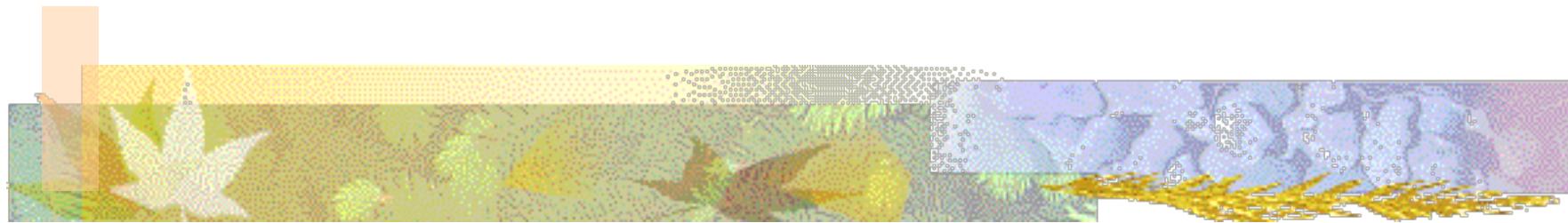




规格严格 功夫到家



第7章 函数



哈尔滨工业大学

计算机科学与技术学院

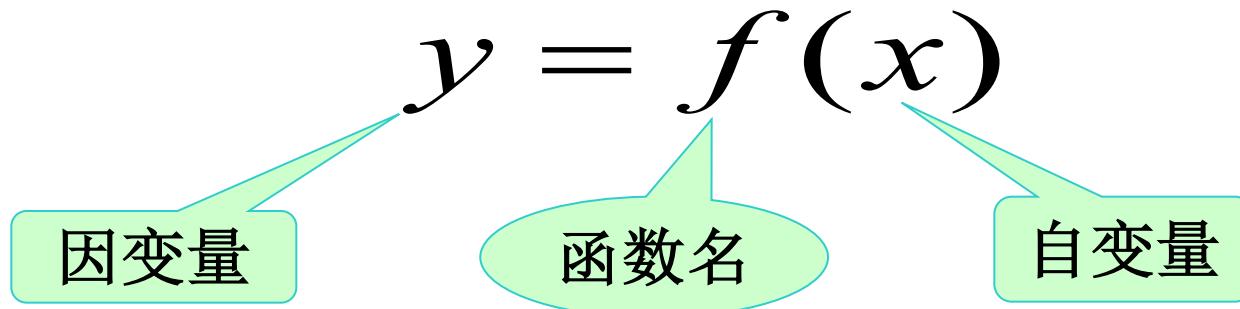
苏小红

sxh@hit.edu.cn

本章学习内容

- ❖ 函数定义、函数调用、函数原型、函数的参数传递与返回值
- ❖ 递归函数和函数的递归调用
- ❖ 函数封装，函数复用，函数设计的基本原则，程序的健壮性
- ❖ 变量的作用域与存储类型，全局变量、自动变量、静态变量、寄存器变量
- ❖ “自顶向下、逐步求精”的模块化程序设计方法

数学中的函数



程序设计中的函数

- 程序设计中的函数不局限于计算
 - 计算类，如打印阶乘表的程序……
 - 判断推理类，如排序、查找……

问题的提出

- 读多少行的程序能让你不头疼?
- 假如系统提供的函数**printf()**由10行代码替换，那么你编过的程序会成什么样子?
 - 实际上一个**printf()**有上千行代码
- **main()**中能放多少行代码?
- 如果所有代码都在**main()**中，怎么团队合作?
- 如果代码都在一个文件中，怎么团队合作???



问题的提出

■ 《三国演义》中有这样一段描写：

- 魏问曰：“孔明寝食及事之烦简若何？”使者曰：“丞相夙兴夜寐，罚二十以上皆亲览焉。所啖之食，日不过数升。”懿顾谓诸将曰：“孔明食少事烦，其能久乎？”
 - 此话音落不久，诸葛亮果然病故于五丈原。
- “事无巨细”，“事必躬亲”
- 管理学的观点是极其排斥这种做法的，认为工作必须分工，各司其职
 - 其中的思想，在程序设计里也适用



7.1 分而治之与信息隐藏

■ 分而治之 (Divide and Conquer, Wirth, 1971)

- 函数把较大的任务分解成若干个较小的任务，并提炼出公用任务

■ 信息隐藏 (Information Hiding, Parnas, 1972)

- 设计得当的函数可把具体操作细节对外界隐藏起来，从而使整个程序结构清楚
- 使用函数时，不用知道函数内部是如何运作的，只按照我们的需要和它的参数形式调用它即可

程序设计的艺术

■ 算法设计艺术

- 程序的灵魂
- Donald E. Knuth,

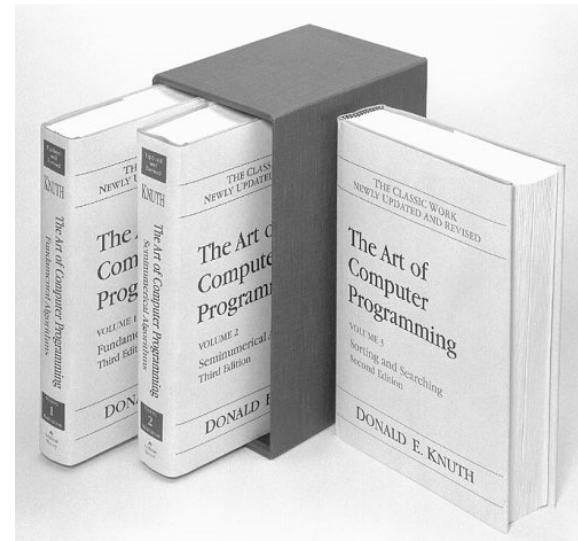
“The Art of Computer Programming” ,

清华大学出版社 (英) , 国防工业出版社 (中)



■ 结构设计艺术

- 程序的肉体
- 模块化 (Parnas, 1972)
 - 结构化 (Structural)
 - 面向对象 (Object-Oriented)
 - 面向组件 (Component-Oriented)
 - 面向智能体 (Agent-Oriented)
 -



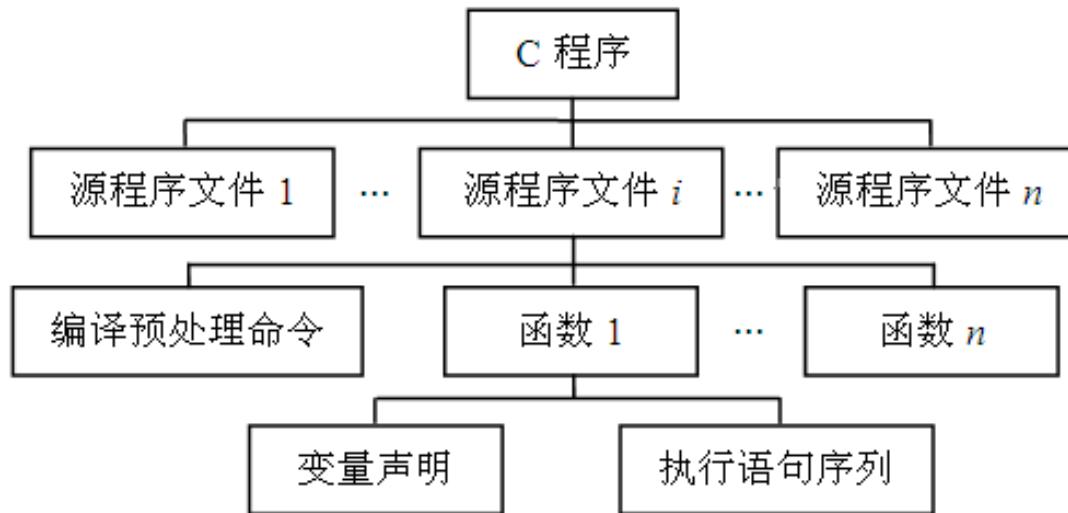
7.2 函数 (Function) 的定义

- 函数是C语言中模块化编程的最小单位
 - 可以把每个函数看作一个模块 (Module)
- 如把编程比做制造一台机器，函数就好比其零部件
 - 可将这些“零部件”单独设计、调试、测试好，用时拿出来装配，再总体调试。
 - 这些“零部件”可以是自己设计制造/别人设计制造/现成的标准产品



7.2 函数 (Function) 的定义

- 若干相关的函数可以合并成一个“模块”
- 一个C程序由一个或多个源程序文件组成
- 一个源程序文件由一个或多个函数组成



7.2.1 函数的分类

- 函数生来都是平等的，互相独立的，没有高低贵贱和从属之分
 - **main()**稍微特殊一点点
 - C程序的执行从**main**函数开始
 - 调用其他函数后流程回到**main**函数
 - 在**main**函数中结束整个程序运行

7.2.1 函数的分类

■ 标准库函数

— ANSI/ISO C 定义的标准库函数

- 符合标准的C语言编译器必须提供这些函数
- 函数的行为也要符合ANSI/ISO C的定义

— 第三方库函数

- 由其他厂商自行开发的C语言函数库
- 不在标准范围内，能扩充C语言的功能（图形、网络、数据库等）

■ 自定义函数

— 自己定义的函数

- 包装后，也可成为函数库，供别人使用

7.2.2 函数的定义 (Function Definition)

返回值
类型

函数名标识符
, 说明运算规
则

参数表相当于
运算的操作数

类型 函数名(类型 参数1, 类型 参数2, ……)

{

声明语句序列

可执行语句序列

return 表达式;

}

函数出口

返回运算的结
果

7.2.2 函数的定义 (Function Definition)

参数表里的变量（叫形式参数，
Formal Parameter）也是内部变量

类型 函数名(类型 参数1, 类型 参数2, ……)

{

声明语句序列
可执行语句序列
return 表达式；

函数体的定界
符

}

函数体

7.2.2 函数的定义 (Function Definition)

函数无返回值，用
void 定义返回值类型

用**void** 定义参数
，表示没有参数

void 函数名 (**void**)
{

 声明语句序列

 可执行语句序列

return;

}

return语句后无
需任何表达式

```
void main()
{
    .....
}
```

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

【例7.1a】计算整数n的阶乘n!

返回值类型

函数名说明
函数的功能

形参表，函
数入口

*

long

Fact

(int n)

/* 函数定

{

int i;

long result = 1;

for (i=2; i<=n; i++)

{

result *= i;

}

return result;

}

函数内部可以定义
只能自己使用的变
量，称内部变量

返回值作为函数
调用表达式的值

7.3 向函数传递值和从函数返回值

函数名(表达式1, 表达式2,);

■ 实际参数(Actual Argument)

— 函数调用(Founction Call)时提供的表达式

■ 有返回值时

— 放到一个数值表达式中

`c = max(a, b);`

— 作为另一个函数调用的参数

`c = max(max(a, b), c);`

`printf("%d\n", max(a, b));`

返回值 = 函数名(实参表列);

■ 无返回值时

— 函数调用表达式

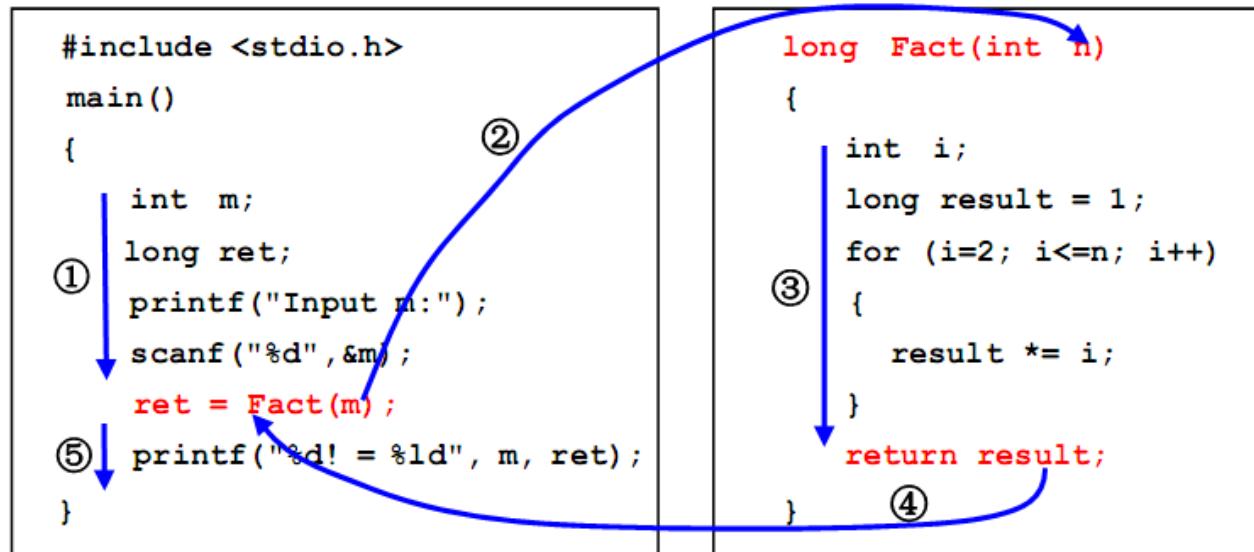
`display(a, b);`

函数名(实参表列);

函数的参数传递

实参和形参必须匹配

— 数目一致，类型一一对应（否则会发生自动类型转换）



② m 3 ret ?

n 3 result 1

③ m 3 ret ?

n 3 result 6

④ m 3 ret 6

n 3 result 6

【例7.1】

7.3.2 函数原型 (Function Prototype)

- 在调用函数前先声明其返回值类型、函数名和参数
- 函数原型有助于编译器对函数参数类型的匹配检查

```
#include <stdio.h>
long Fact(int n);
main()
{
    int m;
    long ret;
    printf("Input m:");
    scanf("%d", &m);
    ret = Fact(m);
    printf("%d! = %ld\n", m, ret);
}
long Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * Fact(n - 1);
}
```

末尾有一个分号，
声明时不要省略形参和返回值的类型

② m 3 ret ?

n 3 result 1

③ m 3 ret ?

n 3 result 6

④ m 3 ret 6

n 3 result 6

【例7.1】

函数定义与函数声明的区别

函数定义

- 指函数功能的确立
- 指定函数名、函数类型、形参及类型、函数体等
- 是完整独立的单位

函数声明

- 是对函数名、返回值类型、形参类型的说明
- 不包括函数体
- 是一条语句，以分号结束，只起一个声明作用

7.3.3 函数封装与防御性程序设计

- 函数封装（Encapsulation）使得外界对函数的影响仅限于入口参数，而函数对外界的影响仅限于一个返回值和数组、指针类型的参数

```
#include <stdio.h>
long Fact(int n);
int main()
{
    int m;
    long ret;
    printf("Input m:");
    scanf("%d", &m);
    ret = Fact(m);
    printf("%d! = %ld\n", m, ret);
    return 0;
}
```

```
long Fact(int n)
{
    int i;
    long result = 1;
    for (i=2; i<=n; i++)
    {
        result *= i;
    }
    return result;
}
```

【例7.1】

传入负数实参
会怎样？

Why?

Input m:-10✓

-10! = 1



防御性程序设计 (*Defensive Programming*)

【例7.2】计算整数n的阶乘n!

- 如何使函数具有遇到不正确使用或非法数据输入时避免出错的能力，增强程序的健壮性?
 - 在函数的入口处，检查输入参数的合法性

```
/* 函数功能：用迭代法计算 n! */  
long Fact(int n)  
{  
    int i;  
    long result = 1;  
    if (n < 0)  
    {  
        printf("Input data error!\n");  
    }  
    else  
    {  
        for (i=2; i<=n; i++)  
            result *= i;  
        return result;  
    }  
}
```

```
Input m:-10↙  
Input data error!  
-10! = 18
```

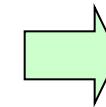


防御性程序设计 (*Defensive Programming*)

【例7.2】计算整数n的阶乘n!

- 如何使函数具有遇到不正确使用或非法数据输入时避免出错的能力，增强程序的健壮性?
 - 在函数的入口处，检查输入参数的合法性

```
/* 函数功能：用迭代法计算 n! */  
long Fact(int n)  
{  
    int i;  
    long result = 1;  
    if (n < 0)  
    {  
        printf("Input data error!\n");  
    }  
    else  
    {  
        for (i=2; i<=n; i++)  
            result *= i;  
        return result;  
    }  
}
```



```
long Fact(int n)  
{  
    int i;  
    long result = 1;  
    if (n < 0)  
    {  
        return -1;  
    }  
    else  
    {  
        for (i=2; i<=n; i++)  
            result *= i;  
        return result;  
    }  
}
```



防御性程序设计 (*Defensive Programming*)

【例7.3】计算整数n的阶乘n!

■ 主函数如何修改?

— 增加对函数返回值的检验



```
#include <stdio.h>
long Fact(int n);
int main()
{
    int m;
    long ret;
    printf("Input m:");
    scanf("%d", &m);
    ret = Fact(m);
    if (ret == -1)          /* 增加对函数返回值的检验 */
        printf("Input data error!\n");
    else
        printf("%d! = %ld\n", m, ret);
    return 0;
}
```

```
long Fact(int n)
{
    int i;
    long result = 1;
    if (n < 0)
    {
        return -1;
    }
    else
    {
        for (i=2; i<=n; i++)
            result *= i;
        return result;
    }
}
```

防御性程序设计 (*Defensive Programming*)

【例7.3】计算整数n的阶乘n!



■ 传入负数的实参时Fact()会返回-1吗?

— 存在死代码的原因何在?

```
#include <stdio.h>
unsigned long Fact(unsigned int n);

int main()
{
    int m;
    long ret;
    printf("Input m:");
    scanf("%d", &m);
    ret = Fact(m);
    if (ret == -1)          /* 增加对函数返回值的检验 */
        printf("Input data error!\n");
    else
        printf("%d! = %ld\n", m, ret);
    return 0;
}
```

```
unsigned long Fact(unsigned int n)
{
    unsigned int i;
    unsigned long result = 1;
    if (n < 0)
    {
        return -1;
    }
    else
    {
        for (i=2; i<=n; i++)
            result *= i;
        return result;
    }
}
```

warning: comparison of unsigned expression < 0 is always false

防御性程序设计 (*Defensive Programming*)

【例7.2】计算整数n的阶乘n!

- 如何修改程序去除冗余代码?
- 如何保证不会传入负数实参?



```
#include <stdio.h>
unsigned long Fact(unsigned int n);

int main()
{
    int m;
    do{
        printf("Input m(m>0) :");
        scanf("%d", &m);
    }while (m<0);
    printf("%d! = %lu\n", m, Fact(m));
    return 0;
}
```

```
unsigned long Fact(unsigned int n)
{
    unsigned int i;
    unsigned long result = 1;
    for (i=2; i<=n; i++)
        result *= i;
    return result;
}
```

```
Input m(m>0) :-1 ✗
Input m(m>0) :10 ✗
10! = 3628800
```

【例7.4】编写计算组合数的程序

$$C_m^k = \frac{m!}{k!(m-k)!}$$

函数复用

```
#include <stdio.h>
unsigned long Fact(unsigned int n);
int main()
{
    int m, k;
    double p;
    do{
        printf("Input m,k (m>=k>0) : ");
        scanf("%d,%d", &m, &k);
    }while (m<k || m<0 || k<0);
    p = (double)Fact(m) / (Fact(k) * Fact(m-k));
    printf("p = %.0f\n", p);
    return 0;
}
```

```
unsigned long Fact(unsigned int n)
{
    unsigned int i;
    unsigned long result = 1;
    for (i=2; i<=n; i++)
        result *= i;
    return result;
}
```

- ① Input m,k (m>=k>0) : 3,2↙
p = 3
- ② Input m,k (m>=k>0) : 2,3↙
Input m,k (m>=k>0) : 3,3↙
p = 1
- ③ Input m,k (m>=k>0) : -2,-4↙
Input m,k (m>=k>0) : 4,2↙
p = 6

7.3.4 函数设计的基本原则

信息隐藏

入口参数有效性检查
敏感操作前的检查
调用成功与否的检查

1

函数规模
要小

2

函数功能
要单一

3

函数接口
定义要清楚

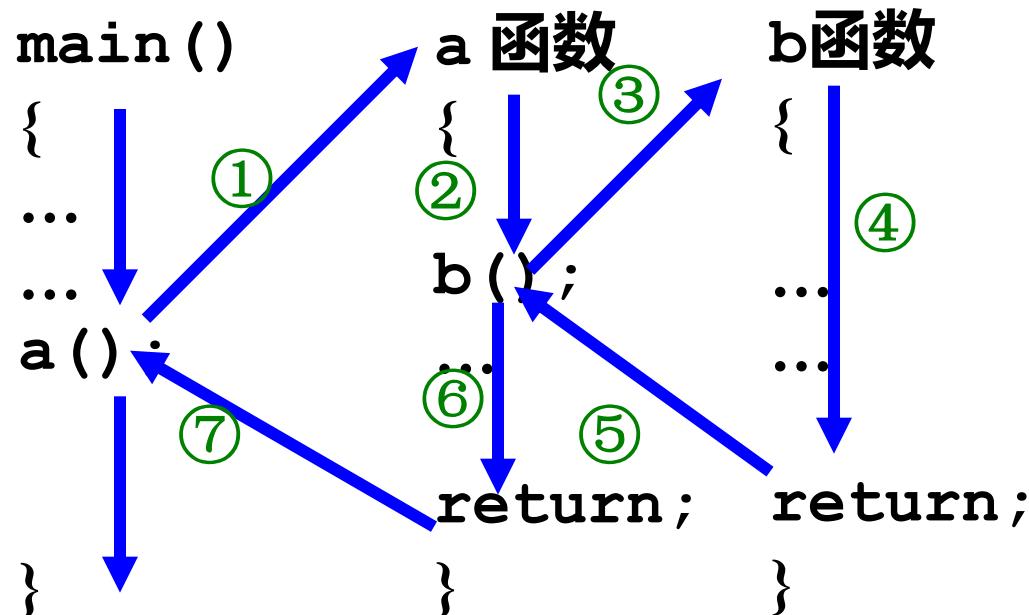
函数的嵌套调用

■ 嵌套调用

- 在调用一个函数的过程中，又调用另一个函数

■ C语言规定函数不能嵌套定义，但可以嵌套调用

- 函数是相互平行的



7.4 函数的递归调用和递归函数

- 如果一个对象部分地由它自己组成或按它自己定义，则我们称它是递归（Recursive）的。
 - 生活中，字典就是一个递归问题的典型实例
 - 字典中的任何一个词汇都是由“其他词汇”解释或定义的，但是“其他词汇”在被定义或解释时又会间接或直接地用到那些由它们定义的词
- 递归方法的基本原理
 - 将复杂问题逐步化简，最终转化为一个最简单的问题
 - 最简单问题的解决就意味着整个问题的解决

递归函数 (*Recursive Function*)

【例7.5】计算 $n! = n * (n-1) * (n-2) * \dots * 1$

```
long fact(int n)
{
    if (n < 0)
        return -1;
    else if (n == 0 || n
        return 1;
    else
        return n * fact(n-1);
}
```

$$n! = \begin{cases} 1 & n = 0, 1 \\ n \times (n-1)! & n \geq 2 \end{cases}$$

函数直接或间接调用
自己，称为递归调用
(Recursive Call)



递归函数 (*Recursive Function*)

【例7.5】计算 $n! = n * (n-1) * (n-2) * \dots * 1$

```
unsigned long fact(unsigned int n)
{
    if (n == 0 || n == 1)
        return 1;
    else
        return n * fact(n-1);
}
```

基线情况
(base case)

一般情况
(general case)

无需考虑
 $n < 0$ 了

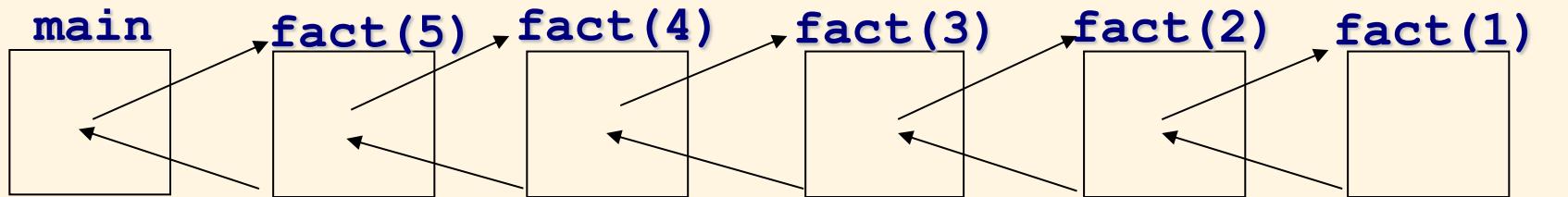
$$n! = \begin{cases} 1 & n = 0, 1 \\ n \times (n-1)! & n \geq 2 \end{cases}$$



递归函数 (*Recursive Function*)

- 递归调用应该能够在有限次数内终止递归
 - 递归调用若不加以限制，将无限循环调用
 - 必须在函数内部加控制语句，仅当满足一定条件时，递归终止，称为条件递归
- 任何一个递归调用程序必须包括两部分
 - 递归循环继续的过程
 - 递归调用结束的过程

```
if (递归终止条件成立)
    return 递归公式的初值;
else
    return 递归函数调用返回的结果值
;
```



$$\text{fact}(5) = 5 * \text{fact}(4) = 120$$

$$\text{fact}(4) = 4 * \text{fact}(3) = 24$$

$$\text{fact}(3) = 3 * \text{fact}(2) = 6$$

$$\text{fact}(2) = 2 * \text{fact}(1) = 2$$

$$\text{fact}(1) = 1$$

$$n! = n \times (n-1)!$$

$$(n-1)! = (n-1) \times (n-2)!$$

$$(n-2)!$$

$$(n-3)!$$

回推过程

递推过程

每个递归函数必须至少有一个**基线条件**
一般情况必须最终能简化为基线条件

:

$$\begin{aligned}
 5! & \\
 4! &= 4 \times 3! \\
 3! &= 3 \times 2! \\
 2! &= 2 \times 1! \\
 1! &= 1
 \end{aligned}$$

递归层数太多易导致栈空间溢出后果很严重，程序被异常中止

递归与迭代

- 用迭代（即循环）方法编写的阶乘函数

```
unsigned long Fact(unsigned int n)
```

```
{
```

```
    unsigned long result = 1;
```

```
    unsigned int i;
```

```
    for (i = 1; i <= n; i++)
```

```
        result *= i;
```

```
    return result;
```

```
}
```

- 递归程序遵循了数学中对阶乘的定义

- 因此递归方法编写程序具有更清晰、可读性更好的优点

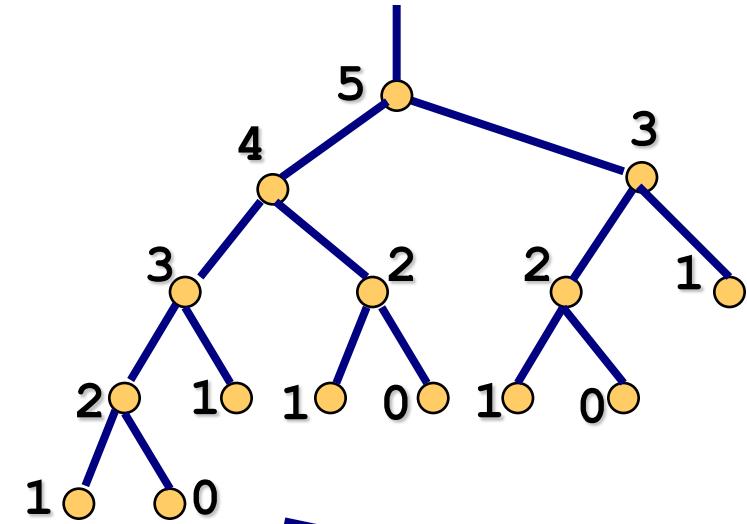
递归与迭代

【例7.6】计算Fibonacci数列

1, 1, 2, 3, 5, 8,

$$fib(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ fib(n-1) + fib(n-2) & n > 1 \end{cases}$$

```
long Fib(int n)
{
    long f;
    if (n == 0) f = 0;
    else if (n == 1) f = 1;
    else f = Fib(n-1) + Fib(n-2);
    return f;
}
```



计算fib(5)共需
15次fib调用

递归与迭代

优点：

- 从编程角度来看，比较直观、精炼，逻辑清楚
- 符合人的思维习惯，逼近数学公式的表示
- 尤其适合非数值计算领域
 - hanoi塔，骑士游历、八皇后问题（回溯法）

缺点：

- 增加了函数调用的开销，每次调用都需要进行参数传递、现场保护等
- 耗费更多的时间和栈空间
- 应尽量用迭代形式替代递归形式

7.5 变量的作用域和存储类型

7.5.1 变量的作用域（Scope）

- 指在源程序中定义变量的位置及其能被读写访问的范围
- 分为
 - 局部变量（Local Variable）
 - 全局变量（Global Variable）

局部变量 (*Local Variable*)

■ 在语句块内定义的变量

- 形参也是局部变量

■ 特点

- 生存期是该语句块，进入语句块时获得内存，仅能由语句块内语句访问，退出语句块时释放内存，不再有效
- 定义时不会自动初始化，除非程序员指定初值
- 并列语句块各自定义的同名变量互不干扰

■ 形参和实参可以同名

全局变量 (*Global Variable*)

在所有函数之外定义的变量

- 生存期是整个程序，从程序运行起占据内存，程序运行过程中可随时访问，程序退出时释放内存
- 有效范围是从定义变量的位置开始到本程序结束

例如：

```
int a,b;  
main()  
{  
    .....  
}  
int x,y;  
f()  
{  
    .....  
}
```

全局变量a、b的作用范围

全局变量x, y 的作用范围

全局变量 (Global Variable)

【例7.7】打印计算
Fibonacci数列每一
项时所需的递归调
用次数

```
Input n:10
fib(1)=1, count=1
fib(2)=1, count=3
fib(3)=2, count=5
fib(4)=3, count=9
fib(5)=5, count=15
fib(6)=8, count=25
fib(7)=13, count=41
fib(8)=21, count=67
fib(9)=34, count=109
fib(10)=55, count=177
```

```
1 #include <stdio.h>
2 long Fib(int a);
3 int count;
4 int main()
5 {
6     int n, i, x;
7     printf("Input n:");
8     scanf("%d", &n);
9     for (i=1; i<=n; i++)
10    {
11        count = 0;
12        x = Fib(i);
13        printf("Fib(%d)=%d, count=%d\n", i, x, count);
14    }
15    return 0;
16 }
/* 函数功能：用递归法计算Fibonacci数列中的第n项的值 */
17 long Fib(int n)
18 {
19     long f;
20     count++;
21     if (n == 0) f = 0;
22     else if (n == 1) f = 1;
23     else f = Fib(n-1) + Fib(n-2);
24 }
```

全局变量

全局变量使函数间的数据交换更容易，更高效，
但建议尽量少用，因为谁都可改写它，所以很难
确定是谁改写了它

7.5.2 变量的存储类型 (Storage Class)

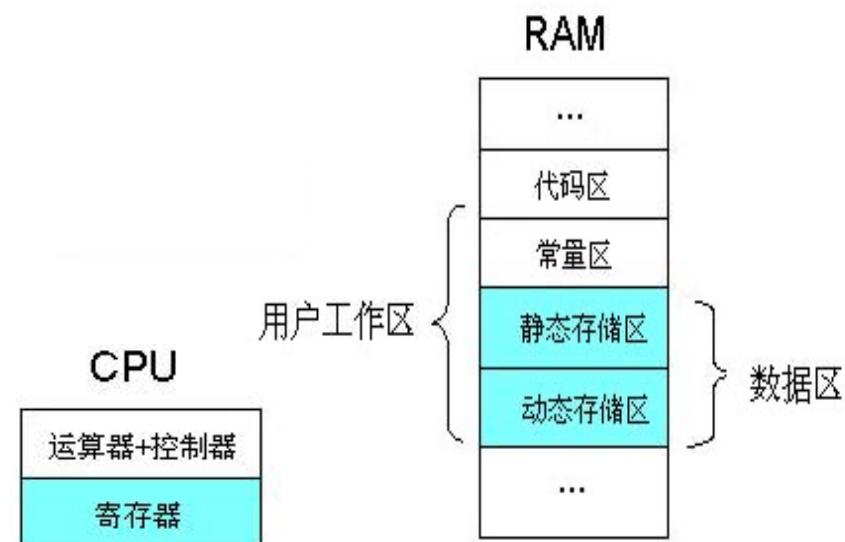
■ 指数据在内存中存储的方式

- 即编译器为变量分配内存的方式，它决定变量的生存期

存储类型 数据类型 变量名;

■ C程序的存储类别

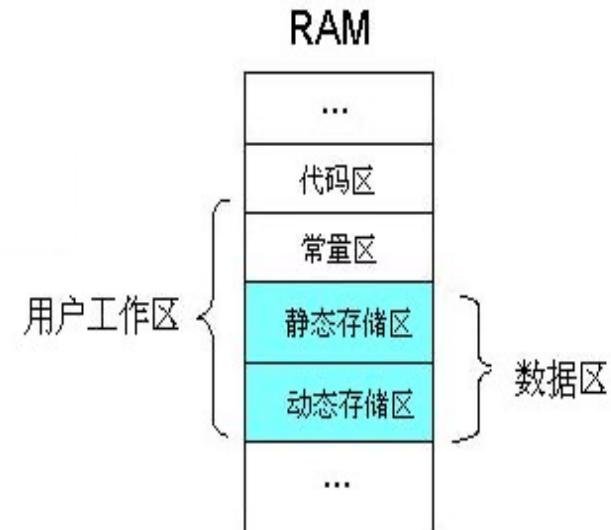
- *auto*型（自动变量）
- *static*型（静态变量）
- *extern*型（外部变量）
- *register*型（寄存器变量）



7.5.2 变量的存储类型 (Storage Class)

变量的生存期 (Lifetime)

决定何时“生”，何时“灭”



静态存储区中的变量：与程序“共存亡”

动态存储区中的变量：与程序块“共存亡”

寄存器中的变量：同动态存储区

自动变量和静态变量

auto 数据类型 变量名；

■ **auto**体现在

- 进入语句块时自动申请内存，退出时自动释放内存
- 动态局部变量，缺省的存储类型

■ 静态变量

static 数据类型 变量名；

- **static** storage class for **local** variables (declared **inside** a block or function) - the lifetime of the entire program
- 生存期为整个程序运行期间

自动变量和静态变量

【例7.9】

```
1 #include <stdio.h>
2 long Func(int n);
3 int main()
4 {
5     int i, n;
6     printf("Input n: ");
7     scanf("%d", &n);
8     for (i=1; i<=n; i++)
9     {
10         printf("%d! = %ld\n", i, Func(i));
11     }
12     return 0;
13 }
14 long Func(int n)
15 {
16     auto long p = 1; /* 定义自动变量 */
17     p = p * n;
18     return p;
19 }
```

自动变量不初始化时，值是随机值，静态局部变量和全局变量自动初始化为0值

/* 定义自动变量 */

6! = 0
7! = 7
8! = 8
9! = 9
10! = 10



自动变量和静态变量

【例7.10】利用静态变量计算整数n的阶乘n!

```
1 #include <stdio.h>
2 long Func(int n);
3 int main()
4 {
5     int i, n;
6     printf("Input n: ");
7     scanf("%d", &n);
8     for (i=1; i<=n; i++)
9     {
10         printf("%d! = %ld\n", i, Func(i));
11     }
12     return 0;
13 }
14 long Func(int n)
15 {
16     static long p = 1;          /* 定义静态变量 */
17     p = p * n;
18     return p;
19 }
```

静态变量仅初始化一次，变量的值可保存到下次进入函数，使函数具有记忆功能

6! = 720

7! = 5040

8! = 40320

9! = 362880

10! = 3628800

寄存器变量

寄存器

- CPU内部容量有限、但速度极快的存储器

register 类型名 变量名；



- 使用频率比较高的变量声明为**register**，可使程序更小、执行速度更快

- 现代编译器有能力自动把普通变量优化为寄存器变量， 并且可以忽略用户的指定
- 所以一般无需特别声明变量为**register**

7.5 变量的作用域和存储类型

全局变量

定义点之前
使用，需用
`extern` 声明

局部变量

静态外部变量（只限本文件使用）

外部变量（非静态外部变量允许其他文件引用）

静态局部变量（离开函数，值仍保留）

自动变量，（离开函数，值就消失）

寄存器变量（离开函数，值就消失）

7.6 模块化程序设计

■ 模块各司其职

- 每个模块只负责一件事情，它可以更专心
- 便于进行单个模块的设计、开发、调试、测试和维护等工作
- 一个模块一个模块地完成，最后再将它们集成

■ 开发人员各司其职

- 按模块分配任务，职责明确
- 并行开发，缩短开发时间

■ 什么时候需要模块化？

- 某一功能，如果重复实现3遍以上，即应考虑模块化，将它写成通用函数，并向小组成员发布

7.6 模块化程序设计



■ 模块化的优点——复用

- 构建新的软件系统可以不必每次从零做起，直接使用已有的经过反复验证的软构件，组装或加以合理修改后成为新的系统，提高软件生产率和程序质量
 - 在其他函数的基础上构造程序
- 拿来拿去主义，指尽可能复用其他人现成的模块
 - 不是人类懒惰的表现，而是智慧的表现
- 一般要靠日积月累才能建立可以被复用的软件库
 - 前期投入多，缺乏近期效益，大部分公司都注重近期效益，是为了生存，所以软件复用对公司来说不是最高优先级

7.6 模块化程序设计

■ 功能分解

- 自顶向下、逐步求精的过程

■ 模块分解的基本原则

- 保证模块的相对独立性——高聚合、低耦合
- 模块的实现细节对外不可见——信息隐藏
 - 外部：关心做什么；内部：关心怎么做

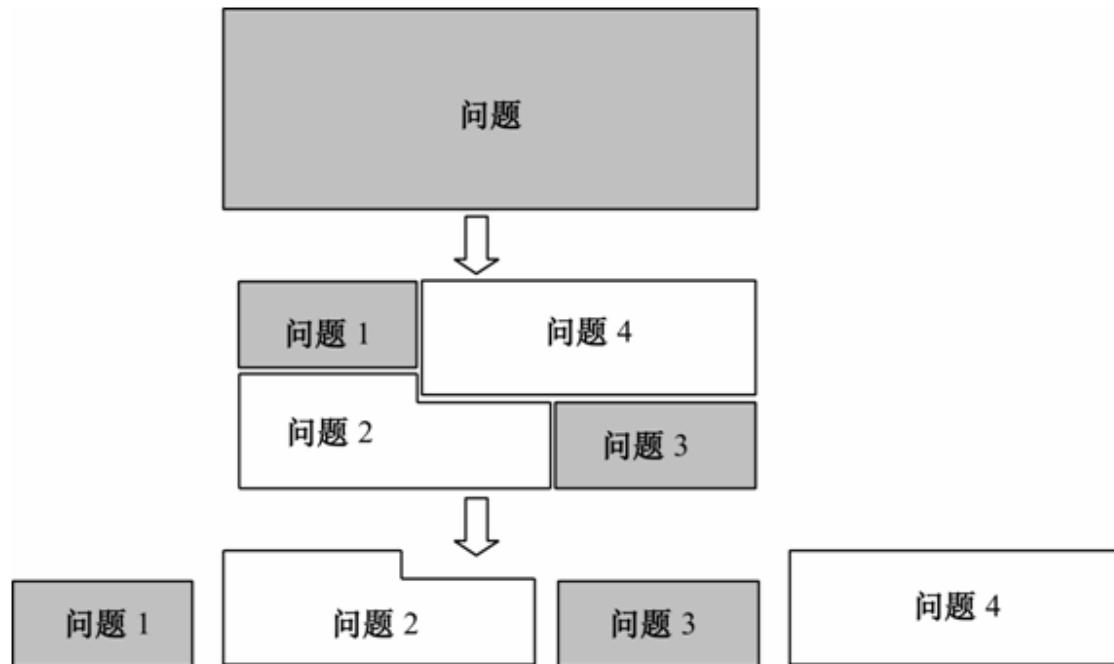
■ 设计好模块接口

- 接口指明列出一个模块的所有的与外部打交道的变量等
- 定义好后不要轻易改动
- 在模块开头（文件的开头）进行函数声明

7.6 模块化程序设计

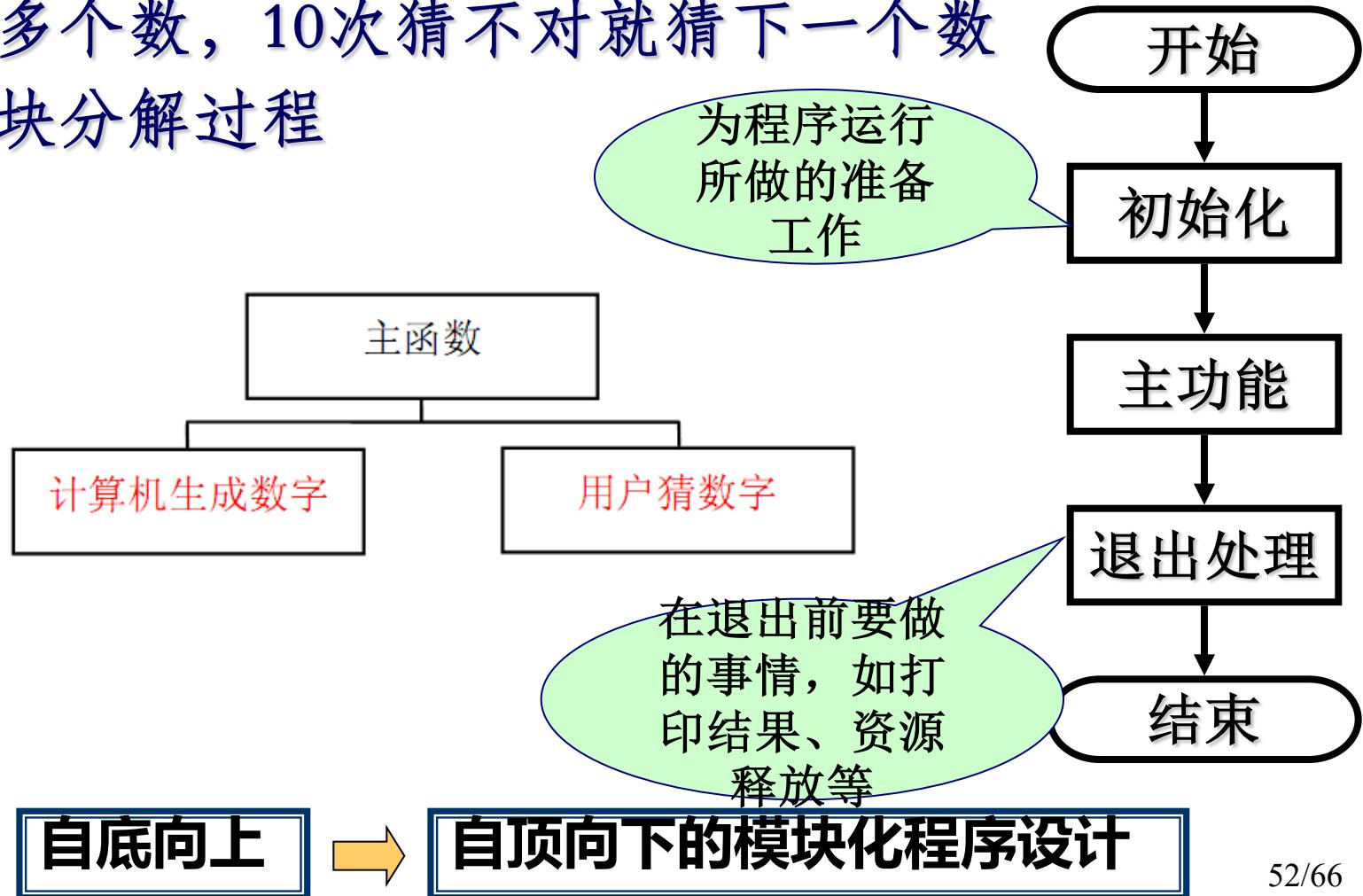
■ 逐步求精 (Stepwise Refinement)

- 由不断的自底向上修正所补充的自顶向下 (Top-down) 的程序设计方法

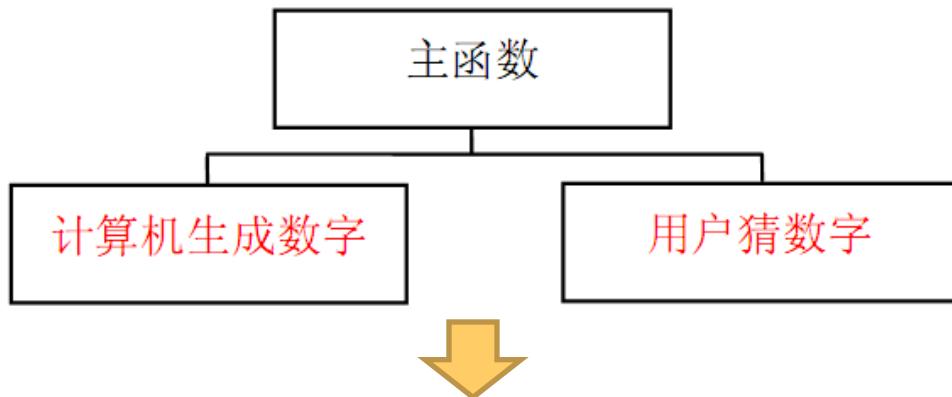


【例7.11】用函数完成猜数游戏

- 猜多个数，10次猜不对就猜下一个数
- 模块分解过程

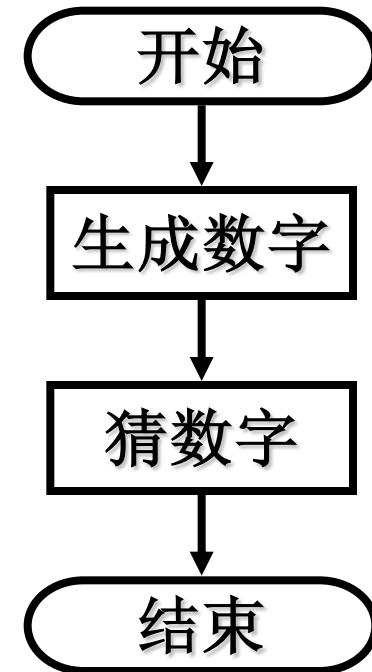


【例7.11】用函数完成猜数游戏

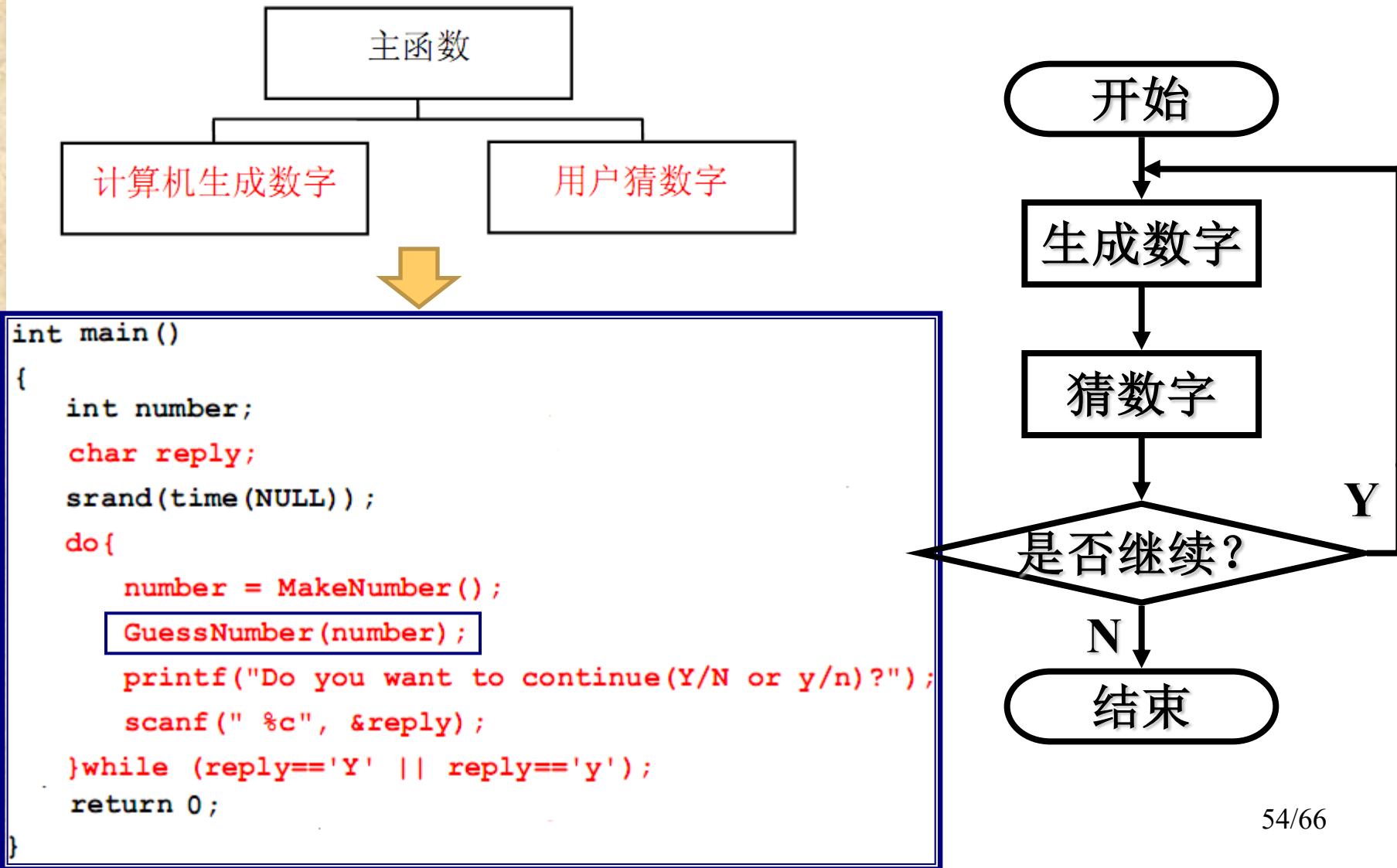


```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

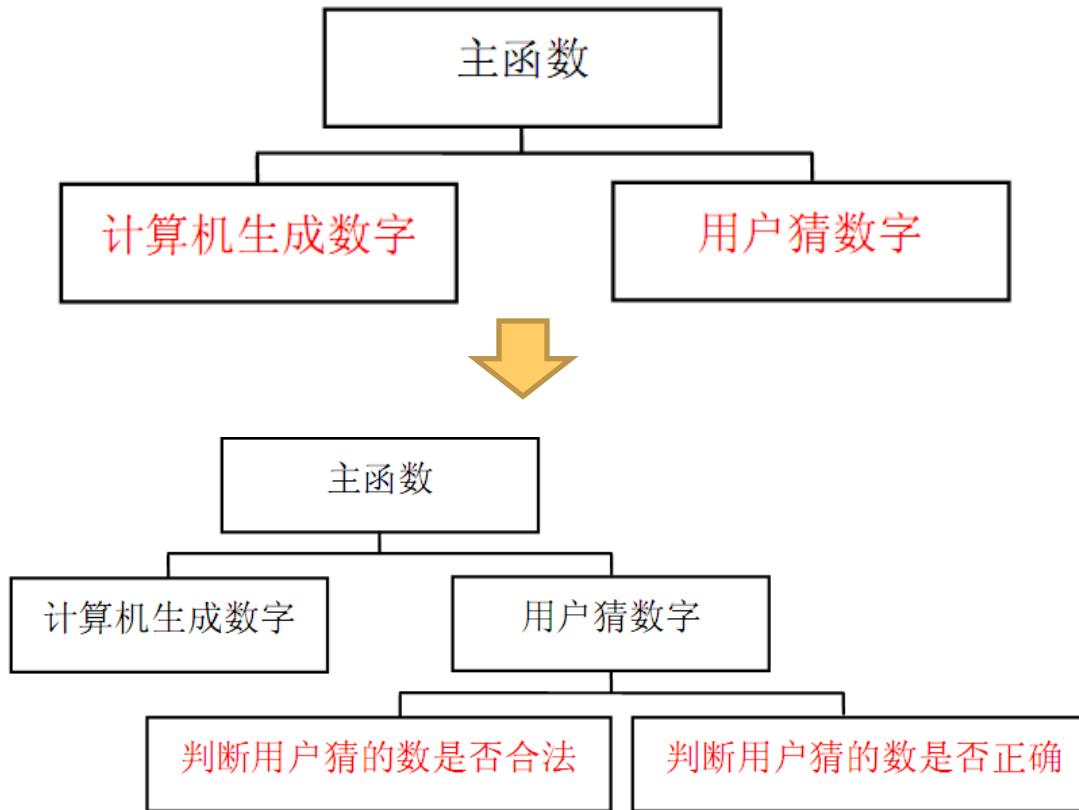
int main()
{
    int number;
    srand(time(NULL));
    number = MakeNumber();
    GuessNumber(number);
    return 0;
}
```



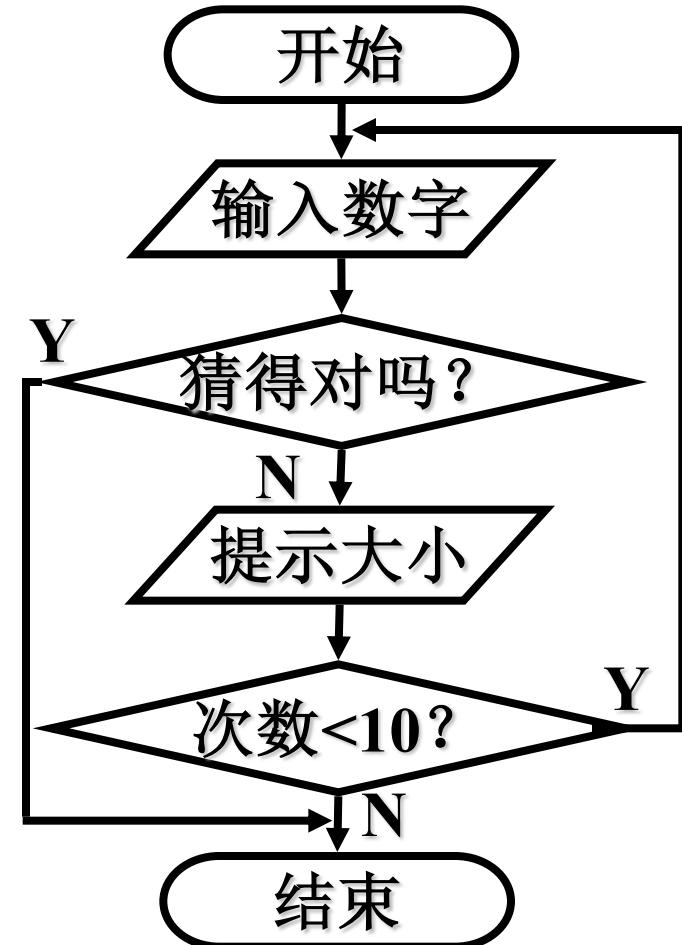
【例7.11】用函数完成猜数游戏



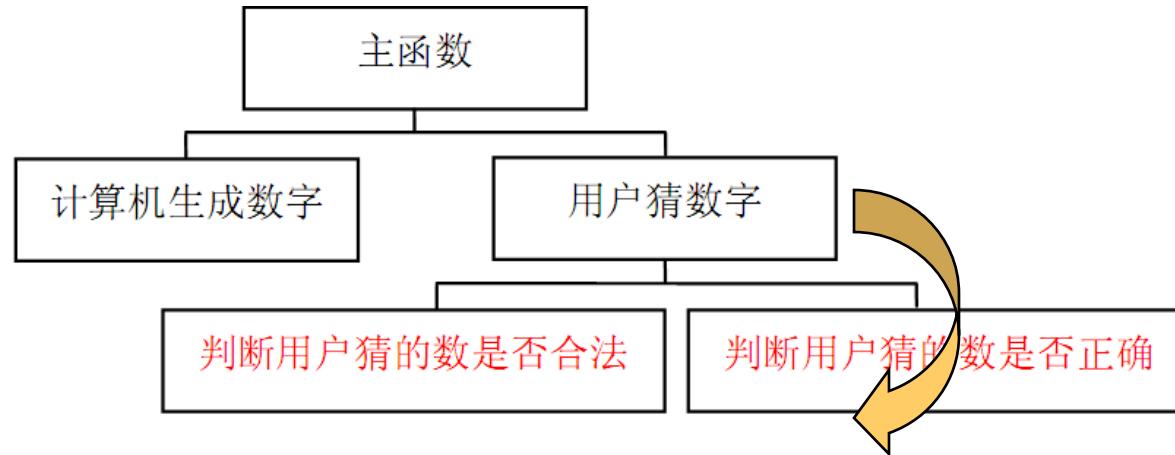
【例7.11】用函数完成猜数游戏



处理用户输入，判断是否有输入错误，
是否在合法的数值范围内

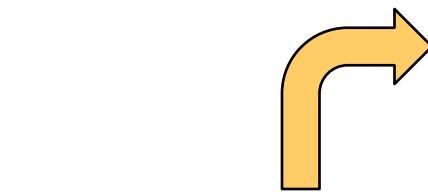


【例7.11】用函数完成猜数游戏



```
void GuessNumber(const int number){  
    // 记录用户猜测次数的计数器置初值为1;  
    do {  
        // 读入用户猜测的数字;  
        // 判断用户猜的数是否有输入错误，是否在合法的数值范围内，并进行错误处理;  
        // 记录用户猜测次数的计数器增1;  
        // 判断用户猜的数是大还是小，并输出相应的提示信息;  
        } while(未猜对并且猜测次数未超过MAX_TIMES次);  
        若猜对  
            输出"Congratulations! You're so cool!";  
        否则若超过MAX_TIMES次仍未猜对  
            输出"Mission failed after 10 attempts.";  
}
```

【例7.11】



```
void GuessNumber(const int number)
{
    记录用户猜测次数的计数器置
    do{
```

读入用户猜测的数字;

判断用户猜的数是否有输入错误，是否在合法的数值范围内，并进行错误处理；

记录用户猜测次数的计数器增1；

判断用户猜的数是大还是小，并输出相应的提示信息；

}while(未猜对并且猜测次数未超过MAX_TIMES次)；

若猜对

输出"Congratulations! You're so cool!"；

否则若超过MAX_TIMES次仍未猜对

输出"Mission failed after 10 attempts."；

```
void GuessNumber(const int number)
{
    令记录用户猜测次数的计数器count=1;
    do{
        读入用户猜测的数字;
        if (用户输入有错误，或者函数IsValidNum() 的返回值为0)
        {
            printf("Input error!\n");
            while (getchar() != '\n'); /*清除输入缓冲区中的错误数据*/
            continue;
        }
        count++; /* 记录用户猜的次数 */
        调用函数IsRight() 判断用户猜的数是大还是小，并输出相应的提示信息
        }while (函数IsRight() 返回值为0 && count <= MAX_TIMES);
        if (函数IsRight() 返回值为1) /* 若猜对，输出相应的提示信息 */
        {
            printf("Congratulations! You're so cool!\n");
        }
        else /* 若超过MAX_TIMES次仍未猜对，输出相应的提示信息 */
        {
            printf("Mission failed after %d attempts.\n", MAX_TIMES);
        }
    }
```

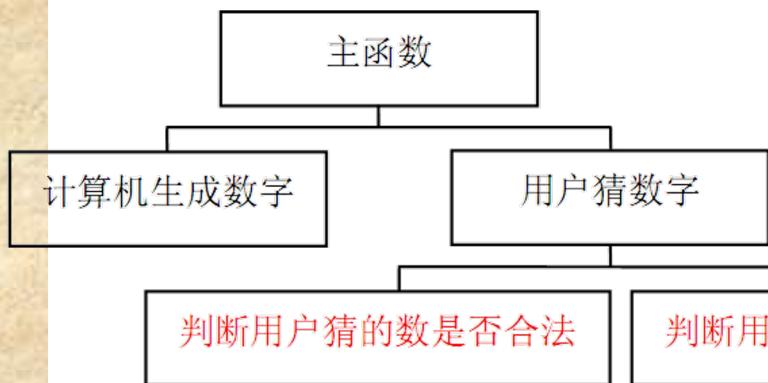
【例7.11】

```
do{
    printf("Try %d:", count);
    ret = scanf("%d", &guess);           /* 读入用户的猜测 */
    /* 处理用户输入，判断是否有输入错误，是否在合法的数值范围内 */
    if (ret != 1 || !IsValidNum(guess))
    {
        printf("Input error!\n");
        while (getchar() != '\n');      /* 清除输入缓冲区中的错误数据 */
        continue;
    }
    count++;                         /* 记录用户猜的次数 */
    right = IsRight(number, guess);  /* 判断用户猜的数是大还是小 */
}while (!right && count <= MAX_TIMES);
```

```
void GuessNumber(const int number)
{
    令记录用户猜测次数的计数器count=1;
    do{
        读入用户猜测的数字;
        if (用户输入有错误, 或者函数IsValidNum() 的返回值为0)
        {
            printf("Input error!\n");
            while (getchar() != '\n'); /*清除输入缓冲区中的错误数据*/
            continue;
        }
        count++;                  /* 记录用户猜的次数 */
        调用函数IsRight()判断用户猜的数是大还是小, 并输出相应的提示信息
        }while (函数IsRight() 返回值为0 && count <= MAX_TIMES);
        if (函数IsRight() 返回值为1)          /* 若猜对, 输出相应的提示信息 */
        {
            printf("Congratulations! You're so cool!\n");
        }
        else                                /* 若超过MAX_TIMES次仍未猜对, 输出相应的提示信息 */
        {
```

```
attempts.\n", MAX_TIMES);
```

【例7.11】用函数完成猜数游戏



```
int IsValidNum(const int number)
{
    if (number >= MIN_NUMBER && number <= MAX_NUMBER)
        return 1;
    else
        return 0;
}
```

```
int MakeNumber(void)
{
    int number;
    number = (rand() % (MAX_NUMBER - MIN_NUMBER + 1)) + MIN_NUMBER;
    assert(number >= MIN_NUMBER && number <= MAX_NUMBER);
    return number;
}
```

使用断言 (assert) 防止某些参数获得非法值，在程序调试和测试时发现错误

```
int IsRight(const int number, const int guess)
{
    if (guess < number)
    {
        printf("Wrong! Too small!\n");
        return 0;
    }
    else if (guess > number)
    {
        printf("Wrong! Too big!\n");
        return 0;
    }
    else
        return 1;
}
```

断言

```
#include <assert.h>  
void assert(int expression);
```

- **expression**为真，无声无息；为假，中断程序

- 断言仅用于调试程序，不能作为程序的功能
- 用来测试某种不可能发生的状况确实不会发生
 - **Debug**版有效
 - **Release**版失效
- 考虑使用断言的几种情况
 - 检查程序中的各种假设的正确性
 - 证实或测试某种不可能发生的状况确实不会发生

程序版式

- 缩进(Indent)——保证代码整洁、层次清晰的主要手段
- 良好风格的程序应严格采用梯形层次对应好各层次

```
#include <math.h>

main()
{
    int i;

    for (i=2; i<100; i++)
    {
        if (IsPrime(i))
            printf ("%d\t", i);
    }
}
```

```
int IsPrime(int n)
{
    int k, i;

    k = sqrt((double)n);

    for (i=2; i<=k; i++)
    {
        if (n % i == 0)
            return 0;
    }
    return 1;
}
```

程序版式

- 现在的许多开发环境、编辑软件都支持自动缩进
 - 根据用户代码的输入，智能判断应该缩进还是反缩进，替用户完成调整缩进的工作
- VC中有自动整理格式功能
 - 只要选取需要的代码，按**ALT+F8**就能自动整理成微软的**cpp**文件格式

命名规则

■ 在Linux/UNIX平台

- 习惯用 `function_name`

■ 本书采用Windows风格函数名命名

- 用大写字母开头、大小写混排的单词组合而成
- `FunctionName`

■ 变量名形式

- “名词”或者“形容词+名词”
- 如 `oldValue` 与 `newValue` 等

■ 函数名形式

- “动词”或者“动词+名词”（动宾词组）
- 如 `GetMax()` 等

对函数接口进行注释说明

```
/* 函数功能:      实现××××功能  
    函数参数:    参数1, 表示××  
                  参数2, 表示××  
    函数返回值:  ×××××  
*/
```

返回值类型 函数名(形参表)

```
{  
    ...  
    return 表达式;  
}
```

```
/* 函数功能:      用迭代法计算 n!  
    函数入口参数: 整型变量 n 表示阶乘的阶数  
    函数返回值:    返回 n! 的值  
*/
```

```
long Fact(int n)  
{  
    int i;  
    long result = 1;  
    for (i=2; i<=n; i++)  
    {  
        result *= i;  
    }  
    return result;  
}
```

挑战性的作业

挑战类型表示的极限 ——50位的 $n!$ 计算?

— 大数的存储问题



```
Enter a number to be calculated:  
40  
1! = 1  
2! = 2  
3! = 6  
4! = 24  
5! = 120  
6! = 720  
7! = 5040  
8! = 40320  
9! = 362880  
10! = 3628800  
11! = 39916800  
12! = 479001600  
13! = 6227020800  
14! = 87178291200  
15! = 1307674368000  
16! = 20922789888000  
17! = 355687428096000  
18! = 6402373705728000  
19! = 121645100408832000  
20! = 2432902008176640000  
21! = 51090942171709440000  
22! = 112400072777607680000  
23! = 25852016738884976640000  
24! = 620448401733239439360000  
25! = 15511210043330985984000000  
26! = 403291461126605635584000000  
27! = 10888869450418352160768000000  
28! = 304888344611713860501504000000  
29! = 8841761993739701954543616000000  
30! = 265252859812191058636308480000000  
31! = 8222838654177922817725562880000000  
32! = 263130836933693530167218012160000000  
33! = 8683317618811886495518194401280000000  
34! = 295232799039604140847618609643520000000  
35! = 10333147966386144929666651337523200000000  
36! = 371993326789901217467999448150835200000000  
37! = 13763753091226345046315979581580902400000000  
38! = 52302261746660111760007224100074291200000000  
39! = 20397882081197443358640281739902897356800000000  
40! = 8159152832478977343456112695961158942720000000000
```



■ Questions and answers

