

第四讲 函数

潘建瑜@MATH.ECNU

目录页

Contents

1 函数基础

2 数据作用域

3 递归、函数重载

4 预编译与多文件结构

1 函数基础

- 函数的定义、声明与调用
- 函数间的参数传递
- 内联函数

函数的定义

- 函数是程序设计中，对功能的抽象，是 C++ 的基本模块
- C++ 程序是由函数构成的（一个或多个函数）
- C++程序必须有且只能有一个 **main** 函数

函数的定义

```
类型说明符  函数名(形式参数列表)
{
   语句
}
```

函数头

函数体

类型说明符 指明本函数的类型，即函数返回值的类型，若没有返回值，可用 void

函数的参数：形参

形式参数列表（简称形参）

类型说明符 变量，类型说明符 变量，. . . . . .

- 函数可以有多个或 0 个形参
- 有多个形参时，用逗号隔开，每个形参需单独指定数据类型
- 如果函数不带参数，则形参可以省略，但小括号不能省
- 形参只在函数内部有效

```
int my_max(int x, int y)  // OK
int my_max(int x,  y)     // ERROR
```

函数的返回值

- 通过 return 语句给出，如： return x+y;
- 若没有返回值，可以不写，也可以写不带表达式的 return;

函数的调用与声明

函数的调用

函数名(实参列表);

- 被调函数可以出现在表达式中，此时必须要有返回值

函数的声明

类型说明符 函数名(形参列表);

- 函数调用前须先声明，可以在主调函数中声明，也可以在所有函数之外声明
- 有时也称为函数原型

函数声明基本原则：
被调函数在主调函数后定义，须在调用前声明；
被调函数在主调函数前定义，可以直接调用

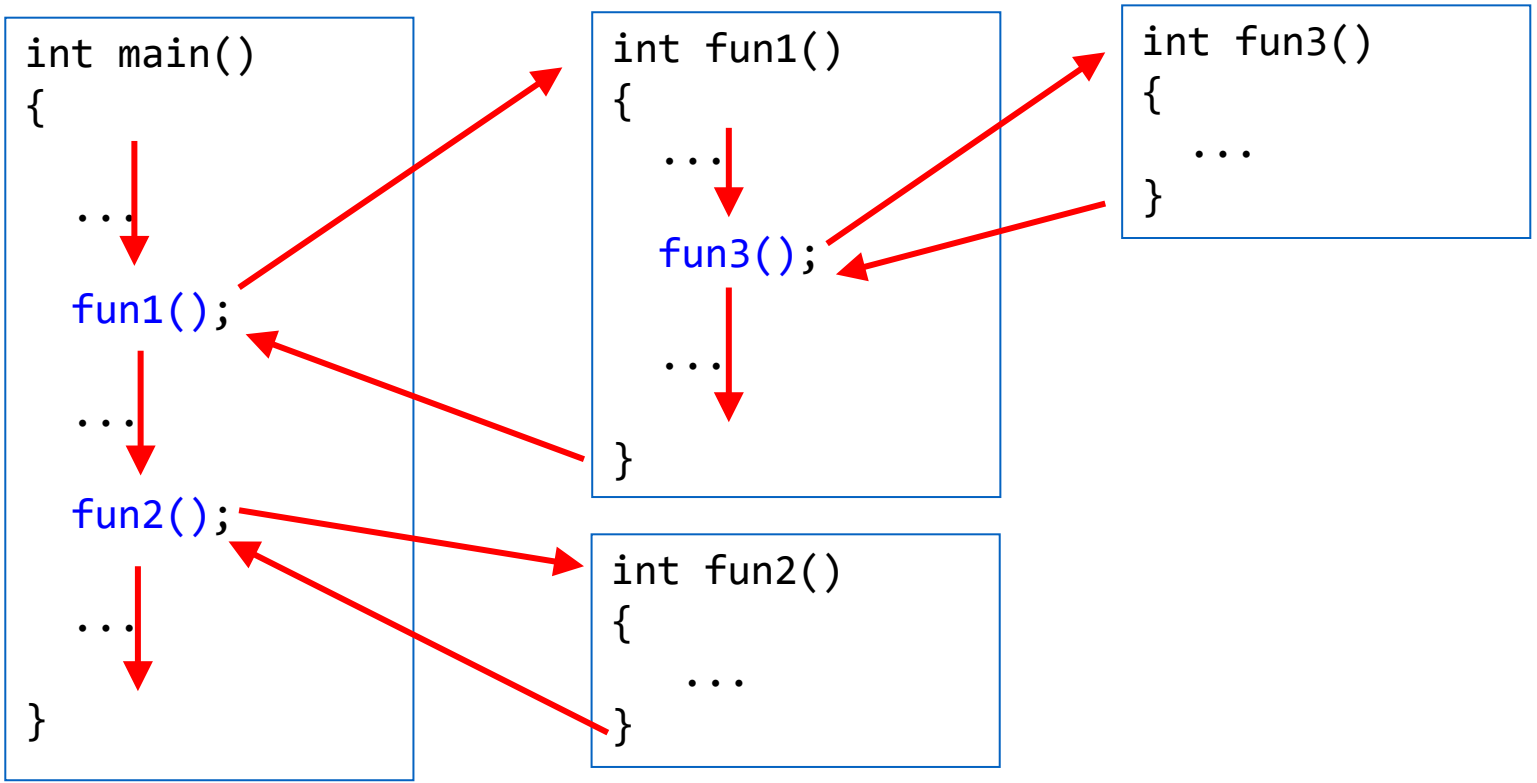
函数举例

ex04_fun_01.cpp

ex04_fun_02.cpp

```
#include <iostream>
using namespace std;
int my_max(int x, int y)  // 函数定义，在主调函数前
{
    if (x > y) return x;
    else return y;
}
 int main()
{
    int  m, n, p;
    cout << "please input m and n: " ;    cin >> m >> n;
    p = my_max(m,n); // 函数调用
    cout << "max(" << m <<"," << n <<") =" << p << endl;
    return 0;
}
```

函数调用过程



C++ 程序总是从主函数开始，最后在主函数中结束

函数举例：二进制转十进制

例：编写函数，将一个二进制正整数转化为相应的十进制数

例如：$(10101)_2 = 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

分析：需解决两个问题

（1）如何提取每个数位上的数字；

（2）如何确定 2 的次数。

提示：从右往左计算，“模10除10法”

ex04_bin2dec.cpp

思考	如何提高计算效率：降低计算 2^k 的运算次数？（Horner 算法）
思考	如何计算 1111 1111 1111 1111 对应的十进制数？（提示：字符数组）

函数举例：计算正弦函数值

例：编写函数，利用 Taylor 展开计算 sin(π/2) 的近似值

(直到级数某项的绝对值小于 10^{-15} 为止)

$$\sin(x)=\frac{x}{1!}-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots=\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k+1}}{(2k+1)!}$$

分析：注意大数越界和误差累积，简化通项计算方法。

ex04_my_sin.cpp

思考　用上面的方法计算 $\sin\left(\frac{41\pi}{2}\right)$，结果如何?

函数举例：回文数

例：找出 11~999 之间的数 m，满足 m、m^2 和 m^3 均为回文数

分析：

（1）回文数：各位数字左右对称的整数，如 11，121，1331

（2）判别方法：利用除以10取余的方法，从最低位开始，依次取出该数的各位数字。按反序重新构成新的数，比较与原数是否相等，若相等，则该数为回文数

ex04_huiwen.cpp

函数举例：计时 clock

例：计时函数 clock：返回进程启动后所使用的 cpu 总滴答数（毫秒）

需包含头文件 ctime

```
#include <ctime>
   ...
clock_t t0, t1;
double totaltime;
   ...
t0 = clock(); // 开始计时
   ...   // 需计时的代码
t1 = clock(); // 计时结束
totaltime=(double)(t1-t0) / CLOCKS_PER_SEC;
```

ex04_clock.cpp

CLOCKS_PER_SEC: 每秒的滴答数, 通常为 1000.

函数举例：计时 time

例：计时函数 time：返回从 1970 年 1 月 1 日 0 时 0 分 0 秒至今的总秒数

需包含头文件 ctime

```
#include <ctime>
...   ...
time_t   t0, t1;
t0 = time(0);  // 或者 t0 = time(NULL)

...   ...  // 需计时的代码

t1 = time(0);
t = t1 - t0;
```

ex04_time.cpp

clock 以滴答（毫秒）为单位，time 以秒为单位

函数举例：计时 chrono（类，C++11）

例：计时 chrono：提供更高分辨率的时间，如微秒、纳秒

需包含头文件 chrono

```
#include <chrono>
   ...
auto T0 = chrono::system_clock::now(); // 当前时间点，开始计时
   ...
auto T1 = chrono::system_clock::now(); // 计时结束
auto TT1 = chrono::duration_cast<chrono::seconds>(T1-T0); // 持续时间，以秒单位
auto TT2 = chrono::duration_cast<chrono::microseconds>(T1 - T0); // 以微秒单位
   ...
```

ex04_chrono.cpp

函数举例：随机数

例：随机数的生成

需包含头文件 cstdlib

```
seed = 11;
srand(seed);  // 设置种子
x = rand();     // 返回一个随机整数
```

ex04_rand_01.cpp

- rand()：返回一个 0 ～ **RAND_MAX** 之间的伪随机整数
- srand(**seed**)：设置种子。如不设定，默认种子为 1
- 相同的种子对应相同的伪随机整数
- 每次执行 rand() 后，种子会自动改变, 但变化规律是固定的

思考　如何生成 [a, b] 中的随机整数?

思考　如何生成 [0, 1] 中的随机双精度数?

ex04_rand_02.cpp
ex04_rand_03.cpp

函数举例：随机数 random（类，C++11）

- 提供了满足各种分布的随机数生成工具，为需要随机数的程序提供广泛的选择
- 三个主要组件：随机数引擎（算法），随机数分布，随机数适配器

```
random_device rd;   // 创建一个种子
mt19937 gen(rd());  // 设置随机数引擎，并用种子初始化随机数生成器
uniform_int_distribution<> dist_int(1, 100); // 指定随机数分布
cout << "Uniform integer: " << dist_int(gen) << std::endl;
```

ex04_random.cpp

函数举例：猜数游戏

例：猜数游戏

由计算机随机产生 [1,100] 之间的一个整数，然后由用户猜测这个数。

- 根据用户的猜测情况给出不同的提示：如果猜测的数大于产生的数，则显示 Larger；小于则显示 Smaller；等于则显示 You won! 同时退出游戏。
- 用户最多有 7 次机会。

ex04_game.cpp

如何每次生成不同的随机整数？ `srand(time(0)); x = rand();`

函数的参数传递机制

传递方式一：值传递

- 形参只在函数被调用时才分配存储单元，调用结束即被释放
- 实参可以是常量、变量、表达式、函数(名)等，
  但它们必须要有确定的值，以便把这些值传送给形参
- 除了一些特殊情况下，实参和形参在数量、类型、顺序上应严格一致
- 传递时是将实参的值传递给对应的形参，即单向传递
- 形参获得实参传递过来的值后，便与实参脱离关系，
  即此后形参的值的改变不会影响实参的值

值传递举例

```
int main()
{
    ... ...
    x = 3.0; n = 2;
    y = my_pow(x, n);
    ... ...
}
```

```
double my_pow(double x, int k)
{
    ... ...
}
```

ex04_my_pow.cpp



值传递举例

思考　如何实现两个变量的值的交换呢?

```
int main()
{
    int x = 3, y = 4;
    cout << "Before swap: x = " << x << ", y = " << y << endl;

    myswap(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
}
```

```
void myswap(int x, int y)
{
    int t = x;
    x = y; y = t;
}
```

ex04_myswap_01.cpp

函数的参数传递机制

传递方式二：引用传递

- 引用是一种特殊类型的变量，可看作是变量的别名
- 通过引用传递，可以实现在改变形参值的同时改变实参值
- 引用的声明：&

```
int a;
int & ra = a; // 声明一个指向 a 的引用

a = 3;
cout << "a = " << a << endl;

ra = 5;  // ra 和 a 共享同一个存储空间
cout << "a = " << a << endl;
```

引用

- 声明一个引用时必须初始化，指向一个存在的对象
- 引用一旦初始化就不能改变，即不能再指向其它对象
- 若引用是形参，则在调用函数时才会被初始化，此时形参是实参的一个别名，对形参的任何操作也会直接作用于实参

```
void swap_new(int & a, int & b)
{  int t = a; a = b; b = t; }
```

ex04_myswap_02.cpp

```
int main()
{
  int x = 5, y = 8;
  cout << "x = " << x << ", y = " << y << endl;
  swap_new(x, y);
  cout << "x = " << x << ", y = " << y << endl;
}
```

内联函数

内联函数声明与使用

- 定义与普通函数一样，只需加关键字 inline
- 与普通函数的区别：编译时会直接用函数体进行替换
- 使用内联函数能节省参数传递、控制转移等开销，从而提高执行效率

```
inline double f(double x)  // 内联函数
{
    return 2*x*x - 1;
}
```

ex04_inline.cpp

- 内联函数通常应该 功能简单、规模小、使用频繁
- 内联函数体内 不建议使用循环语句和 switch 语句
- 有些函数无法定义成内联函数，如递归调用函数

2 数据作用域

- 什么是作用域，局部作用域
- 局部变量，全局变量
- 作用域解析运算符，命名空间
- 生存期，静态变量

局部作用域（函数体）

```
void myswap(int x, int y)
{
    int t = x;
    x = y; y = t;
}
```

数据：x, y, t

```
int main()
{
    int x = 3, y = 4;
    cout << "Before swap: x = " << x << ", y = " << y << endl;

    myswap(x, y);
    cout << "After swap: x = " << x << ", y = " << y << endl;
}
```

数据：x, y

作用域

数据的作用域：数据在程序中有效的区域

C++ 中常见的作用域

- 局部作用域（函数体，语句块等）
- 函数原型作用域（函数声明）
- 命名空间作用域
- 类作用域

函数原型作用域

- 函数原型作用域：函数声明时形参的作用范围

```
int my_max(int x, int y);
```

→ x，y 的作用域仅限于形参列表的左右括号之间

在函数声明时可省略变量名，但类型不能省！

```
int my_max(int, int);  // 只有函数声明时才能省，函数定义时不能省！
```

局部作用域

- 函数体内声明的变量，作用域从声明开始，到声明所在的语句块结束为止。

ex04_var_local.cpp

```
double my_pow(double x, int k)
{
    if (k == 1) return x;
    else
    {
        double y = 1.0;
        for (int i = 1; i <= k; i++)
            y = y * x;
        return y;
    }
}
```

x, k 的作用域

y 的作用域

i 的作用域

局部变量与全局变量

每个变量都有作用域，即在程序中哪些地方可以使用该变量（可见）

- 函数定义时的形参和函数中定义的变量，均为局部变量，只在该函数体内有效；
- 语句块中定义的变量是局部变量，只在该语句块中有效；
- for 循环初始语句中定义的变量是局部变量，只在 for 循环中有效；
- 在所有函数外定义的变量为全局变量，在它后面定义的函数中均可以使用；

  若要在它前面定义的函数中使用该全局变量，则需声明其为外部变量，即：

```
extern  类型说明符  变量名;
```

ex04_var_01.cpp
ex04_var_02.cpp

局部变量与全局变量（续）

- 若局部变量与全局变量同名，则优先使用局部变量！

ex04_var_global_01.cpp

```
#include<iostream>
using namespace std;
int k = 2;  // 全局变量
int main()
{
    int i = 5, x;  // 局部变量
    x = i + k;  cout << "x = " << x << "\n" << endl;
    {
        int k = 16; // 局部变量
        x = i + k;
        cout << "x = " << x << "\n" << endl;
    }
    x = i + k;
    cout << "x = " << x << "\n" << endl;
}
```

作用域解析运算符

- 若存在同名的局部变量和全局变量，则缺省自动引用局部变量

  若需引用全局变量，可在变量名前加 **::**

ex04_var_global_02.cpp

```
#include<iostream>
using namespace std;
int i = 2;  // 全局变量
int main()
{
    int i = 5; // 局部变量
    {
        int i = 7;  // 局部变量
        cout << "i = " << i << endl;  // i = 7
    }
    cout << "i = " << i;   // i=5
    cout << "全局变量 i = " << ::i; // 引用全局变量，i = 2
}
```

命名空间

大型程序通常由不同模块组成，不同模块中的类和函数可能存在重名。

为解决这个问题，C++引入命名空间概念。

- 命名空间的定义

```
namespace 命名空间名
{
  (命名空间内的各种声明，包括变量声明，函数声明，类声明等)
}
```

  - 命名空间内的元素，可以是变量、函数、类等的声明，均称为 名字

- 命名空间的使用：using
  - 可以将命名空间中的所有名称都导入到当前作用域中
  - 也可以只导入指定的某个名字

命名空间举例

```
namespace mynames    // 定义一个命名空间
{
    int k = 10;
    double pi = 3.14;
    int my_max(int x, int y);  // 函数只是声明，不是定义
    double my_power(double x, int k);
}
```

ex04_namespace_01.cpp
ex04_namespace_02.cpp
ex04_namespace_03.cpp

```
using namespace mynames;    // 导入 mynames 中的所有名字
```

```
using mynames::my_max;      // 只导入 my_max
```

```
mynames::my_power(1.2,3);   // 直接使用 mynames 中的 my_power 函数
```

标准命名空间

标准库的所有函数、类、对象等，都在命名空间 std 中。

```
using namespace std;    // 导入标准命名空间中所有名字
```

```
using std::cout;      // 只导入 cout
```

```
std::cout << "Hello";  // 直接调用 std 中的 cout
```

ex04_namespace_04.cpp

变量的生存期

- 每个变量都有生存期，分静态生存期和动态生存期
  - 静态生存期：生存期与程序的运行期相同，即一直有效
  - 动态生存期：当变量所在的程序块执行完后即消失

  - 全局变量和静态变量：静态生存期
  - 动态变量：动态生存期

    局部变量是动态变量，具有动态生存期

静态变量

静态变量的声明

```
static 类型说明符 变量名;
```

ex04_var_static.cpp

```
int my_plus(int x)
{
    static int i = 5;
    i = x + i;
    return i;
}
```

- 静态局部变量不会随函数的调用结束而消失，下次调用函数时，该变量会保留上次调用后的值!
- 没有初始化的静态变量会自动赋初值 0
- 静态变量只初始化一次!

3 递归、函数重载

- 函数嵌套与递归
- 形参的缺省值
- 函数重载

函数的嵌套与递归

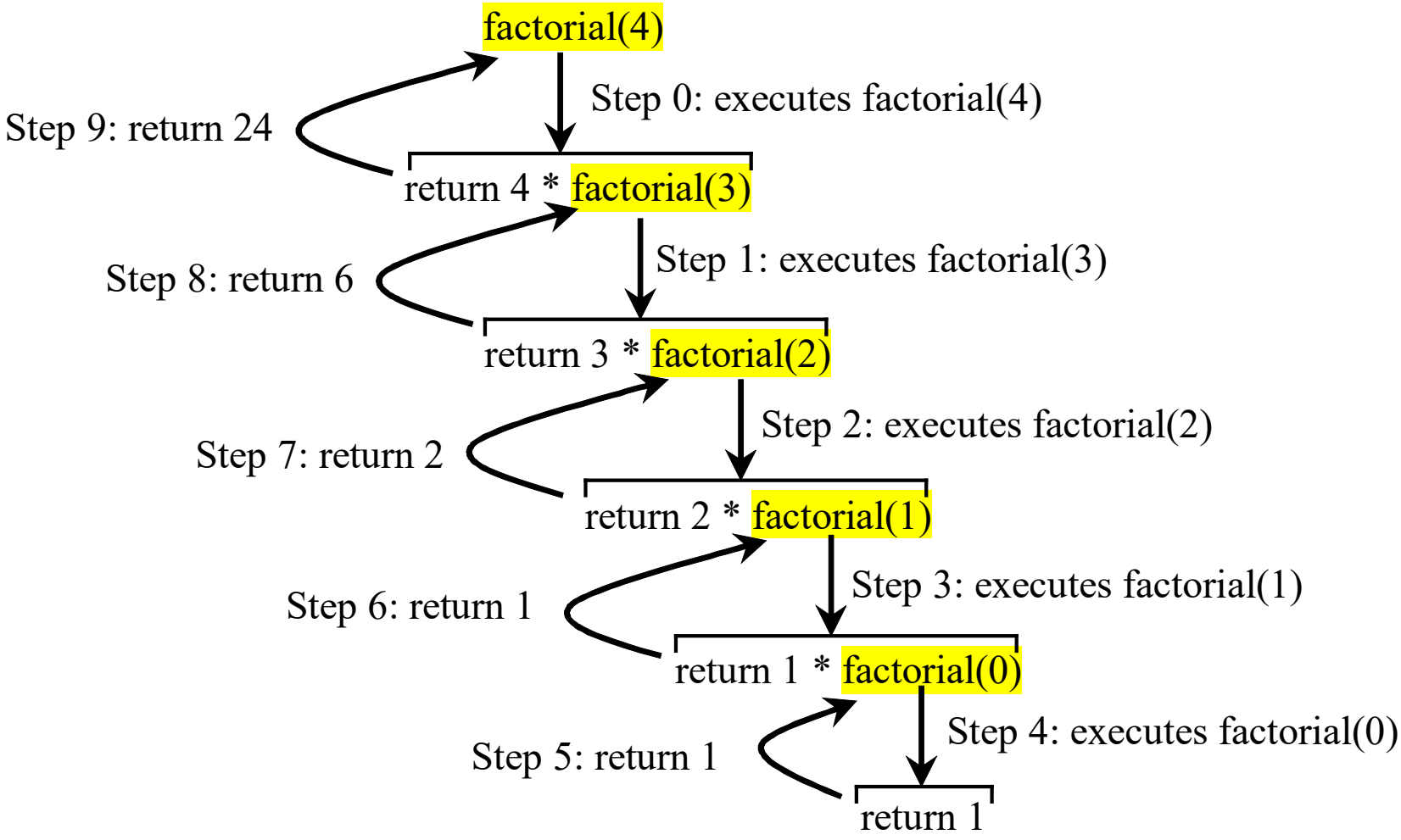
- 函数可以嵌套调用，但不能嵌套定义
- 函数也可以递归调用（函数可以直接或间接调用自己）

例：利用右边的公式计算阶乘：$$n!=\begin{cases}1 & (n=0)\\ n(n-1)! & (n>0)\end{cases}$$

ex04_factorial.cpp

对同一个函数的多次不同调用，编译器会给函数的形参和局部变量分配不同的存储空间，互不影响。

递归


factorial(4)
Step 0: executes factorial(4)
Step 9: return 24
return 4 * factorial(3)
Step 1: executes factorial(3)
Step 8: return 6
return 3 * factorial(2)
Step 2: executes factorial(2)
Step 7: return 2
return 2 * factorial(1)
Step 3: executes factorial(1)
Step 6: return 1
return 1 * factorial(0)
Step 4: executes factorial(0)
Step 5: return 1
return 1

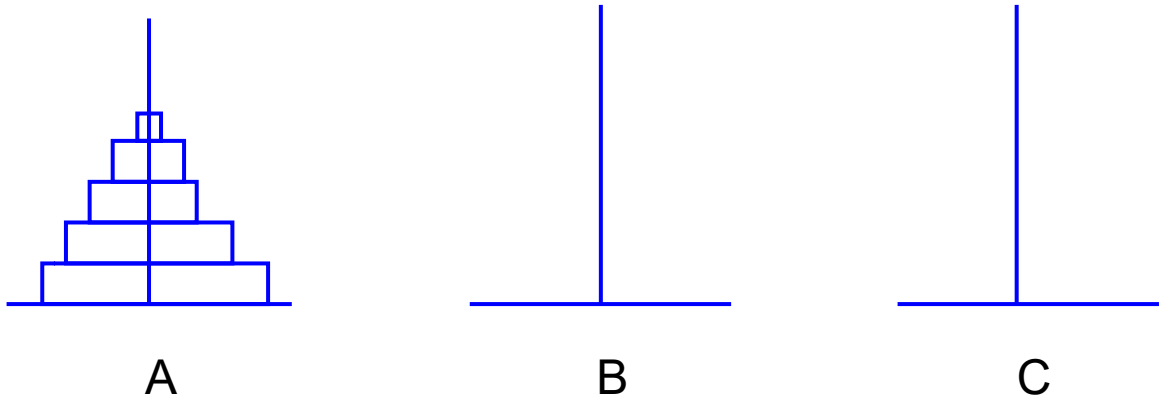
举例：汉诺塔问题

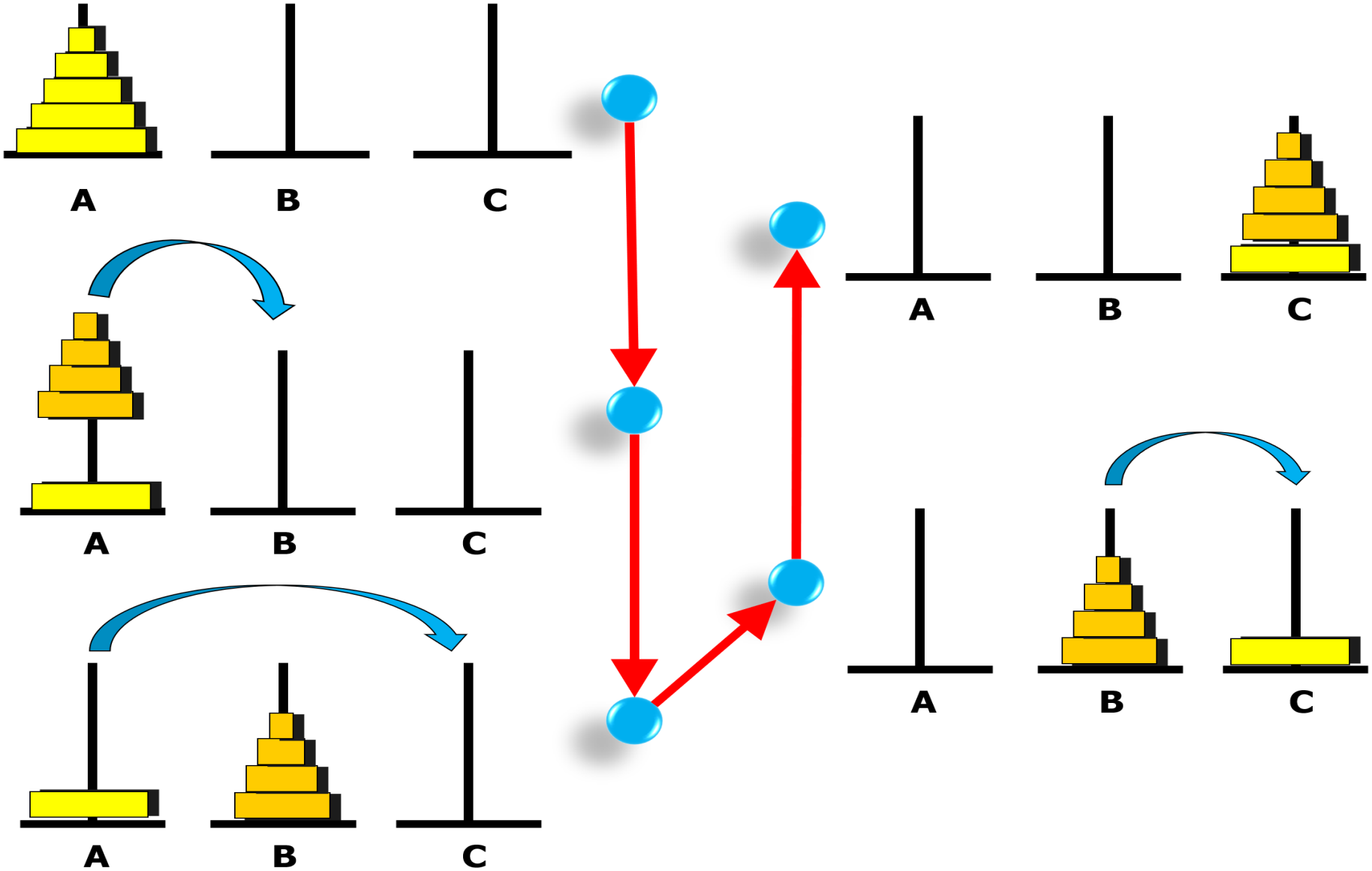


- 有三根针 A、B、C；
- A 针上有 N 个大小不同的盘子，大的在下，小的在上；
- 把这 N 个盘子从 A 针移到 C 针，在移动过程中可以借助 B 针；

要求：(1) 每次只允许移动一个盘，

(2) 在移动过程中在三根针上的盘子都要保持大盘在下，小盘在上。




A
B
C
A
B
C
A
B
C
A
B
C
A
B
C

举例：汉诺塔问题

分析：该问题可分解为下面三个步骤：

(1) 将 A 上 n-1 个盘子移到 B 针上（借助 C 针）；

(2) 把 A 针上剩下的一个盘子移到 C 针上;

(3) 将 n-1 个盘子从 B 针移到 C 针上（借助 A 针）；

上面三个步骤包含两种操作：

① 将多个盘子从一个针移到另一个针，这是一个递归过程，我们用 hanoi 函数实现

② 将 1 个盘子从一个针上移到另一针上，该过程用 move 函数实现

ex04_hanoi.cpp

形参的缺省值

- 函数在定义时，可以预先给形参设置一个值（即缺省值），函数被调用时，如给定实参，则采用实参值，否则就采用这个预先给定的缺省值
- 好处：调用时可以不提供或提供部分实参

```
int add(int x = 5, int y = 6)
{ return x + y; }

int main()
{
    add(10,20);  // 10 + 20
    add();       // 5 + 6
    add(10);     // 10 + 6
}
```

如何给形参设定缺省值

给多个形参设置缺省值的注意点

- 如果有多个形参，可以给部分形参设置缺省值
- 按 **从右到左** 顺序设置，即带缺省值的形参，其右边不能有不带缺省值的形参
- 调用函数时，实参与形参的配对按从左到右的顺序进行

```
int add(int x, int y = 5, int z = 6);  // OK
int add(int x = 1, int y = 5, int z);  // ERROR!
int add(int x = 1, int y, int z = 6);  // ERROR!
```

设置缺省值的位置

- 缺省值可以在定义是设置，也可以在声明时设置，要求：

  (1) 同一作用域 中，哪个在前就由哪个设置，后面的不能再设置

  （若先声明，后定义，则声明时设定，定义时就不能再设定，反之亦然）

  (2) 不同作用域 中的声明，可设定不同的缺省值

```
int add(int x = 5, int y = 6); // 设置缺省值
int main()
{
  add();  // 函数调用在函数定义之前
}
int add(int x, int y) // 这里不能设置缺省值
{  return x + y;   }
```

```
int add(int x = 5, int y = 6) // 设置缺省值
{   return  x + y;   }

int main()
{
  add();   // 函数调用在函数定义之后
}
```

ex04_DefaultValue_01/02/03.cpp

函数重载

C++ 允许功能相近的函数在相同的作用域内采用相同的函数名，从而形成重载

什么是函数重载

两个以上的函数，具有 **相同的函数名**，但 **形参不同**（个数或类型不同），调用时，编译器会根据实参和形参的最佳匹配，自动确定调用哪一个函数，这就是函数重载

```
int add(int x, int y)  // 整数的加法
{  return x + y;  }

double add(double x, double y) // 浮点数的加法
{  return x + y;  }
```

ex04_overload.cpp

函数重载（续）

注意：函数重载与返回值类型无关，即如果函数名相同，形参个数与类型也相同，则无论函数返回值的类型是否相同，编译时会认为是语法错误！

- 不要将功能不同的函数定义为重载函数！
- 在使用带有默认形参的重载函数时，要注意防止二义性！

```
int add(int x, int y = 1);
int add(int x);
```

```
add(10);  // ???
```

4 预编译与多文件结构

- 头文件与宏定义
- 条件编译
- 多文件结构
- 外部变量，外部函数
- 库函数

编译预处理

- **编译预处理**也称**预编译**，编译预处理是指编译器在实际编译之前所进行的一些预处理
- 编译预处理负责读取源程序，对其中的编译预处理指令和特殊符号进行处理，生成新的源代码，然后传递给编译器进行编译

- 编译预处理主要包含以下三个功能：文件导入 (如导入头文件)，宏定义和条件编译
- 编译预处理指令以 "#" 开头

编译预处理：导入头文件

```
#include <头文件名>
    // 按标准方式导入头文件，即在系统目录中寻找指定的文件
#include “头文件名”
    // 先在当前目录中寻找，然后再按标准方式搜索
```

常用头文件

iostream, iomanip, cmath, ctime, cstdlib

cstring, cctype, cstdio, vector, string, fstream, ...

编译预处理：宏定义

宏定义

```
#define PI 3.14159  // 定义宏
#undef PI  // 删除由 #define 定义的宏
```

在很多情况下可以由 const 实现该功能

#define 还可以用来定义带参数的宏，但也可以用内联函数取代

```
#define MAX(x,y) ( (x)>(y) ? (x) : (y) )
```

编译预处理：条件编译

```
#if 常量表达式
    程序正文     // 当“常量表达式”非零时编译
#endif
```

```
#if 常量表达式
    程序正文     // 当“常量表达式”非零时编译
#else
    程序正文     // 否则编译这段程序
#endif
```

条件编译

```
#if 常量表达式1
    程序正文     // 当 “常量表达式1” 非零时编译
#elif 常量表达式2
    程序正文     // 否则，当 “常量表达式2” 非零时编译
#elif 常量表达式3
    程序正文     // 否则，当 “常量表达式3” 非零时编译
... ...
#else
    程序正文     // 否则编译这段程序
#endif
```

条件编译

```
#ifdef 标识符
    程序正文      // 当 “标识符” 已由 #define 定义时编译
#else
    程序正文      // 否则编译这段程序
#endif
```

```
#ifndef 标识符
  程序正文   // 当 “标识符” 没有定义时编译
#else
  程序正文   // 否则编译这段程序
#endif
```

多文件结构

- 一个程序可以由多个文件组成，编译时可以使用工程/项目来组合。
- 若使用命令行编译，则可以同时编译。或分别编译生成目标文件，然后进行链接。

外部变量和外部函数

- 如果需要用到其它文件中定义的变量和函数，则需要用 **extern** 声明其为外部变量和外部函数。

extern 类型名 变量名;

extern 函数声明;

C++ 库函数

库函数分类

- 标准库函数：系统提供的头文件中定义的函数（cmath, ctime, ...）
  参见 https://www. cppreference.cn
- 非标准库函数：编译器商家或软件商提供的函数

- 使用库函数时要导入相应的头文件
- 充分使用库函数不仅可以大大减少编程工作量，还可以提高代码的可靠性和执行效率
  （为了提高效率，有些库函数会用汇编语言编写）

举例：定积分近似计算（随机取点）

例：计算定积分 $\int_0^{\frac{\pi}{2}} \sin(x)\,\mathrm{d}x$ 的近似值（随机取点）

$$S=\int_a^b f(x)\,\mathrm{d}x=\lim_{\Delta x_i\to 0}\sum \Delta x_i f(\xi_i)=\lim_{n\to\infty}\sum_{i=1}^{n} hf(\xi_i)=\lim_{n\to\infty}\frac{b-a}{n}\sum_{i=1}^{n} f(\xi_i)$$

$$\left(h=\frac{b-a}{n}\right)$$

ex04_Integral.cpp

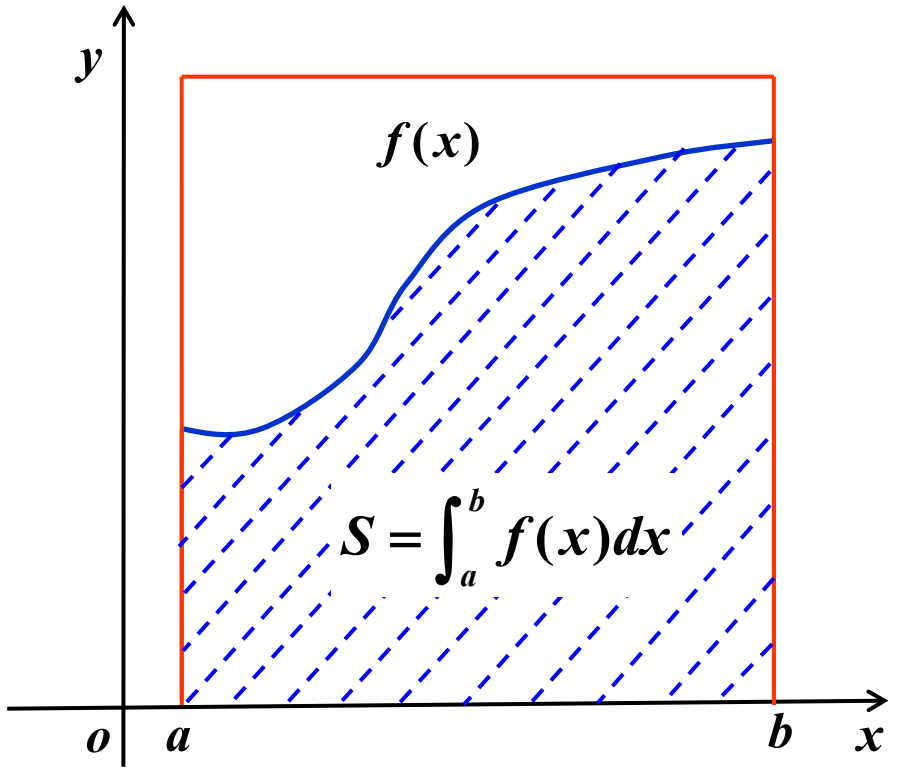
```
srand(time(0));
for(int i=0; i<n; i++)
{
    xi = a + h*(i+double(rand())/RAND_MAX);
    S += sin(xi);
}
```

举例：蒙特卡洛方法计算定积分

例：用蒙特卡洛（Monte Carlo）投点法计算定积分的近似值。 （留作练习）

- 选定一个简单大区域：要求覆盖阴影部分，且容易计算面积；
- 往这个大区域中随机投点（假定投 N 个点），统计落在阴影部分的点的个数（设为 M 个）
- 计算阴影部分面积的近似值

$$S = \frac{M}{N} \times \text{大区域的面积}$$



第四讲上机作业（一）

1、编写两个函数，分别计算两个正整数的最大公约数与最小公倍数，要求用辗转相除法计算最大公约数，最小公倍数则可以使用最大公约数，并在主函数中计算 **2012** 与 **1509** 的最大公约数与最小公倍数。

（函数名分别为 `gcd` 和 `lcm`，程序取名 **hw04_01.cpp**）

```
int gcd(int x, int y)
int lcm(int x, int y)
```

2、编写函数，判断一个正整数是否为素数，并在主函数中找出三位数中所有的素数，在屏幕上输出时每行输出 **8** 个。（程序取名 **hw04_02.cpp**）

```
bool is_prime(int n)
```

3、 编写函数，判断给定的年份是否为闰年，并在主函数中输出 **21** 世纪（**2000** 至 **2099** 年）中所有的闰年，每行输出 **5** 个。（程序取名 **hw04_03.cpp**）（闰年：能被 **400** 整除；或者能被 **4** 整除但不能被 **100** 整除）

```
bool is_leap_year(int year)
```

第四讲上机作业（一）

4、 编写程序，用 **while** 实现猜数游戏
（可在示例程序 `ex04_game.cpp` 上修改，程序取名 **hw04_04.cpp**）

5、**Emirp 数**：如果一个素数反转后仍然是素数，则称这个素数为 **emirp** 数。
如 **13** 是素数，反转后 **31** 也是素数，故 **13** 和 **31** 都是 **emirp** 数。
编写程序，输出前 **100** 个 **emirp** 数，每行输出 **5** 个。（程序取名 **hw04_05.cpp**）
要求：先编写两个函数：`is_prime` 和 `reverse`，分别用于判断素数和计算反转数。

```
bool is_prime(int n)
int reverse(int n)
```

6、**梅森 (Mersenne) 素数**：如果一个素数可以写成 2^p-1 的形式，则称该素数为梅森素数。
编写程序，找出所有 $p<32$ 的梅森素数，并以如右形式输出
（程序取名 **hw04_06.cpp**）

```
2   3
3   7
5   31
...     ...
```

第四讲上机作业（二）

7、 3*n*+1 问题： 给定一个正整数 ***n***，不断按照下面的规律进行运算：如果当前数是偶数，则下一个数为当前数除以 2，如果当前数为奇数，则下一个数为当前数乘 3 加 1。 不断重复上述过程，直到当前数是 1 为止。这样形成的数列的长度称为数 ***n*** 的链数。

例如：从 3 开始，得到的数列为：**3, 10, 5, 16, 8, 4, 2, 1**，所以 3 的链数为 **8** 。

(a) 编写一个函数（函数名 `num_chain` ），使用循环结构计算给定的正整数的链数；

(b) 找出 **[90, 100]** 中，链数最大的那个数。（程序取名 **hw04_07.cpp**）

```
int num_chain(int n)  // 使用循环计算链数
```

8、使用递归方法实现上面问题中的函数 `num_chain` （主程序取名 hw04_08.cpp）

9、编写函数，用递归方法计算 Fibonacci 数：

$$f(1)=1,\quad f(2)=1,\quad f(n)=f(n-1)+f(n-2),\quad n=2,3,...$$

并在主函数中计算第 40 个 Fibonacci 数。（程序取名 **hw04_09.cpp**）

```
long fibo(int n)
```

第四讲上机作业（二）

10、给定一个正整数，使用递归方法找出其所有的素数因子（在屏幕上输出）

（程序取名 **hw04_10.cpp** ）

```
void prime_factor(int n)
```

例如，**84** 的所有素数因子为 **2，2，3，7**

11、蒙特卡洛（**Monte Carlo**）投点法计算定积分 $\int_0^{\frac{\pi}{2}} \sin(x)\,dx$ 的近似值。（程序取名 **hw04_11.cpp**）

12、（可选题）试统计汉诺塔问题总的移动步数。（程序取名 **hw04_12.cpp**）