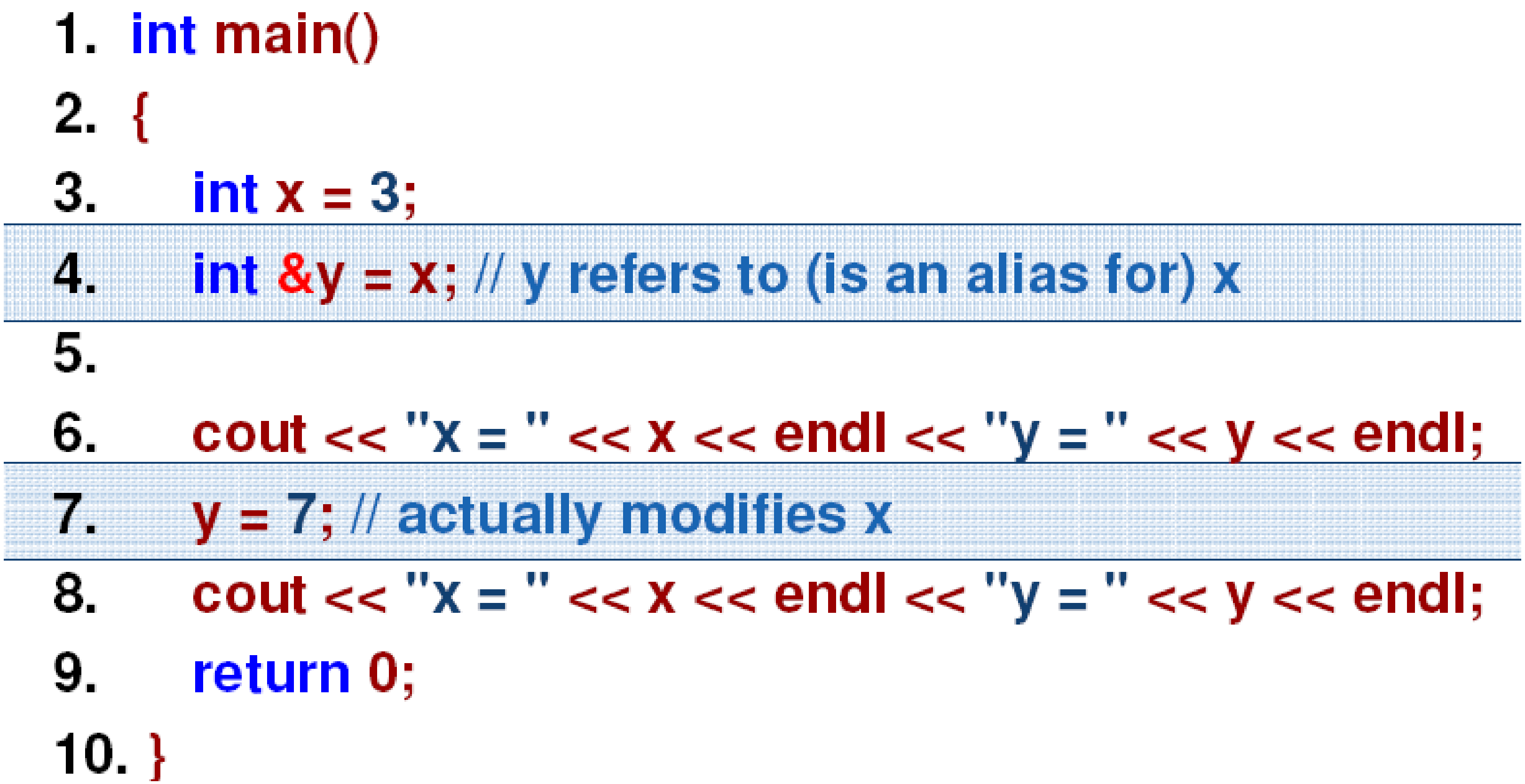
- ❑ 函数参数传递的两种方式：
- ❑ Pass-by-Value, 传值
- ❑ Pass-by-Reference, 传引用

  Reference Parameter, 引用参数

  Pointer Parameter, 指针参数(Ch 8.4)

6.14 References and Reference Parameters

- ❑ Reference(引用): 变量的别名(Alias), 通过别名可以直接访问(读/写)该变量



```
int main()
{
    int x = 3;
    int &y = x; // y refers to (is an alias for) x

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7; // actually modifies x
    cout << "x = " << x << endl << "y = " << y << endl;
    return 0;
}
```

6.14 References and Reference Parameters

- 要求: 声明引用变量时, 必须同时进行初始化, 即指出是哪个变量的引用/别名

```
1. int x = 100;
2. int &y;
3. y = x;
```

error C2530: 'y' : references **must** be initialized

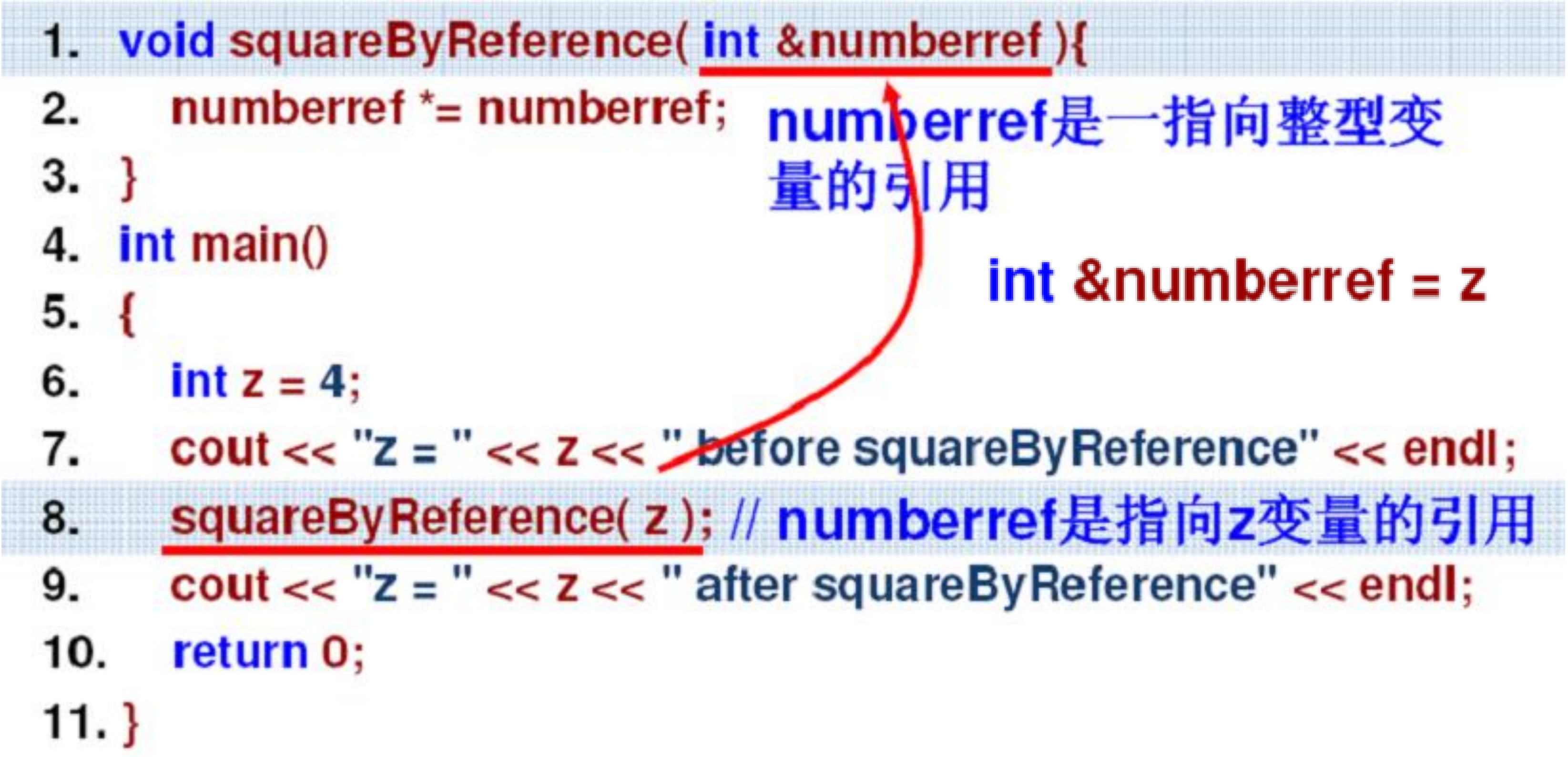
```
1. int x = 100, z = 200;
2. int &y = x;
3. y = z;
```

将z的值200赋值给别名y对应的内存空间

y = 200, x = 200

6.14 References and Reference Parameters

❑ Reference Parameter(引用参数): 函数调用过程中, 建立实参的别名



```
void squareByReference( int &numberref ){
    numberref *= numberref;
}
int main()
{
    int z = 4;
    cout << "z = " << z << " before squareByReference" << endl;
    squareByReference( z ); // numberref是指向z变量的引用
    cout << "z = " << z << " after squareByReference" << endl;
    return 0;
}
```

numberref是一指向整型变量的引用

int &numberref = z

6.14 References and Reference Parameters

```
// 错误代码
void inputGrade( int grade )
{
    cout << “请输入成绩: ”;
    cin >> grade; // 输入90
 }
int main()
 {
    int grade = 0;
    inputGrade( grade );
    cout << grade << endl;
    return 0;
 }
```

```
// 正确代码
void inputGrade(int &graderef)
{
    cout << “请输入成绩: ”;
    cin >> graderef;
}
int main()
{
    int grade = 0;
    inputGrade( grade );
    cout << grade << endl;
    return 0;
}
```

6.14 References and Reference Parameters

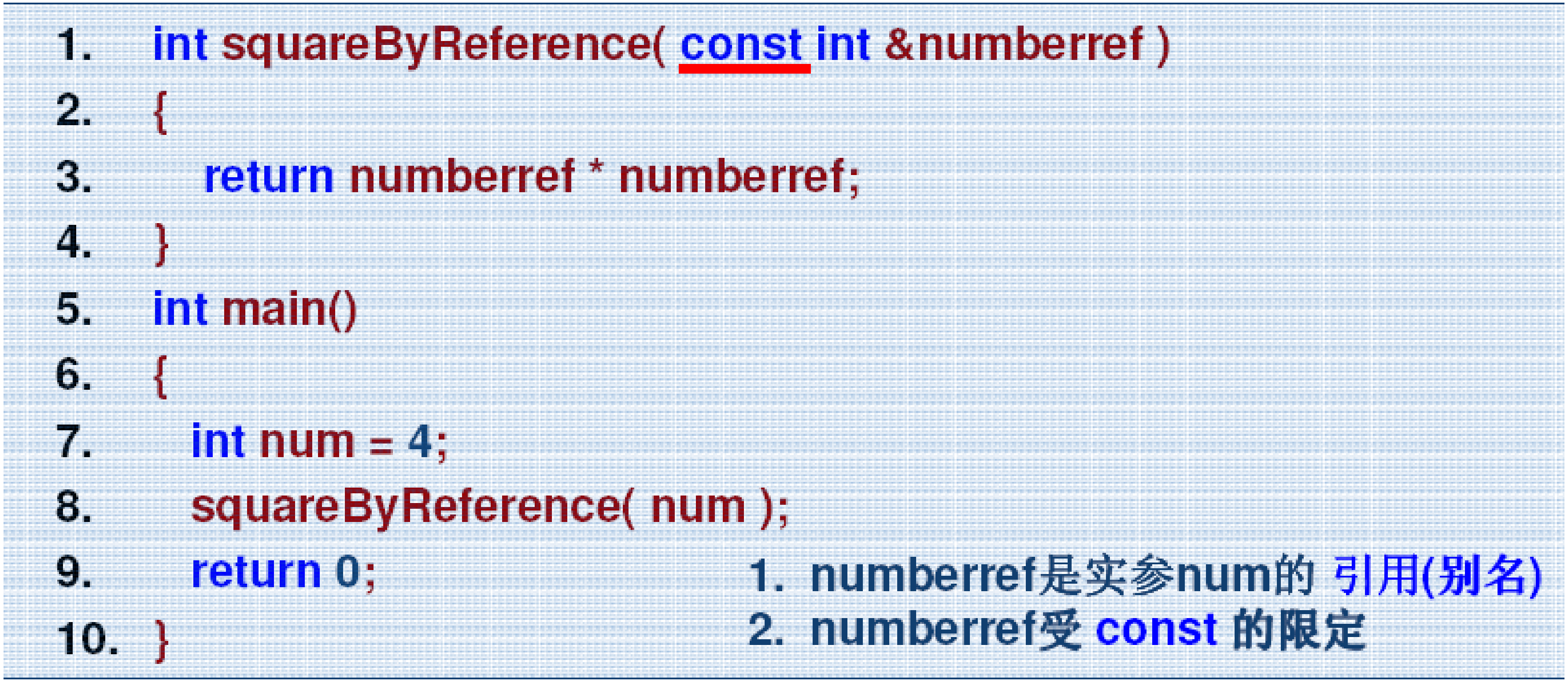
□ 引用参数的意义:

① 通过引用参数, 可以在函数体内设定或者改变实参的值;

② 避免函数调用时传值方式所需要的对实参进行的值拷贝操作, 可以有效提高程序性能.

6.14 References and Reference Parameters

- 问题: 不希望改变实参的值, 又希望避免值拷贝操作?



```
int squareByReference( const int &numberref )
{
    return numberref * numberref;
}
int main()
{
   int num = 4;
   squareByReference( num );
   return 0;
}
```

1. numberref是实参num的 引用(别名)
2. numberref受 const 的限定

6.14 References and Reference Parameters

- ❑ **Modifiable arguments,** 引用或指针参数

```
void func( int &num );
void func( int *num );
```

- ❑ **Small non-modifiable arguments,** 传值

```
int func( int num );
```

- ❑ **Large non-modifiable arguments,** 常量参数引用

```
int func( const GradeBook &book );
```

6.14 References and Reference Parameters

❑ 函数返回引用

```
int& test()
{
    int ret = 0;
    return ret;
}
```

```
int& ref = test();
```

Dangling Reference

warning C4172: returning address of local variable or temporary

```
int& test()
{
    static int ret = 0; // 必须为静态变量
    return ret;
}
```

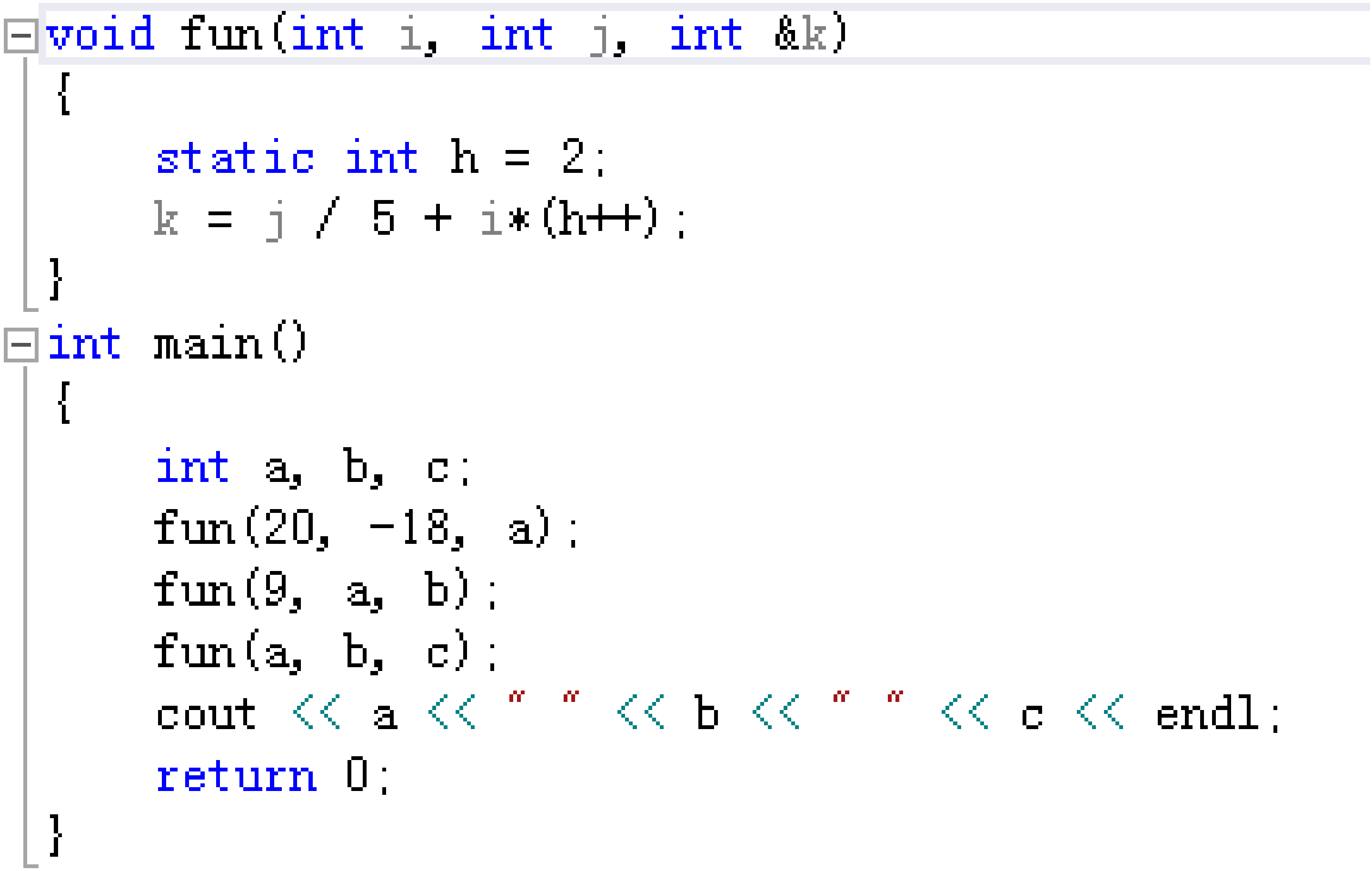
Q & A

- 1.在排序(sorting)过程中, 交换两个数据的值是很常见的操作, 设计swap函数实现对实参值的交换, 即:

```
int a = 10, b = 20;
swap( a, b ); //  a = 20, b = 10
```

Q & A

- 2.



```
void fun(int i, int j, int &k)
{
    static int h = 2;
    k = j / 5 + i*(h++);
}
int main()
{
    int a, b, c;
    fun(20, -18, a);
    fun(9, a, b);
    fun(a, b, c);
    cout << a << " " << b << " " << c << endl;
    return 0;
}
```

- 37 34 154

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

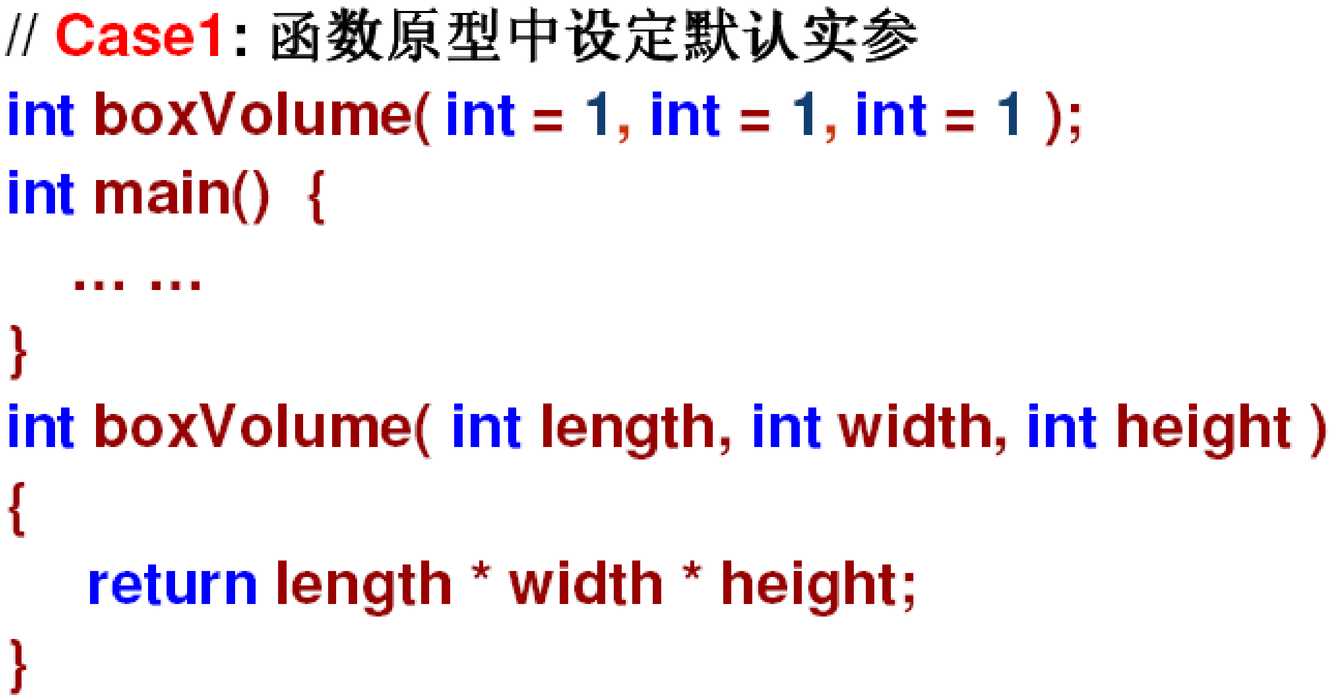
6.15 Default Arguments

- 如果在调用函数时, 经常对某个(些)形参传递相同的实参值, 那么可以为这个(些)形参设置 **Default Argument** (默认/缺省实参)

- 缺省值可以是任意表达式, 包括常量、全局变量或者函数调用(返回值)等.

6.15 Default Arguments

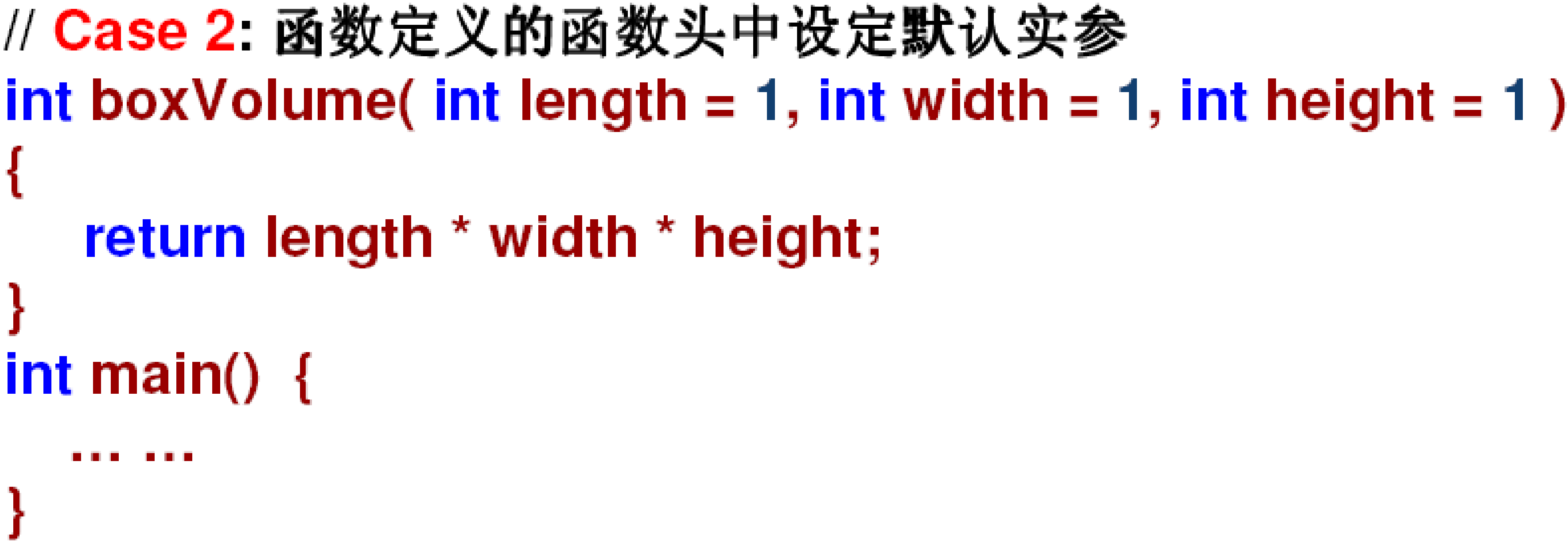
- ❑ 默认实参应在函数名第一次出现时设定, 即函数原型(或类接口), 或者函数定义的函数头部, 且仅设定一次.



```
// Case1: 函数原型中设定默认实参
int boxVolume( int = 1, int = 1, int = 1 );
int main()  {
     ... ...
}
int boxVolume( int length, int width, int height )
{
    return length * width * height;
}
```

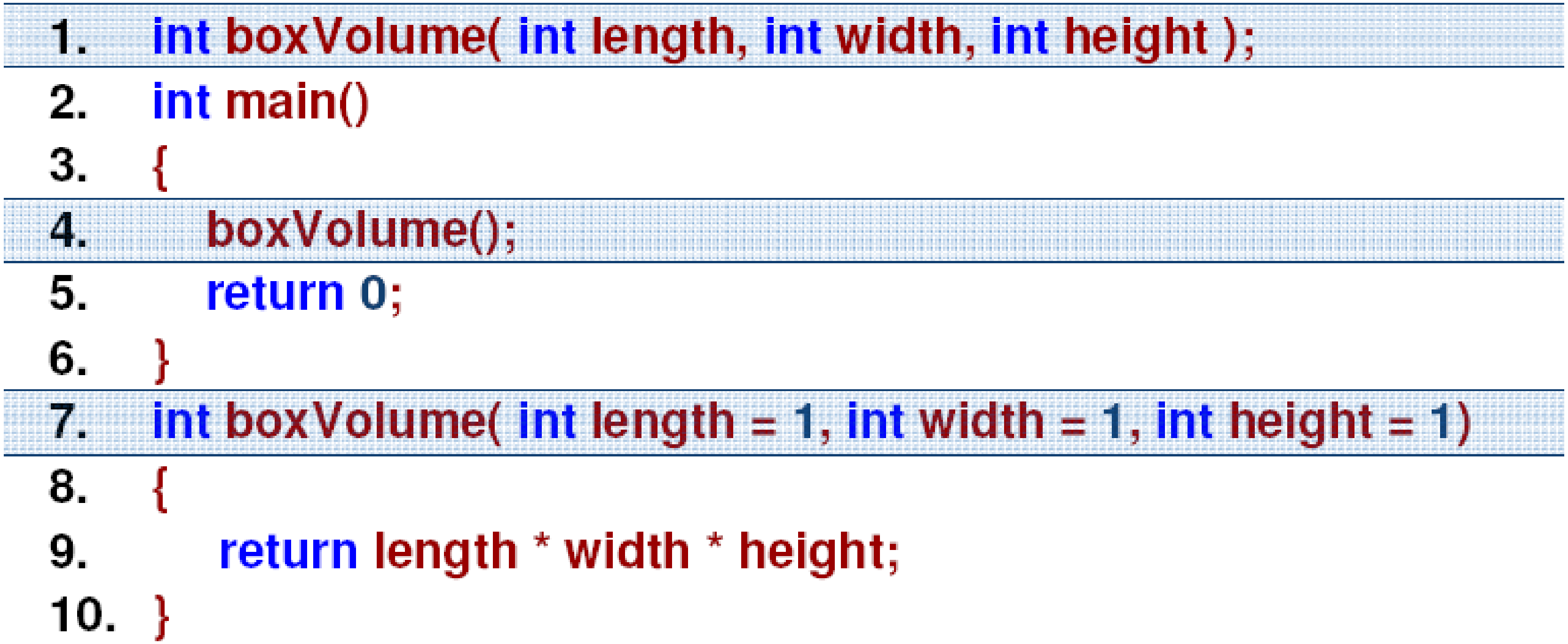
6.15 Default Arguments

- □默认实参应在函数名第一次出现时设定, 即函数原型(或类接口), 或者函数定义的函数头部, 且仅设定一次.



```
// Case 2: 函数定义的函数头中设定默认实参
int boxVolume( int length = 1, int width = 1, int height = 1 )
{
    return length * width * height;
}
int main()  {
   ... ...
}
```

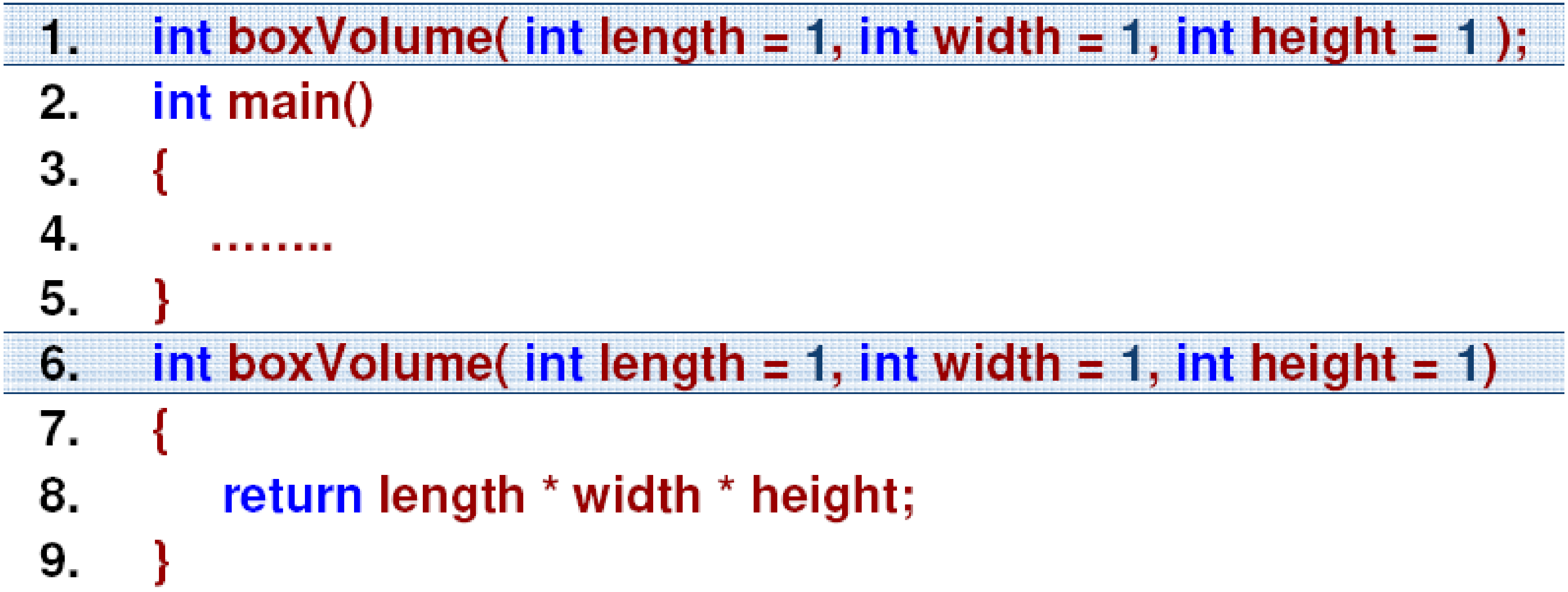
6.15 Default Arguments



```
int boxVolume( int length, int width, int height );
int main()
{
    boxVolume();
    return 0;
}
int boxVolume( int length = 1, int width = 1, int height = 1)
{
     return length * width * height;
}
```

- ❑ 无效设置, 编译器认为boxVolume函数无默认参数
- ❑ error C2660: 'boxVolume' : function does not take 0 parameters

6.15 Default Arguments

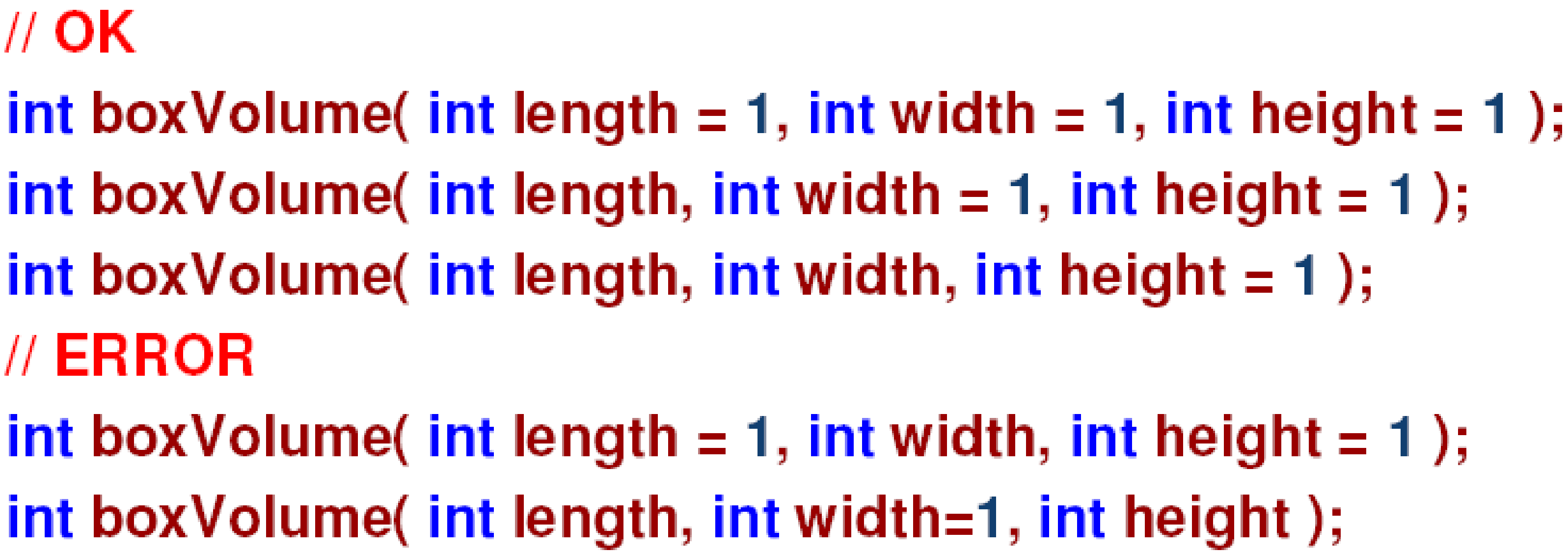


```
int boxVolume( int length = 1, int width = 1, int height = 1 );
int main()
{
      ........
}
int boxVolume( int length = 1, int width = 1, int height = 1)
{
      return length * width * height;
}
```

- error C2572: 'boxVolume' : redefinition of default parameter

6.15 Default Arguments

- ❑默认实参必须是函数参数列表中右端(尾部)的参数, 即: 如果某个参数设定了默认实参,那么该参数右边的所有参数都必须具有默认实参!



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

6.15 Default Arguments

- 显式传递给函数的实参是从左至右给形参赋值

- 函数调用时, 若省略了某个(些)实参, 那么将自动插入默认实参

6.15 Default Arguments

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

1. `boxVolume();`

   length = 1, width = 1, height = 1

2. `boxVolume(10);`

   length = 10, width = 1, height = 1

3. `boxVolume(10, 5);`

   length = 10, width = 5, height = 1

4. `boxVolume(10, 5, 2);`

   length = 10, width = 5, height = 2

6.15 Default Arguments

```
class Test{
public:
void abc(int = 1, int = 2, int = 3);
};
```

```
#include "test.h"
int main()
{
    Test test;
    test.abc(2); test.abc(2, 3);
    return 0;
}
```

```
#include <iostream>
using namespace std;
#include "test.h"
void Test::abc( int a, int b, int c )
{
    cout << "a=" << a << ", b=" << b << ", c=" << c << endl;
}
```

```
a=2, b=2, c=3
a=2, b=3, c=3
```

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- **6.16 Unary Scope Resolution Operator**
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.16 Unary Scope Resolution Operator

- ❑全局变量与局部变量同名, 如何访问全局变量?
- ❑Unary Scope Resolution Operator, 一元作用域解析符

6.16 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;
}
```

即使没有同名局部变量存在，也建议用::修饰全局变量！

Topics

- ❑ 6.11 Function Call Stack and Activation Records
- ❑ 6.12 Functions with Empty Parameter Lists
- ❑ 6.13 Inline Functions
- ❑ 6.14 References and Reference Parameters
- ❑ 6.15 Default Arguments
- ❑ 6.16 Unary Scope Resolution Operator
- ❑ **6.17 Function Overloading**
- ❑ 6.18 Function Templates
- ❑ 6.19 Recursion
- ❑ 6.20 Example Using Recursion: Fibonacci Series
- ❑ 6.21 Recursion vs. Iteration

6.17 Function Overloading

计算int型变量和double型变量的平方值

- ❑ 方案1: double square(double);
- ❑ 缺点: 归于最高级别的操作数运算, 性能损失
- ❑ 方案2:

  double squared(double);

  int squaren(int);
- ❑ 缺点: 针对每种类型参数, 需要调用对应的函数, 使用不方便

6.17 Function Overloading

判断两/三个整型数之间的最大值

- ❑ int max2(int, int);
- ❑ int max3(int, int, int);
- ❑ 缺点: 使用不方便

6.17 Function Overloading

- 函数重载: C++允许函数同名, 但要求参数不同(数量、类型或顺序)
- double square(double);
- int square(int);
- double square(int);
- int max(int, int);
- int max(int, int, int);

- Function Overloading(函数重载)
- Operator Overloading(运算符重载)

6.17 Function Overloading

C++中如何实现重载函数调用

- □ double square(double);

  int square(int);

- □ square(7.5);

  square(7);

- □ 编译器根据实参类型进行类型匹配, 以选择合适的函数进行调用

- □ 如果没有能严格匹配的, 则根据相容类型的参数隐性转化来寻求匹配

6.17 Function Overloading

```
double square( double num ){
      cout << "double version called!" << endl;
      return num * num;
}
float square( float num ){
      cout << "float version called!" << endl;
      return num * num;
}
int square( int num ){
      cout << "int version called!" << endl;
      return num * num;
}

int main()
{
      square( 1.0 );
      square( 10 );
      square( 'a' );
      return 0;
}
```

6.17 Function Overloading

C++中如何支持重载函数

- ❑ Name mangling (Name decoration)
- ❑ 编译器对函数名标识符结合函数参数的数量和类型进行编码, 以确保能调用正确的重载函数。

6.17 Function Overloading

- int square(int x)
- double square(double y)
- void nothing1(int a, float b, char c, int &d)
- int nothing2(char a, int b, float &c, double &d)
- int main()

- // Borland C++ 5.6.4, 不包含返回类型

```
@   square      $q   i
@   square      $q   d
@   nothing1    $q   i    f    c    ri
@   nothing2    $q   c    i    rf   rd
_main           // not mangled, 不能重载
```

6.17 Function Overloading

- 重载与缺省参数

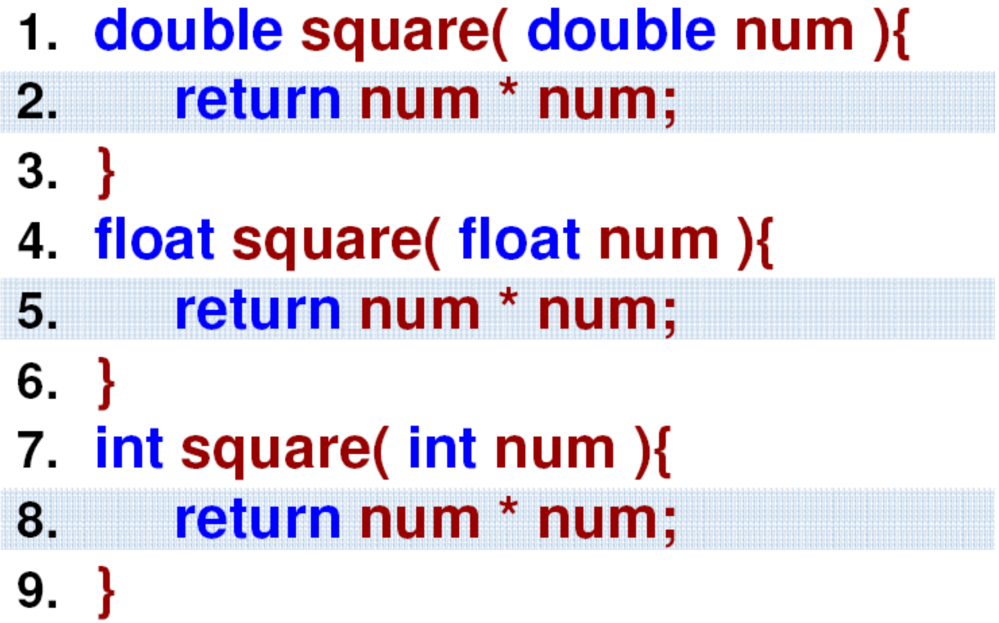
```
void test ( int a  = 1 )
{
    cout << "Function 1 is called" << endl;
}
void test()
{
    cout << "Function 2 is called" << endl;
}
int main()
{
    test();
    return 0;
}
```

```
error C2668: 'test' : ambiguous
call to overloaded function
```

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.18 Function Templates



```
double square( double num ){
      return num * num;
}
float square( float num ){
      return num * num;
}
int square( int num ){
      return num * num;
}
```

□如何避免重复编码？

6.18 Function Templates

- 重载函数是针对不同数据进行类似的操作

```
int max( int, int, int );
float max( float, float, float);
```

- 更进一步, 如果操作的程序逻辑完全相同,仅仅是操作数据的类型不同, 那么可以使用函数模板.

6.18 Function Templates

```
// Fig. 6.26: maximum.h
template < class T > // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
 T maximu
 // determi
 if ( value2
    maximu
 // determi
 if ( value3 > maximumValue )
    maximumValue = value3;
return maximumValue;
} // end function template maximum
```

- template关键字 + 尖括号< >括起来的template parameter list (模板参数列表)
- 模板参数列表:
  - 参数称为formal type parameter (形式类型参数)、占位符, 在函数调用时替换为实参的数据类型.
  - 每个参数都必须以关键词class (或typename)起头, 参数和参数之间必须以逗号分隔, 如<class T, class V>

6.18 Function Templates

```
// Fig. 6.26: maximum.h
template < class T > // or template< typename T >
T maximum( T value1, T value2, T value3 )
{
 T maximumValue = value1; // assume value1 is maximum
 // c                                              umValue
 if (
    r
 // determine whether value3 is greater than maximumValue
if ( value3 > maximumValue )
   maximumValue = value3;
return maximumValue;
} // end function template maximum
```

- T 可用于指定函数返回值、形参以及局部变量的类型
- 应至少有一形参用T修饰

6.18 Function Templates

- 函数模板的调用与普通函数一致

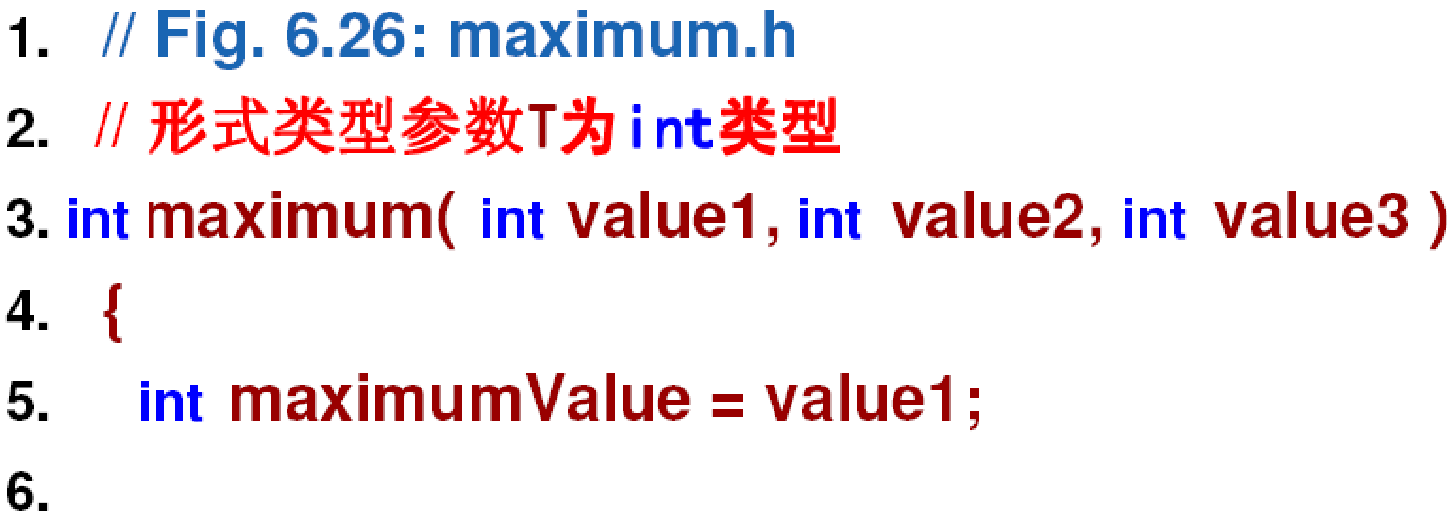
  maximum(int, int, int);

  maximum(float, float, float);

  maximum(char, char, char);

- 编译器根据实参的类型对模板进行特化, 生成各个function template specializations,即使用实参的类型替换模板定义中的形式类型参数(即占位符T)

6.18 Function Templates



```
// Fig. 6.26: maximum.h
// 形式类型参数T为int类型
int maximum( int value1, int  value2, int  value3 )
{
   int maximumValue = value1;

```

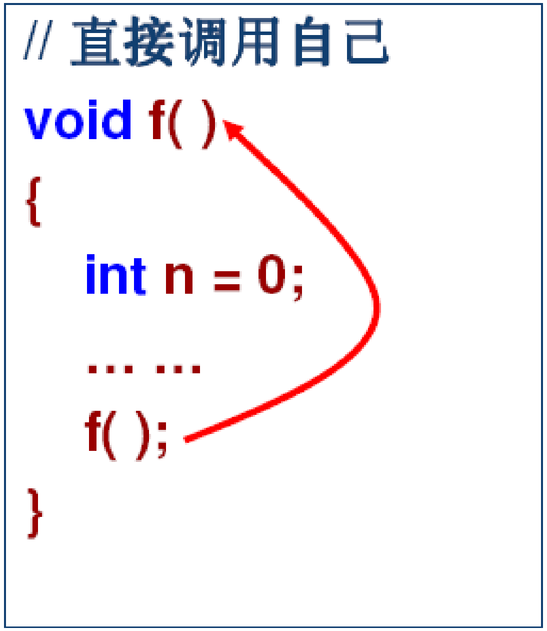
- ❑ **maximum(int, int, int);**
- ❑ 编译器确定形式类型参数T为int 型
- ❑ 生成函数模板特化

Topics

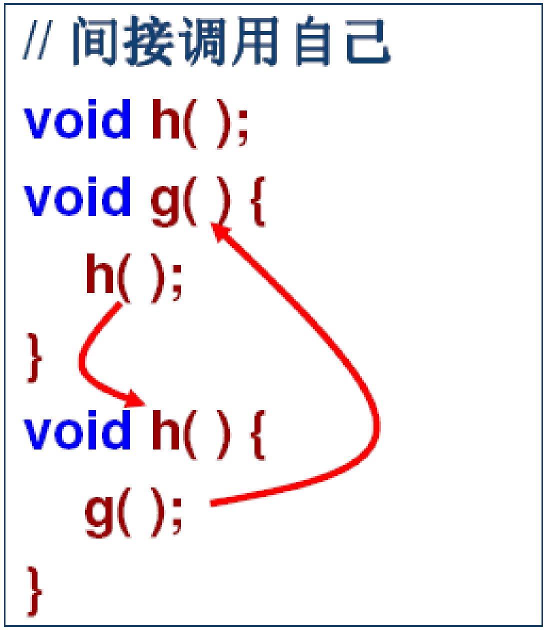
- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- **6.19 Recursion**
- 6.20 Example Using Recursion: Fibonacci Series
- 6.21 Recursion vs. Iteration

6.19 Recursion

- ❑ 递归函数(recursive function)
- ❑ 如果一个函数在其函数体中直接或间接地调用了自己, 则该函数称为 “递归函数” .



```
// 直接调用自己
void f( )
{
    int n = 0;
    … …
    f( );
}
```



```
// 间接调用自己
void h( );
void g( ) {
    h( );
}
void h( ) {
    g( );
}
```

6.19 Recursion

- 例如: 设计函数factorial求阶层n!
- 1! 直接知道结果1

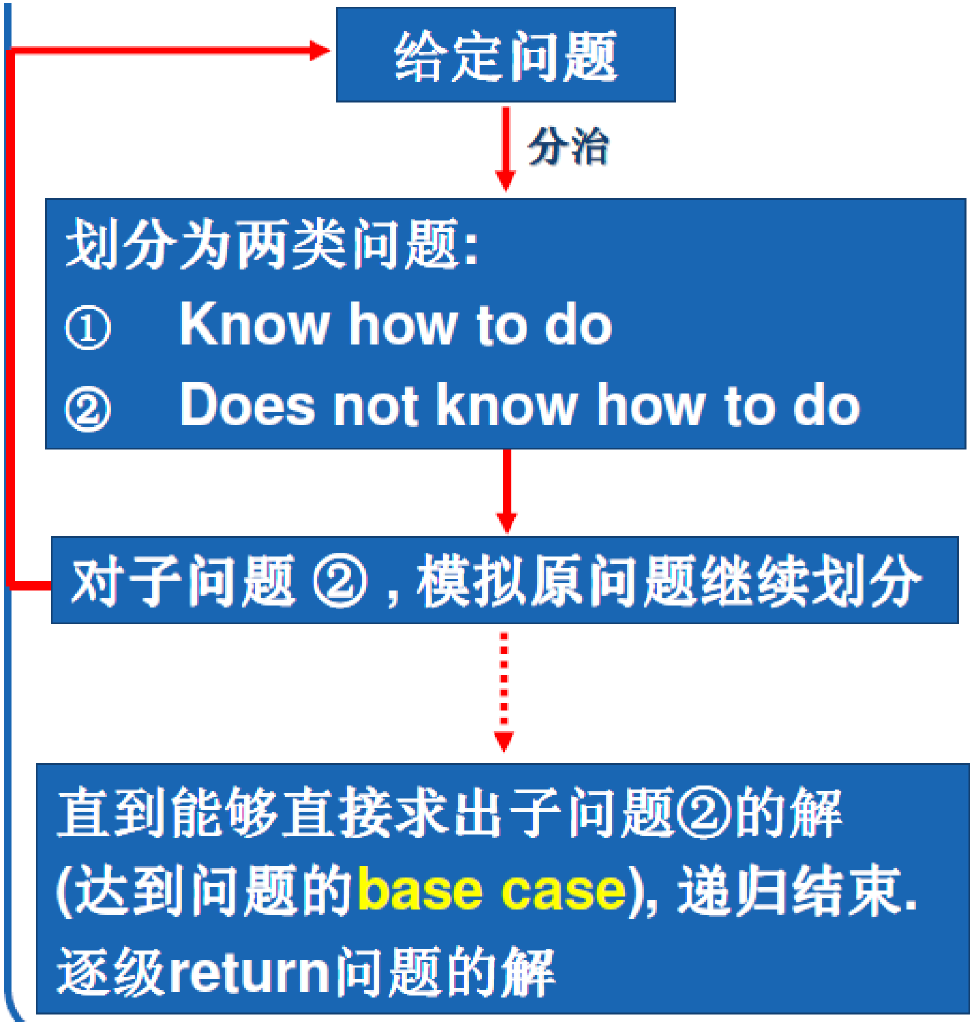
```
if( n == 1)
   return 1;
```

- 100! →100 * 99!

```
if( n > 1 )
   return n * factorial( n – 1 );
```

6.19 Recursion

如何设计递归程序



给定问题

分治

划分为两类问题:

① Know how to do

② Does not know how to do

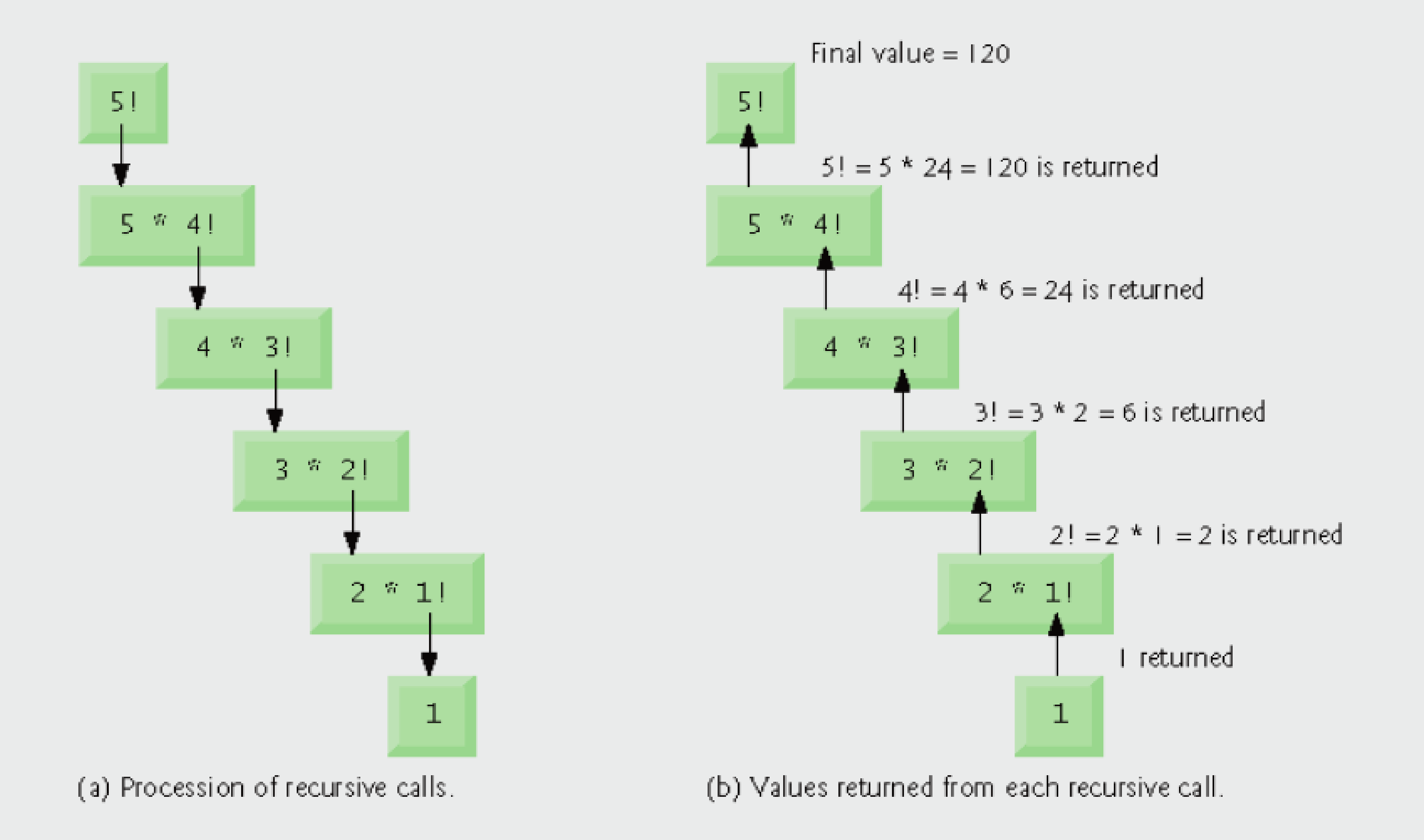
对子问题②，模拟原问题继续划分

直到能够直接求出子问题②的解(达到问题的base case)，递归结束.

逐级return问题的解

核心思路: 每次函数调用时都使问题进一步简化，从而产生**越来越小**的问题，最终简化到**基本情况**. 这时函数能识别并处理这个基本情况.

6.19 Recursion(递归) n!示例


5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 returned
1
(a) Procession of recursive calls.
(b) Values returned from each recursive call.

6.19 Recursion(递归)
n!示例

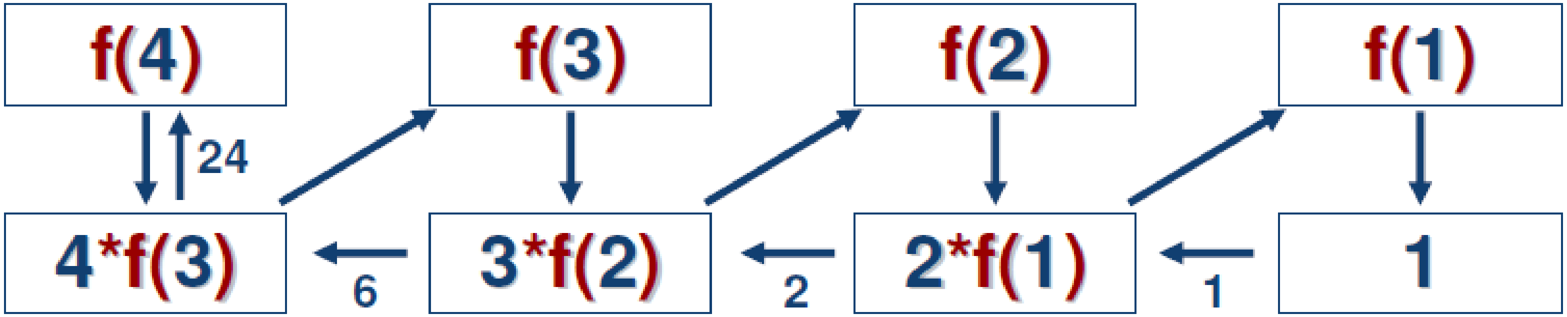
```
int f ( int n ){
  if (n == 1)
     return 1;
  else
     return n*f(n-1);
}
```

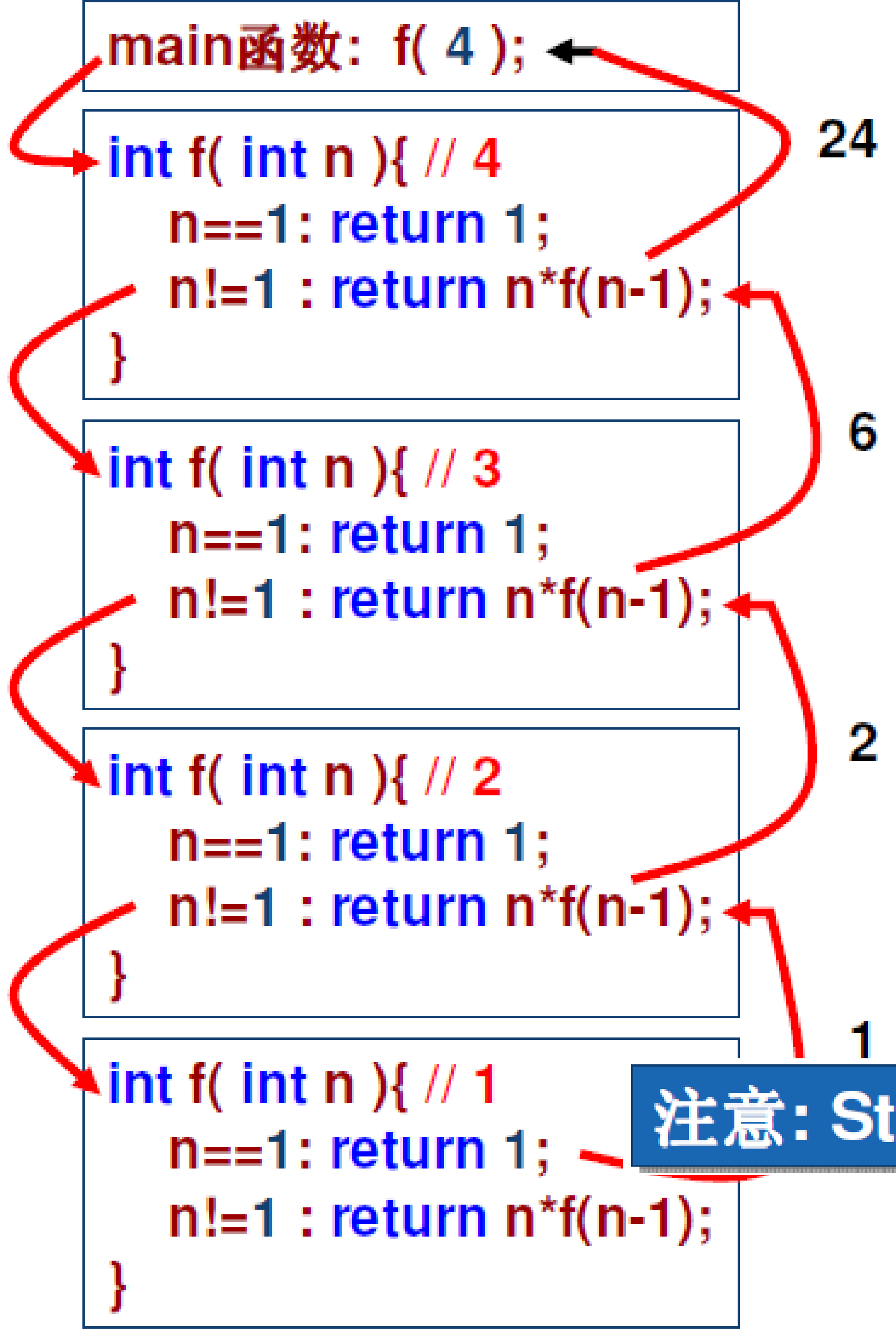
- □递推

  推导比原问题简单的问题

- □ 回归

  简单问题得到解后,回归获得原问题的解





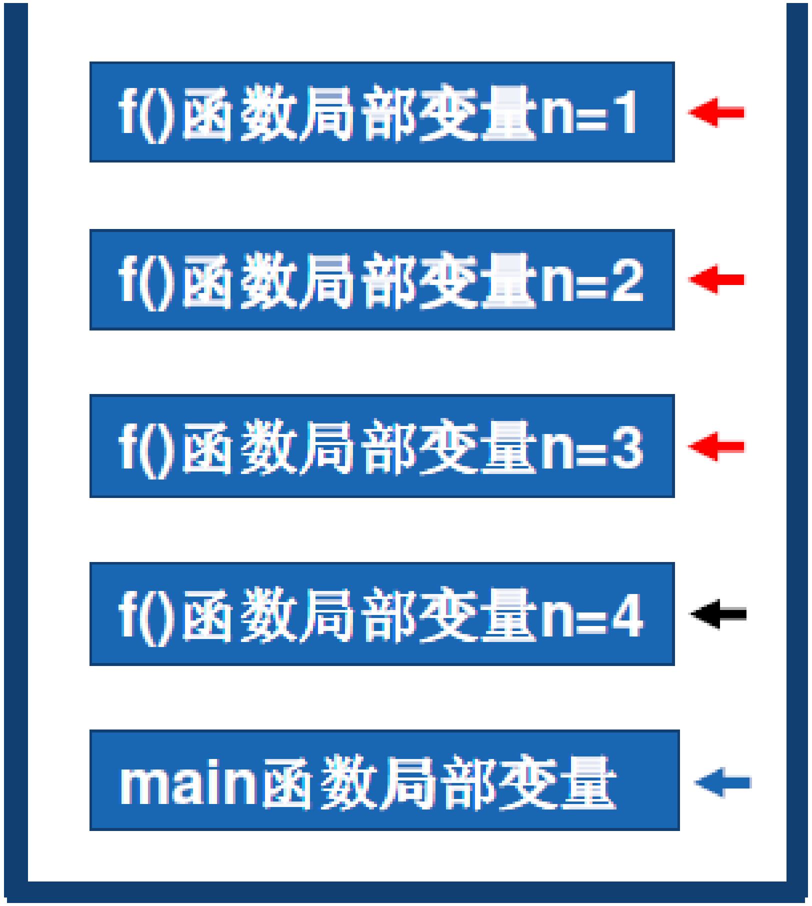
```
main函数: f( 4 );
```

```
int f( int n ){ // 4
    n==1: return 1;
    n!=1 : return n*f(n-1);
}
```

```
int f( int n ){ // 3
    n==1: return 1;
    n!=1 : return n*f(n-1);
}
```

```
int f( int n ){ // 2
    n==1: return 1;
    n!=1 : return n*f(n-1);
}
```

```
int f( int n ){ // 1
    n==1: return 1;
    n!=1 : return n*f(n-1);
}
```



f()函数局部变量n=1

f()函数局部变量n=2

f()函数局部变量n=3

f()函数局部变量n=4

main函数局部变量

函数调用栈

注意: Stack overflow, 栈溢出问题

```
// Fig. 6.29: fig06_29.cpp
// Testing the recursive factorial function.
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

unsigned long factorial( unsigned long ); // function prototype

int main()
{
    // calculate the factorials of 0 through 10
    for ( int counter = 0; counter <= 10; counter++ )
        cout << setw( 2 ) << counter << "! = " << factorial( counter )
             << endl;

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

// recursive definition of function factorial
unsigned long factorial( unsigned long number )
{
    if ( number <= 1 ) // test for base case
        return 1; // base cases: 0! = 1 and 1! = 1
    else // recursion step
        return number * factorial( number - 1 );
} // end function factorial
```

无符号长整型: 没有符号位

unsigned long:

0 ~ 4294967295(2^32-1)

int:

-2147483648 ~ 2147483648

Topics

- □ 6.11 Function Call Stack and Activation Records
- □ 6.12 Functions with Empty Parameter Lists
- □ 6.13 Inline Functions
- □ 6.14 References and Reference Parameters
- □ 6.15 Default Arguments
- □ 6.16 Unary Scope Resolution Operator
- □ 6.17 Function Overloading
- □ 6.18 Function Templates
- □ 6.19 Recursion
- □ **6.20 Example Using Recursion: Fibonacci Series**
- □ 6.21 Recursion vs. Iteration

6.20 Example Using Recursion: Fibonacci Series

Fibonacci Series

- ❑ **0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...**

从数列第3项开始, 每一项等于前两项之和

意义: 随着数列项数的增加, 前一项与后一项之比越来越逼近黄金分割的数值**0.6180339887**
……(1.618)

- ❑ 定义

$$\text{fibonacci(n)=}\begin{cases} 0 & (n=0) \\ 1 & (n=1) \\ \text{fibonacci(n-1) + fibonacci(n-2)} & (n \geq 2) \end{cases}$$

6.20 Example Using Recursion: Fibonacci Series

- ❑ 基本情况(base case)

  n = 0时, 结果为0

  n = 1时, 结果为1

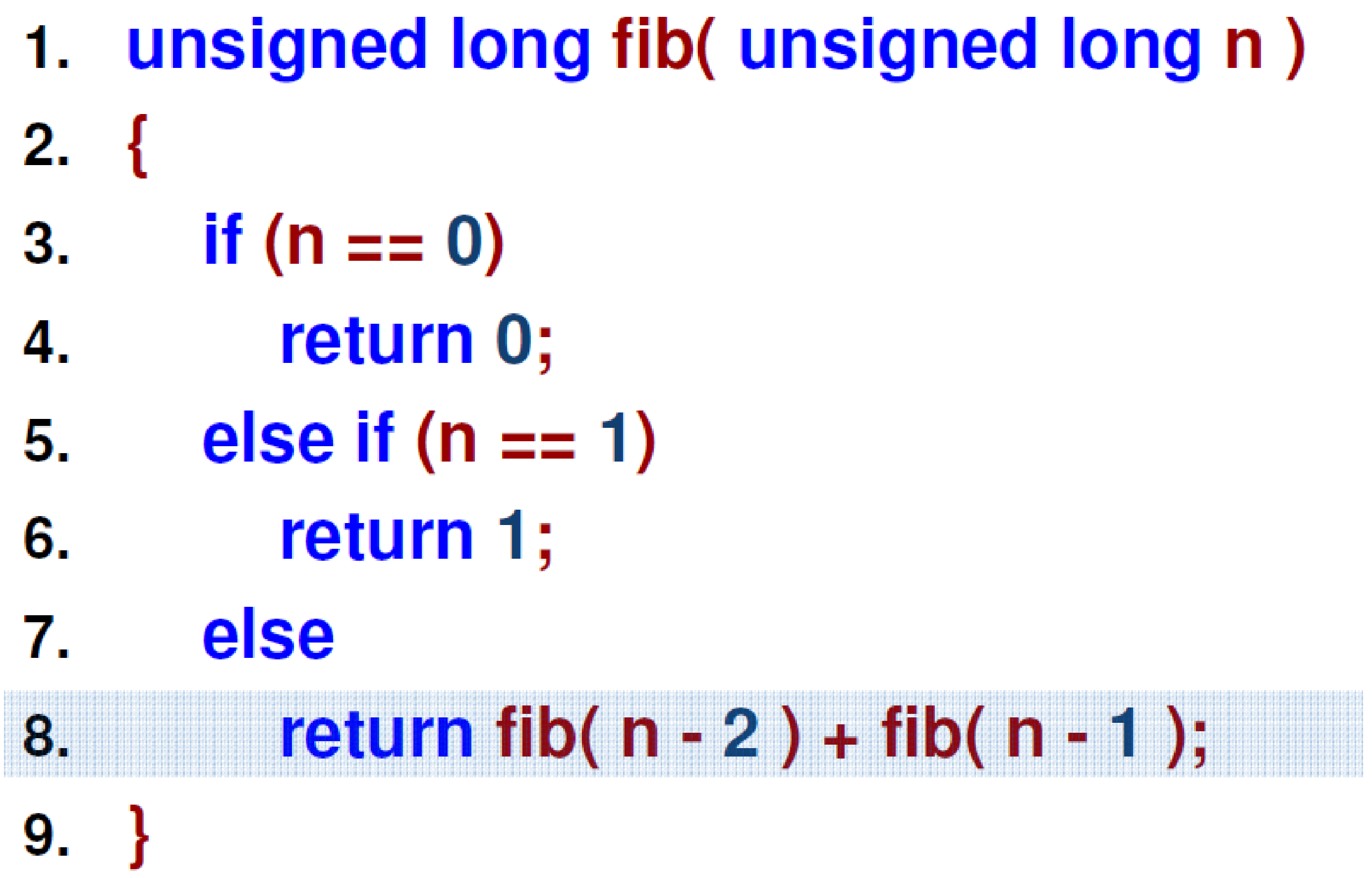
- ❑ 一般情况(general case)

  问题可分解为

  fibonacci(n-1) + fibonacci(n-2) (n≥2)

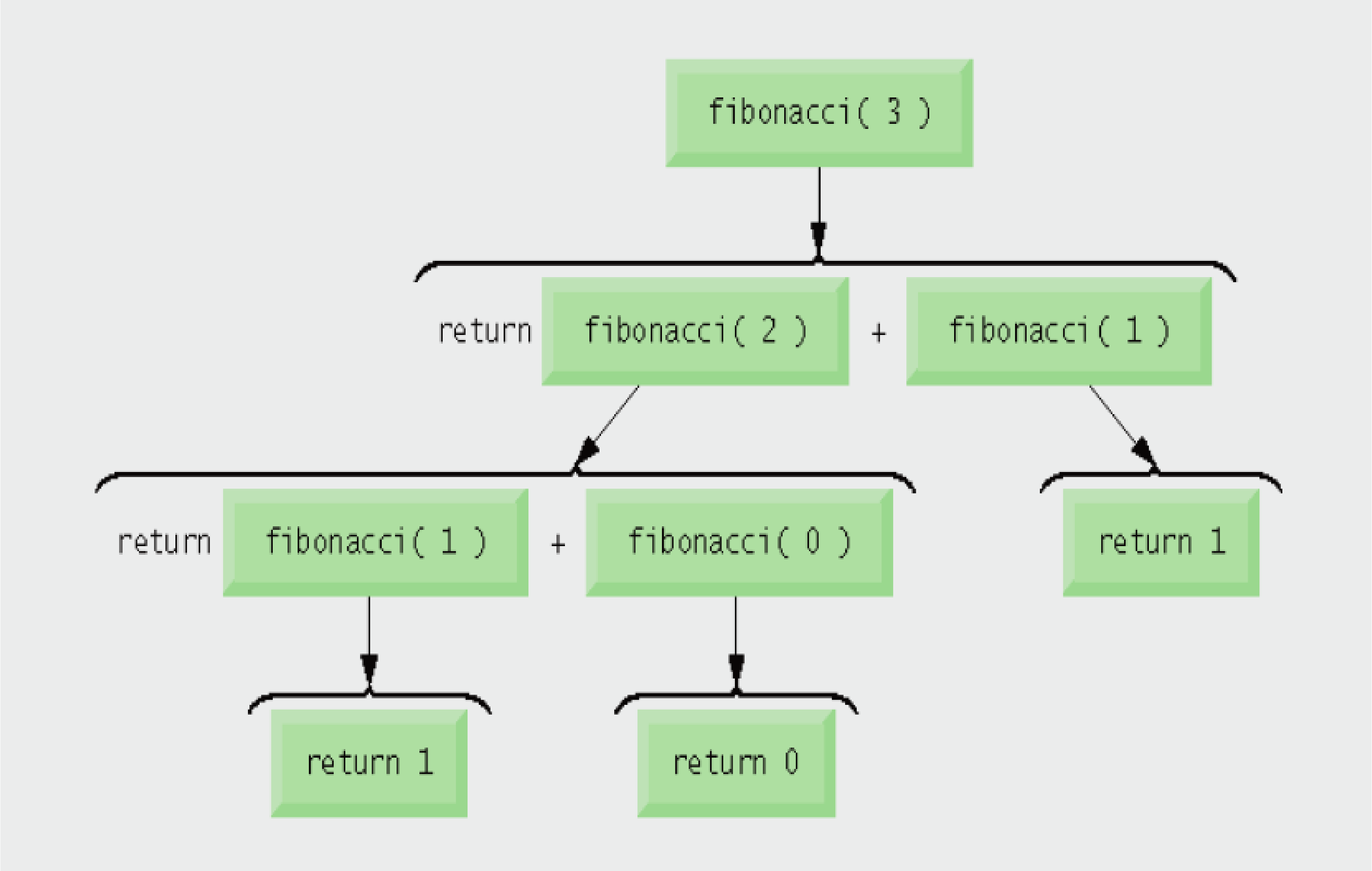
6.20 Example Using Recursion: Fibonacci Series

- 递归



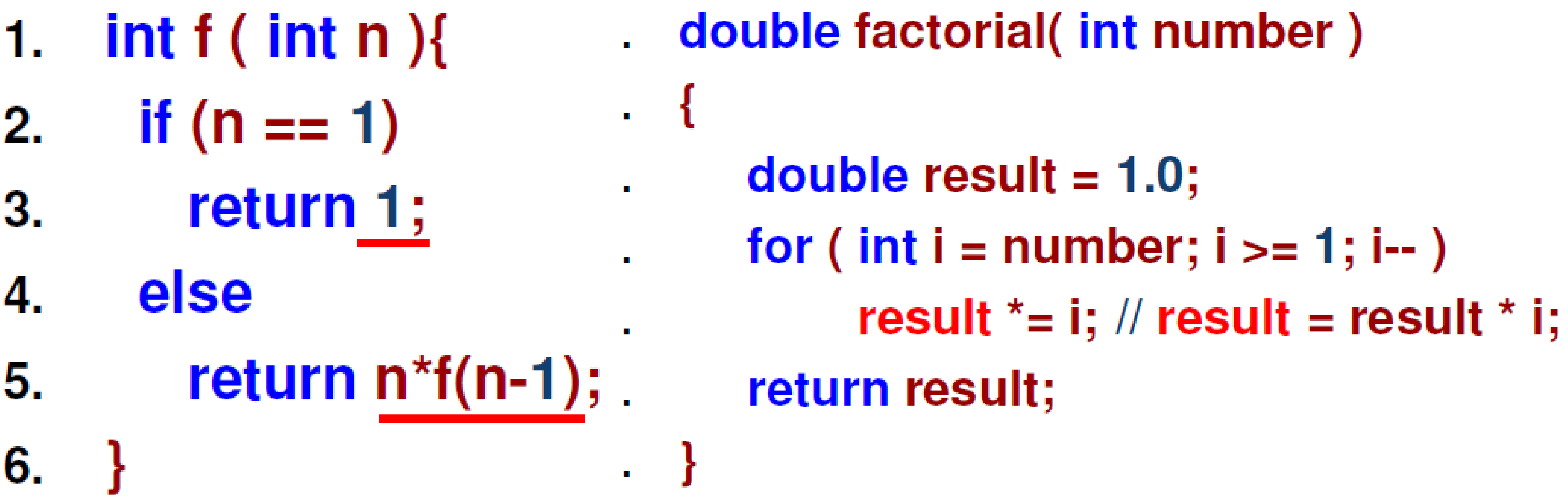
```
unsigned long fib( unsigned long n )
{
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return fib( n - 2 ) + fib( n - 1 );
}
```

6.20 Example Using Recursion: Fibonacci Series



Iteration

❑ 迭代法(iteration)也称辗转法, 是一种不断用变量的旧值推导新值的过程.



```
int f ( int n ){
  if (n == 1)
     return 1;
  else
     return n*f(n-1);
}
```

```
double factorial( int number )
{
    double result = 1.0;
    for ( int i = number; i >= 1; i-- )
          result *= i; // result = result * i;
    return result;
}
```

Iteration

如何设计迭代程序

- 确定迭代变量
  - 至少存在一个直接或间接地不断由旧值递推出新值的变量
- 建立迭代关系式
  - 如何从变量的前一个值推出其下一个值的公式(或关系)
- 对迭代过程进行控制 --- 何时结束迭代过程:
  - 一种是所需的迭代次数是个确定的值, 构建一个固定次数的循环来控制迭代过程
  - 另一种是所需的迭代次数无法确定, 需要分析出结束迭代过程的条件

6.20 Example Using Recursion: Fibonacci Series

- 迭代

```
int currentFib, n;
int f1 = 0, f2 = 1;
cin >> n;
for( int i = 1; i <= n; i++ )
{
    currentFib = f1 + f2;
    f1 = f2;
    f2 = currentFib;
}
```

注意:

- 迭代变量
- 递推公式
- 迭代结束条件

Topics

- 6.11 Function Call Stack and Activation Records
- 6.12 Functions with Empty Parameter Lists
- 6.13 Inline Functions
- 6.14 References and Reference Parameters
- 6.15 Default Arguments
- 6.16 Unary Scope Resolution Operator
- 6.17 Function Overloading
- 6.18 Function Templates
- 6.19 Recursion
- 6.20 Example Using Recursion: Fibonacci Series
- **6.21 Recursion vs. Iteration**

6.21 Recursion vs. Iteration

- ❑ 控制语句
  - ❖ 选择结构vs 循环结构
- ❑ 停止条件
  - ❖ **base case vs loop-continuation condition**
  - ❖ 递归: 当且仅当存在预期的收敛时, 才可采用递归算法
- ❑ 使用不当
  - ❖ 栈溢出vs 死循环
- ❑ 开销
  - ❖ 活动记录, 函数调用栈
  - ❖ 迭代通常发生在函数内, 无需调用函数

6.21 Recursion vs. Iteration

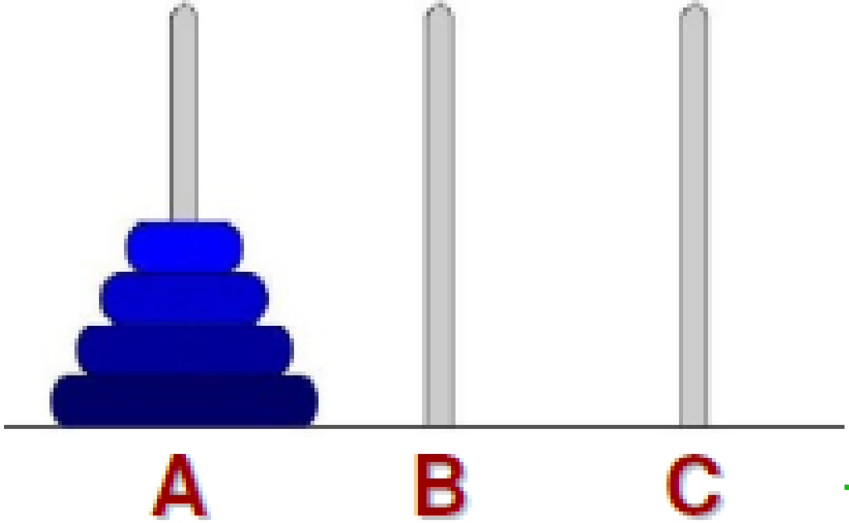
- □ 当递归算法描述问题更加直观、清晰时, 建议采用递归!

汉诺塔(Hanoi tower)问题

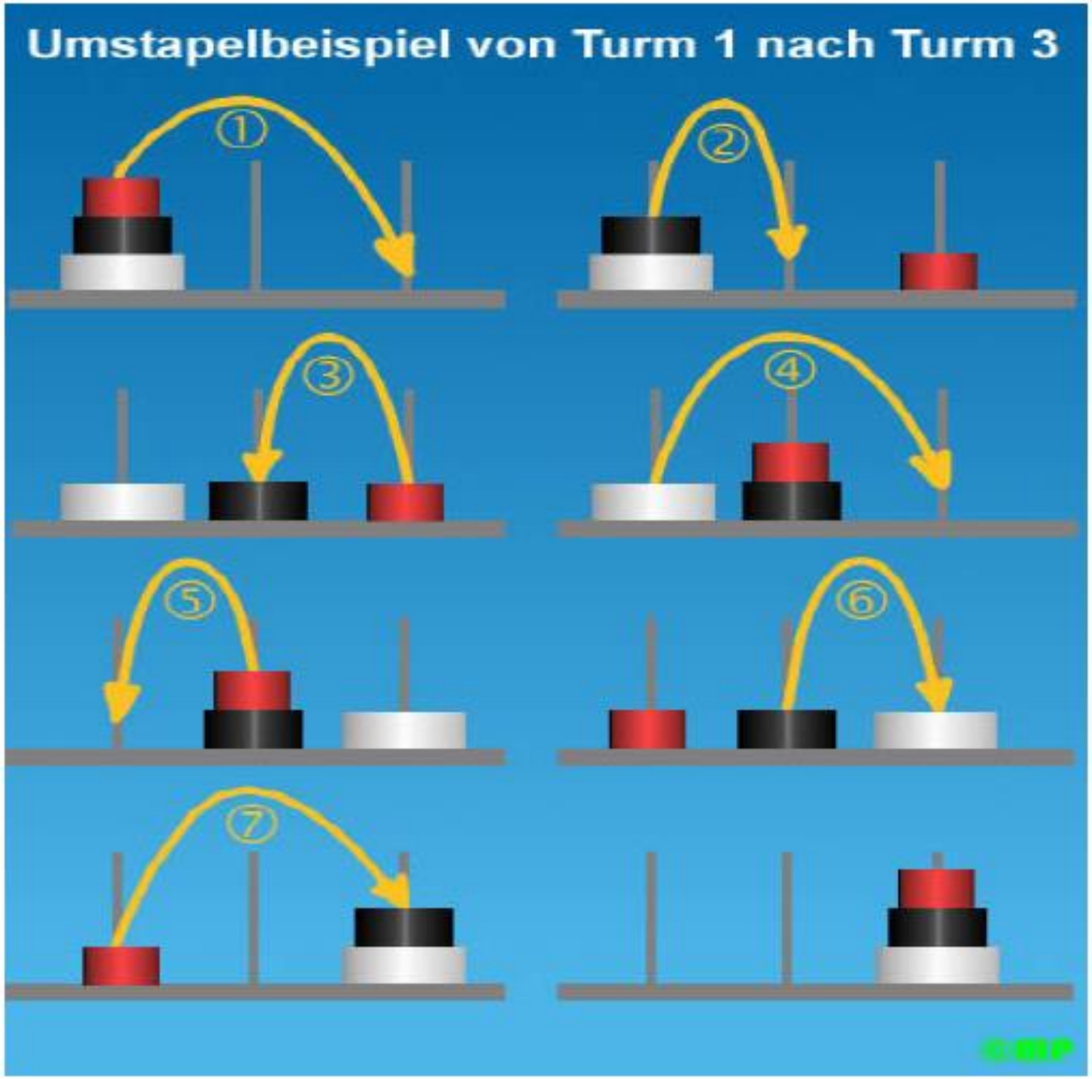
- 传说(legend)
- • 印度某间寺院有三根柱子, 上串64个金盘. 寺院里的僧侣依照一个古老的预言, 每次只能移动一个圆盘, 并且大盘不能叠在小盘上面.
- • 预言说当这些盘子移动完毕, 世界就会灭亡. 这个传说叫做梵天寺之塔问题(Tower of Brahma puzzle).

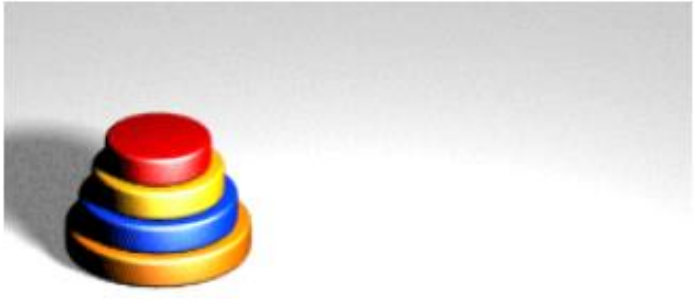
汉诺塔(Hanoi tower)问题

- 有三根相邻的柱子, 标号为A, B, C. A柱子上从下到上按金字塔状叠放着n个不同大小的圆盘, 现在把所有盘子一个一个移动到柱子C上, 可以借助于B柱. 要求每次移动: 同一根柱子上都不能出现大盘子在小盘子上方，请打印移动的步骤.



汉诺塔(Hanoi tower)问题





汉诺塔(Hanoi tower)问题

- 如果只有一个盘子, 则不需要利用B座, 直接将盘子从A移动到C. (移动2^1-1次)
- 如果有2个盘子, 先将一个盘子从A移到到B, 然后把第2个盘子从A移动到C, 再把盘子从B移到到C (2^2-1)
- 如果有3个盘子, 那么根据2个盘子的结论, 可以借助C将A上的两个盘子从A移动到B; 将盘子1从A移动到C, A变成空座; 借助A座, 将B上的两个盘子移动到C. 这说明: 可以借助一个空座, 将3个盘子从一个座移动到另一个(2^3-1).
- 如果有4个盘子, 那么首先借助空座C, 将盘子A上的三个盘子从A移动到B; 将盘子1移动到C, A变成空座; 借助空座A,将B座上的三个盘子移动到C (2^4-1)
- 上述的思路可以一直扩展到64个盘子的情况: 可以借助空座C将盘子1上的63个盘子从A移动到B; 将盘子1移动到C,A变成空座; 借助空座A, 将B座上的63个盘子移动到C(2^{64}-1)

汉诺塔(Hanoi tower)问题

- 基本情况
- 如果只有一个盘子, 则不需要利用B, 直接将盘子从A移动到C.

- 一般情况
- 将n-1个盘子借助于C从A移到B
- 把第n个盘子从A移到C
- 再把n-1个盘子从B移到C

```
void towers( int, char, char, char ); // function prototype
int main()
{
   int nDisks;
   cout << "Enter the starting number of disks: ";
   cin >> nDisks;
   towers( nDisks, ‘A’, ‘C’, ‘B’ );
   return 0;
}
void towers( int disks, char start, char end, char temp )
{
   if ( disks == 1 ) // base case
      cout << start << " --> " << end << '\n';
   else {
      // move last disk from start to end
      towers( disks - 1, start, temp, end );
      // move (disks-1) disks from temp to end
      cout << start << " --> " << end << '\n';
      towers( disks - 1, temp, end, start );
   }
}
```

Summary

- 理解程序模块的设计和构建
- 熟练使用C++中与函数相关标准库函数(例如相关数学库函数、输入输出函数、工具类的函数、字符串等)
- 熟练掌握随机数的生成
- 会使用枚举类型
- 掌握函数的定义、调用. (包括函数原型、强制参数类型转换、函数签名的使用)
- 理解形参和实参
- 理解函数相关的标识符的存储类别、作用域规则
- 理解函数调用栈和活动记录
- 理解内联函数
- 掌握按值调用和按引用调用功能
- 理解函数重载和函数模板
- 了解递归算法的基本概念和基本原理

Homework

- □ 实验必选题目：
- □ 28, 29, 30, 33, 35, 36, 41, 54
- □ 实验任选题目：
- □ 37, 38, 57
- □ 作业题目(Homework)：
- □ 16, 45, 55