

C语言与程序设计

The C Programming Language



第5章 函数与程序结构

华中科技大学计算机学院

黄宏



第5章 函数与程序结构

- 结构化编程和C程序的一般结构
- 函数的机制，包括函数定义、函数声明、函数调用、变量的存储类型等。
- 递归



5.1 C程序的一般结构

5.1.1 结构化程序设计

2条编程标准：

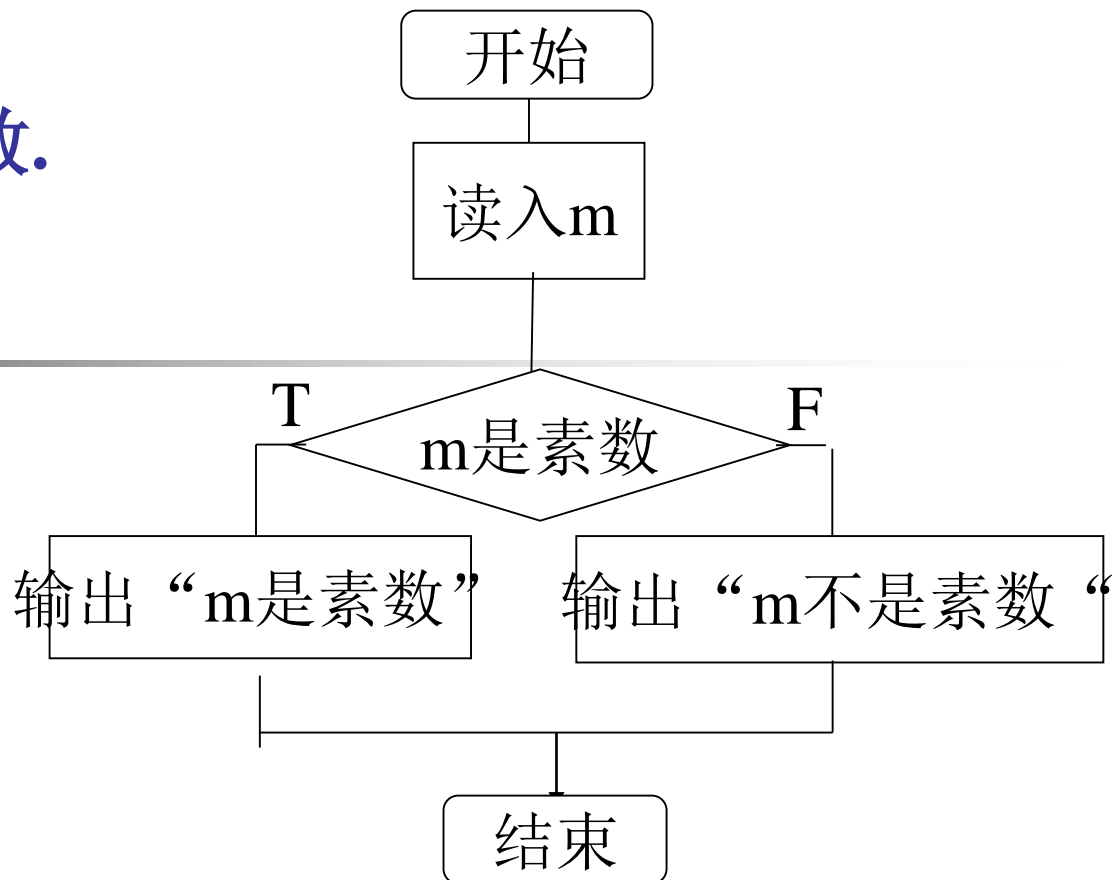
- (1) 程序中的控制流尽可能简单。**
- (2) 把一个问题逐步细化分解为若干子问题，用函数实现子问题。**



模块化程序设计

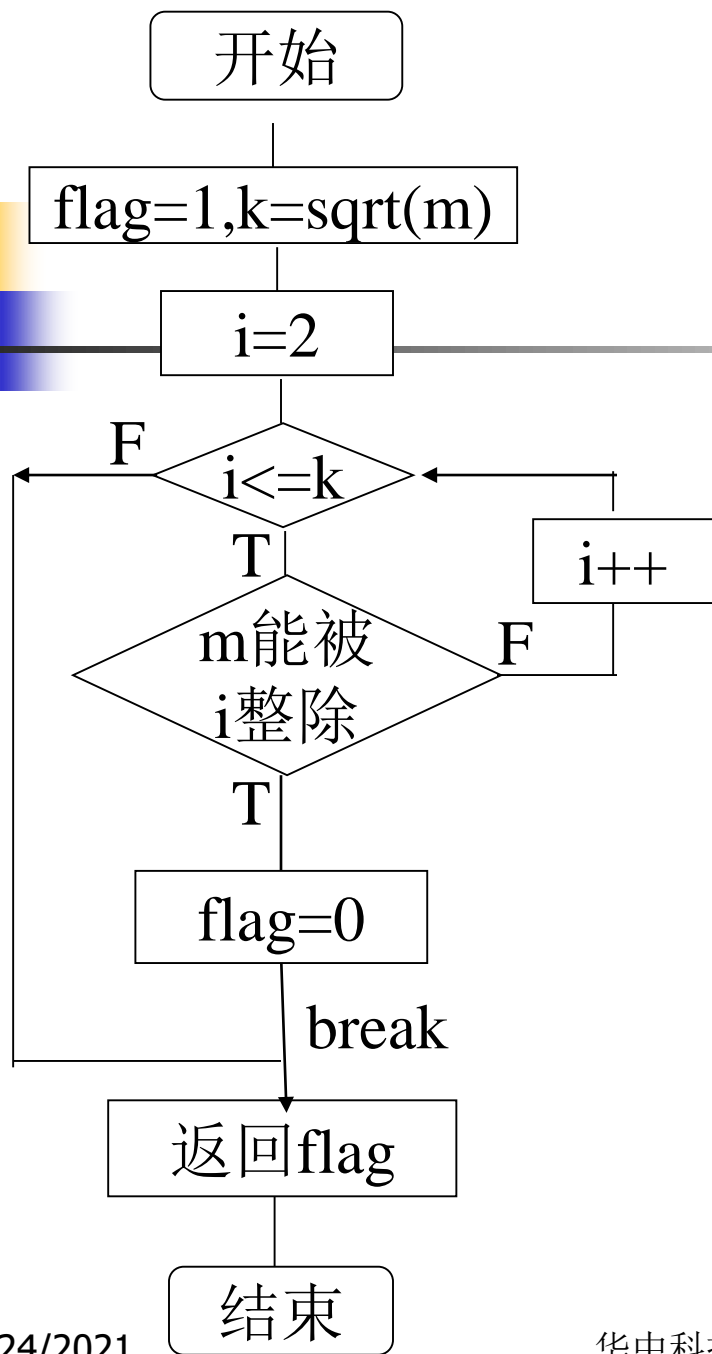
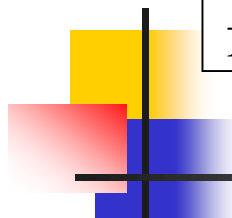
- 子程序代码公用，当需要多次完成同样功能时，只需要一份代码，可多次调用
- 程序结构模块化，可读性、可维护性及可扩充性强
- 简化程序的控制流程，程序编制方便，易于修改和调试

例 判断m是否素数.



将判断一个整数是否素数定义成函数，
是素数函数返回1，否则，返回0。

```
int isprime(int m);
```



```
#include<math.h>
int isprime(int m)
{
    int flag,k,i;
    flag=1; // 初始是素数
    k=sqrt(m);
    for(i=2;i<=k;i++)
        if (!(m%i)) {
            flag=0;
            break;
        }
    return flag;
}
```

例4.22：输出2~100的所有素数，每行输出10个数。

每输出10个数，换一次行。

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

```
#include N 10
```

```
int isprime(int m); /* 函数原型 */
```

```
int main( )
```

```
{
```

```
    int n,k=1;
```

```
    printf(“%5d”,2);
```

```
    for (n=3; n<100; n+=2 ) {
```

```
        if( isprime(n) ) /* 函数调用 */
```

```
        { printf(“%5d”,n);
```

```
            if(!(++k/N)) putchar('\n');
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```



5.1.2 蒙特卡罗模拟：猜数游戏

- 计算机随机产生一个数（1~1000）。
- 游戏者猜直到正确为止，计算出猜数的次数。
- 发出提示信息“Too high”或“Too low”。

5.1.2 蒙特卡罗模拟：猜数游戏

- **模拟算法：**编程实现现实世界中的随机事件，例如，抛硬币、掷骰子和玩牌等。
- **蒙特卡罗模拟：**使用随机数来模拟。
- **随机数：**具有不确定性和偶然性特点，应用领域：
软件测试—测试数据，加密系统—密钥，网络—验证码
- **随机数发生器：**生成随机数的函数

```
int rand (void) ;    // 在 stdlib.h中
```
- **伪随机数：**依靠计算机内部算法产生的“随机”数



自顶向下的分解问题：

分解成以下两个子任务：

- (1) 计算机产生供猜测的随机数 (1~1000)
- (2) 游戏者猜数，直至猜对。

主程序结构

do {

计算机产生一个1到1000的随机数;

游戏者猜数, 直至猜对;

printf("Play again? (Y/N) ");

scanf("%1s", &cmd);

} while (cmd == 'y' || cmd == 'Y');

int GetNum (void);

void GuessNum(int x);



主程序结构

```
do {  
    magic = GetNum( ); /* 产生随机数*/  
    GuessNum(magic); /* 猜数 */  
    printf("Play again? (Y/N) ");  
    scanf("%1s", &cmd);  
} while (cmd == 'y' || cmd == 'Y');
```



子任务1: GetNum(void)

- ① 调用标准库函数rand产生一个随机数;
- ② 将这个随机数限制在1~1000之间。

利用函数，可以实现程序的模块化，把程序中常用的一些算法或操作编成通用的函数，以供随时调用，大大简化主函数的流程，使程序设计简单和直观，提高程序的易读性和可维护性。

rand是接口**stdlib.h**中的一个函数，
它返回一个非负并且不大于常量**RAND_MAX**的随机整数，
RAND_MAX（32767）的值取决于系统。

int GetNum(void) ;

```
/******
```

函数名称: **GetNum**

函数功能: 产生一个1到**MAX_NUMBER**之间的随机数，供游戏者猜测。

函数参数: 无

函数返回值: 返回产生的随机数

```
*****/
```

```
int GetNum(void) /* 注意：后面无分号 */
```

```
{
```

```
    int x;
```

```
    printf("A magic number between 1 and %d has been chosen.\n",  
           MAX_NUMBER);
```

```
    x=rand();
```

```
    x= x % MAX_NUMBER + 1;
```

```
    return x;
```

```
}
```

```
/* 调用标准库函数rand产生一个随机数 */
```

```
/* 将这个随机数限制在1~MAX_NUMBER之间 */
```



函数 `void GuessNum(int x)`

`/*游戏者猜数，直至猜对*/`

`for(;;) {`

`输入猜测的数给变量guess`

`if（猜对了） 结束函数`

`else if（小了） 输出太小的提示`

`else 输出太大的提示`

`}`

[源程序\ex5_1_12.c](#)

main函数

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX_NUMBER 1000
int GetNum (void);      /* 函数原型 */
void GuessNum(int) ;    /* void GuessNum(int x) ;函数原型 */
int main(void)
{
    char command;
    int magic;
    do
    {
        magic = GetNum( ); /*调用GetNum产生随机数*/
        GuessNum(magic); /* 调用GuessNum猜数 */
        printf("Play again? (Y/N) ");
        scanf("%1s", &command); /* 询问是否继续 */
    } while (command == 'y' || command == 'Y');
    return 0;
}
```

行吗？

伪随机数算法

- 伪随机数是指用数学**递推公式**所产生的随机数。
- 应用最广的**递推公式**：（即**线性同余法**）

$$a_0 = \text{seed}$$

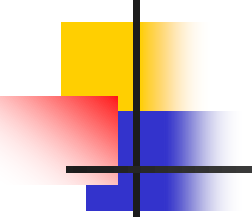
$$a_n = (A * a_{n-1} + B) \% M, n \geq 1$$

其中 **A, B, M** 是产生器设定的常数, 用户不能更改。

seed: 种子参数, 该发生器从称为**种子**（一个无符号整型数）的初始值开始用确定的算法产生随机数。

- ◆ 产生器给定了初始值
- ◆ **seed**应该在哪儿定义并初始化?
- ◆ 允许用户设置初始值（修改）
- ◆ 怎么修改?

```
int rand()  
{  
    seed = (A * seed + B) % M;  
    return seed;  
}
```

- 
- 该发生器从称为**种子**（一个无符号整型数）的初始值开始用确定的算法产生随机数。显然，通过种子产生第一个随机数后，后续的随机序列也就是确定的了，这种依靠计算机内部算法产生的随机数称为伪随机数。由此可见，随机数的产生依赖于种子，为了使程序在反复运行时能产生不同的随机数，必须改变这个种子的值，这称为初始化随机数发生器，由**函数srand**来实现。

初始化随机数

```
void srand(unsigned int x )  
{  
    seed=x;  
}
```

- **初始化随机数发生器**：改变种子**seed**值的函数

void srand (unsigned) ;

- **用什么值做种子？**

用系统时间（由**time**函数得到）作为种子。

函数**time**返回自**1970年1月1日**以来经历的秒数。

srand (time (NULL)) ;

该初始化语句放在程序什么位置？

- 函数**srand**和**rand**的原型在**stdlib.h**中，
函数**time**的原型在**time.h**中

源程序\ex5_1.c

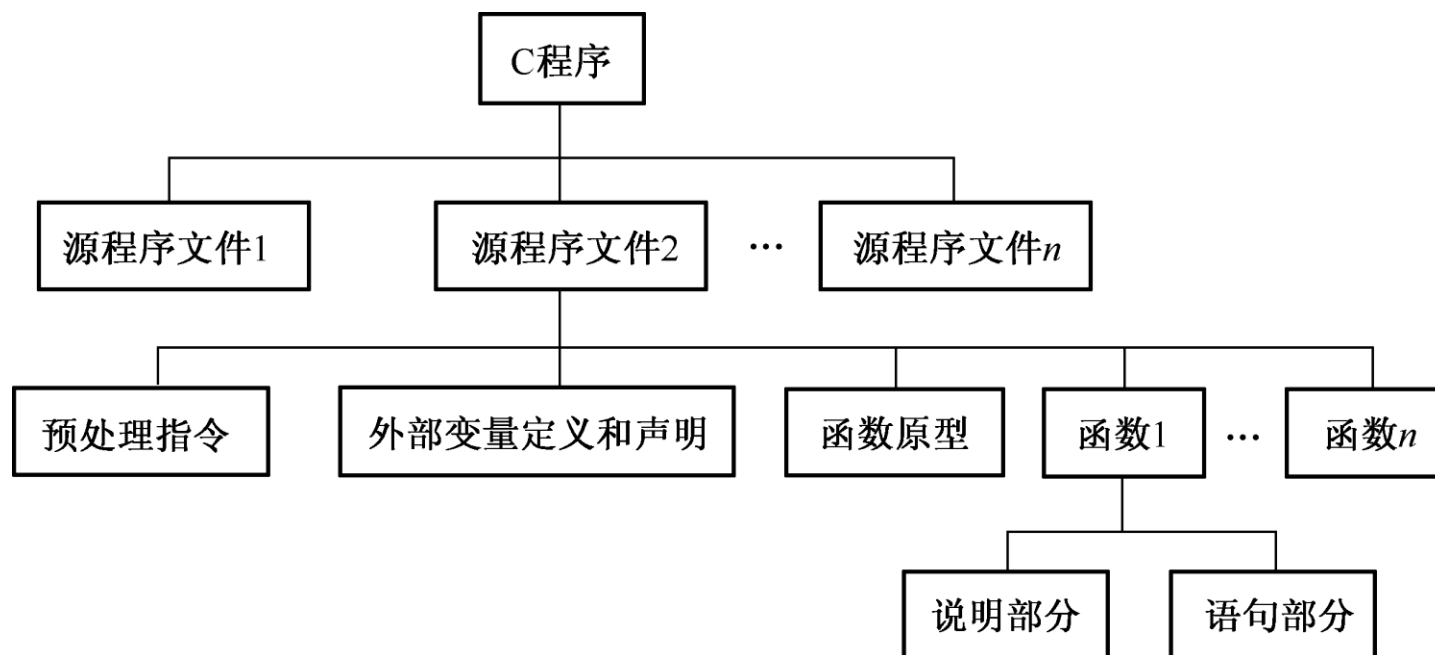


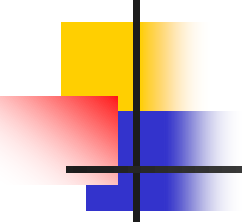
```
unsigned int seed; // 外部变量
```

```
int rand( )  
{  
    seed= (A*seed+B) % M;  
    return seed;  
}
```

```
void srand(int x )  
{  
    seed=x;  
}
```

5.1.3 C程序的结构



- 
- 每个源文件可单独编译生成目标文件，组成一个**C**程序的所有源文件都被编译之后，由连接程序将各目标文件中的目标函数和系统标准函数库的函数装配成一个可执行**C**程序。



5.2 函数的定义与函数的声明

- 程序中若要使用自定义函数实现所需的功能，需要做三件事：
 - ① 按语法规则编写完成指定任务的函数，即定义函数；
 - ② 有些情况下在调用函数之前要进行函数声明；
 - ③ 在需要使用函数时调用函数

5.2.1 函数的定义

函数定义的一般形式为：
类型名 函数名（参数列表）

{

声明部分

语句部分

}

形式参数（简称形参）
无参数：有**void**或无

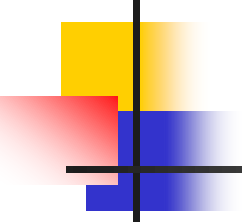
无返回值：类型为**void**



5.2.2 函数的返回值

- **return**语句可以是如下两种形式之一：
 - (1) **return ;** **/* void函数 */**
 - (2) **return 表达式 ;** **/* 非void函数 */**
- **void**函数也可以不包含**return**语句。如果没有**return**语句，当执行到函数结束的右花括号时，控制返回到调用处，把这种情况称为离开结束。
- 表达式值的类型应该与函数定义的返回值类型一致，如果不相同，就把表达式值的类型自动转换为函数的类型。

函数返回的值，程序可以使用它， 也可以不使用它



```
while(...) {  
    getchar();          /* 返回值不被使用 */  
    c=getchar();        /* 返回值被使用  */  
    ...  
}
```



【例5.2】 写一个函数IsPrime

- **IsPrime**判断整数n是否为素数。如果n是素数，则返回1；如果n不是素数，则返回0。
- 源程序\ex5 2.c
- 在一个函数中可以有多条**return**语句，此种情况下的**return**语句通常被作为选择语句的子句出现，最终被执行的只是其中的一个。因为，一旦某个return语句被执行，控制立即返回到调用处，其后的代码不可能被执行。

5.2.3 函数的声明

- 函数定义出现在函数调用后
 - 被调用函数在其它文件中定义
- 必须在函数调用之前给出函数原型

类型名 函数名（参数类型表）；

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

或

`int max(int , int) ;`

无参数：必须写**void**



良好的编程风格

- 在函数调用之前必须给出它的函数定义、函数原型或两者都给出。
- 引入标准头文件的主要原因是它含有函数原型。




5.3 函数调用与参数传递

5.3.1 函数调用

1. 函数调用的形式

函数名（实参列表）



实参是一个表达式
无参函数：为空



函数调用的形式

(1) 作为表达式语句出现。

```
GuessNum( );  
putchar(c);
```

(2) 作为表达式中的一个操作数出现。例如：

```
c=getchar()  
magic = GetNum( );
```

(3) 作为函数调用的一个实参出现，即嵌套调用。

```
printf("%10.0f",sqrt(x));  
while(putchar (getchar())!='#' );
```



2. 函数调用的执行过程

- 为了说明函数的调用过程，请看下面程序。

【例5.3】编写程序实现如下功能：分列整齐地显示整数1到10的2至5次幂。输出结果如下所示：

Int	Square	Cube	Quartic	Quintic
1	1	1	1	1
2	4	8	16	32
3	9	27	81	243
4	16	64	256	1024
5	25	125	625	3125
6	36	216	1296	7776
7	49	343	2401	16807
8	64	512	4096	32768
9	81	729	6561	59049
10	100	1000	10000	100000

- 源程序\ex5_3.c

3. 实参的求值顺序

- 由具体实现确定，有的从左至右计算，有的按从右至左计算。

a=1;

power(a,a++)

----从左至右: **power(1,1)**

----从右至左: **power(2,1)** (多数)

为了保证程序清晰、可移植，应避免使用会引起副作用的实参表达式。



5.3.2 参数的值传递

- 参数的传递方式是“值传递”，实参的值单向传递给相应的形参。如果实参、形参都是 x ，被调用函数不能改变实参 x 的值。

【例5.4】 改写例5.3来说明值传递概念。

```
#include<stdio.h>
double power(int,int); /* 函数原型 */
int main(void)
{
    int x, n;
    /* 分列整齐地显示整数1到10的2至5次幂 */
    for(x=1;x<=10;x++)
    {
        for(n=2;n<=5;n++)
            printf("%10.0f", power(x,n));
        printf("\n");
    }
    return 0;
}

double power(int x, int n) // 计算x的n次方
{
    double p;
    for (p=1;n>0;n--)
        p*=x;
    return p;
}
```



传地址（引用）调用

- 传地址（引用）调用是将变量的地址传递给函数，函数既可以使用，也可以改变实参变量的值。**标准库函数scanf就是一个引用调用的例子**，通过地址实参返回一个或多个数据。程序员也可以定义这种带有地址形参的函数，这部分内容将在第9章中介绍。
- `int x; scanf("%d",&x);`
- `int x; scanf(“%d”,x); /*错误,传递的是x的值,被调用函数不可能改变x的值*/`



5.4 作用域与可见性

- 作用域是指标识符（变量或函数）的有效范围，也就是指程序正文中可以使用该标识符的那部分程序段。
- 可见性：块多重嵌套时，同名外层变量不可见
- 按照作用域，变量分：
局部变量和全局变量



5.4.1 局部变量和全局变量

- **局部变量**：在函数内部定义的变量，作用域是定义该变量的程序块，程序块是带有说明的复合语句（包括函数体）。不同函数可同名，同一函数内不同程序块可同名。**形式参数是局部变量。**
- **全局变量**：在函数外部定义的变量，其缺省作用域从其定义处开始一直到其所在文件的末尾，由程序中的部分或所有函数共享。

```
int p=1, q=5;
```

全局变量

```
float f1( int a)
```

```
{ int b,c;
```

a,b,c有效

```
.....
```

局部变量

```
}  
char c1,c2;
```

p,q有效

```
main( )
```

```
{ int m, n;
```

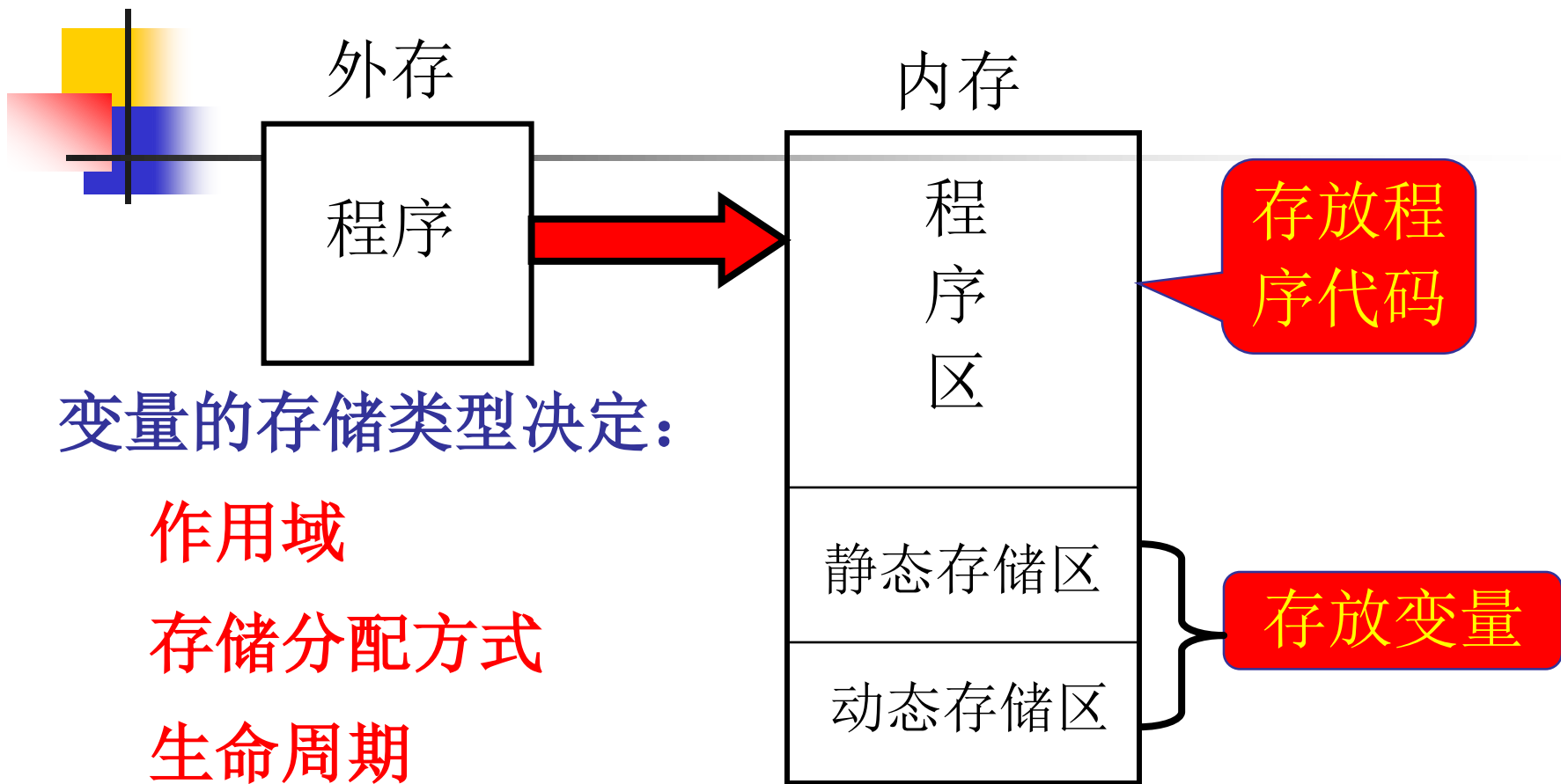
m,n有效

```
.....
```

```
}
```

c1,c2有效

5.5 变量的存储类别



变量的存储类型决定：

作用域

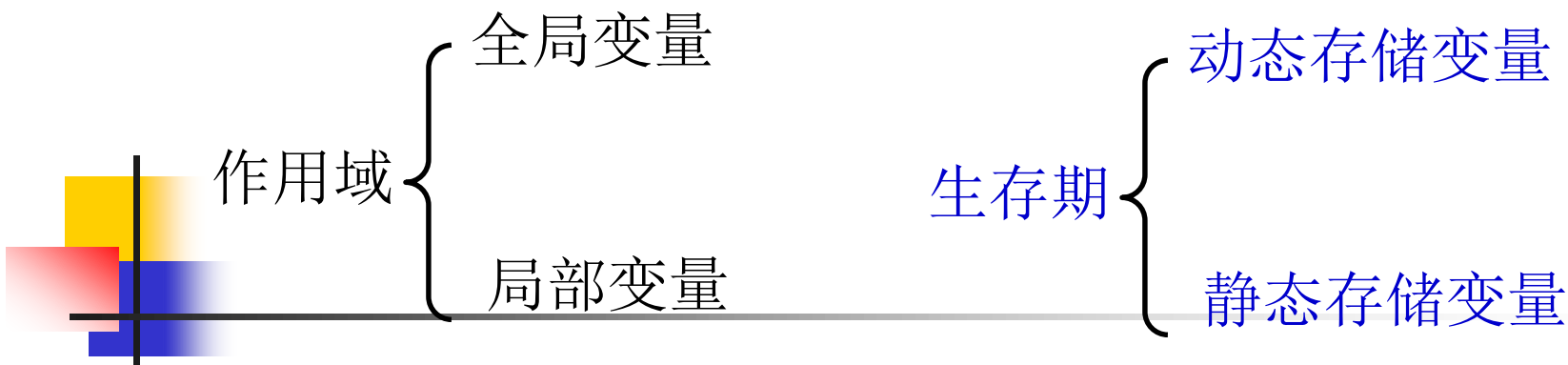
存储分配方式

生命周期

初始化方式

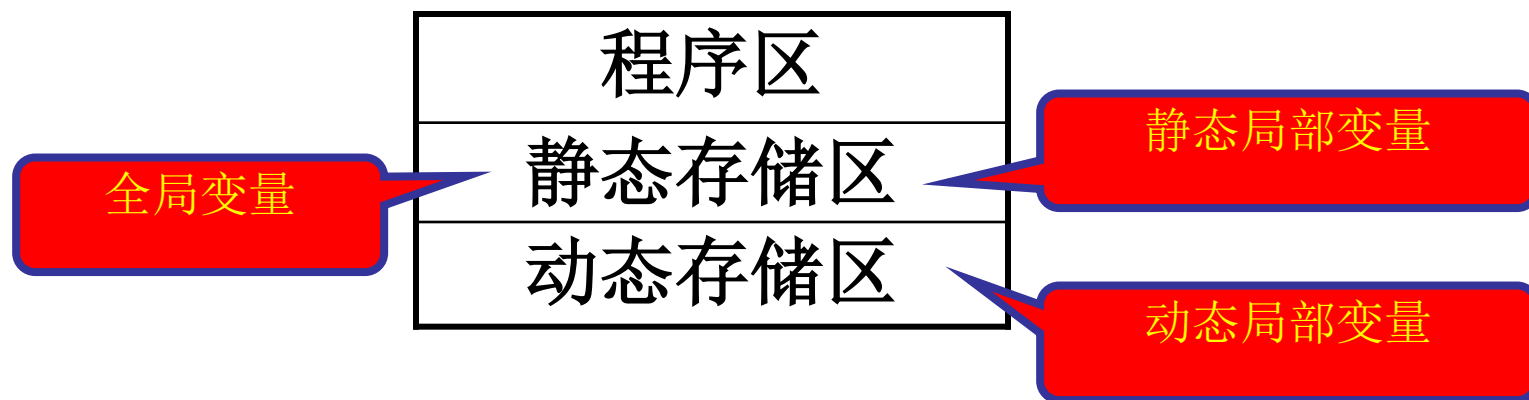
需要区分变量的存储类型

关键字有：**auto**、**extern**、**static**和**register**



静态存储：在文件运行期间有固定的存储空间，直到文件运行结束。

动态存储：在程序运行期间根据需要分配存储空间，**函数结束后立即释放空间**。若一个函数在程序中被调用两次，则每次分配的单元有可能不同。





5.5.1 存储类型auto

- 局部变量的缺省存储类型是**auto**，称自动变量
- `auto int a;` /*等价于`int a;` 也等价于`auto a;` */
- **作用域**：局部于定义它的块，从块内定义之后直到该块结束有效。
- **存储分配方式**：动态分配方式,即在程序运行过程中分配和回收存储单元。
- **生命周期**：短暂的，只存在于该块的执行期间。
- **初始化方式**：定义时没有显示初始化，其初值是不确定的；有显示初始化，则每次进入块时都要执行一次赋初值操作。



5.5.2 存储类型extern

- 外部变量的存储类型是**extern**，但在定义时不使用关键字**extern**。
- 外部变量的**作用域**：从定义之后直到该源文件结束的所有函数，通过用**extern**做声明，外部变量的作用域可以扩大到整个程序的所有文件。
- **存储分配方式**：静态分配方式，即程序运行之前，系统就为外部变量在静态区分配存储单元，整个程序运行结束后所占用的存储单元才被收回。
- **生命周期**：永久的，存在于整个程序的执行期间。
- **初始化方式**：定义时没有显示初始化，初值**0**；有显示初始化，只执行一次赋初值操作。

extern 声明

```
srand(int x)
```

```
{
```

```
    seed=x;
```

```
}
```

```
int seed=C;
```

```
int rand( )
```

```
{
```

```
    seed = (A*seed+B) % M ;
```

```
    return seed;
```

```
}
```

引用出错：无作用

外部变量的**定义性**声明语句
从定义点开始有效直至文件尾

extern声明

```
srand(int x)
```

```
{
```

```
    extern int seed;
```

```
    seed=x;
```

```
}
```

```
int seed=C;
```

```
int rand( )
```

```
{
```

```
    seed = (A*seed+B) % M ;
```

```
    return seed;
```

```
}
```

外部变量的**引用性**声明语句

外部变量的**定义性**声明语句
从定义点开始有效直至文件尾



定义性声明和引用性声明的区别

- ◆ 外部变量的定义性声明---- 分配存储单元
 - ✓ 位于所有的函数之外
 - ✓ 定义一次
 - ✓ 可初始化
- ◆ 外部变量的引用性声明----通报变量的类型
 - ✓ 位于在函数内或函数外
 - ✓ 出现多次
 - ✓ 不能初始化

外部变量的定义性声明和引用性声明

- 局部变量只有定义性声明，没有引用性声明。而外部变量有定义性声明和引用性声明，两者具有严格的区别。
- 外部变量的定义必须在所有的函数之外，且只能定义一次，目的是为之分配存储单元。
- 外部变量的引用性声明既可以出现在函数内，也可以出现在函数外，而且可以出现多次，仅用于通报变量的类型，并不分配存储单元，只是表明在代码中要按声明的类型使用它。外部变量的初始化只能出现在其定义中。
- 在C程序的不同源文件之间，或者在同一源文件的不同函数之间必须共享变量时，外部变量是很有用的。

外部变量 PK 形式参数

- 函数之间可以通过外部变量进行通信。
- 一般地，函数间通过形式参数进行通信更好。这样有助于提高函数的通用性，降低副作用。

```
int main(void)
{ int m;
  m = rand( )%10+1;
  guess(m);
  guess(12);
}
void guess(int x)//猜x
{ ..... }
```

Int magic;

Int main(void)

```
{ Int m;
  m = rand( )%10+1;
  magic=m;
  } guess();
```

void guess() //猜magic

void guess() //猜magic

```
{ ..... }
```



```
int min;
```

外部变量

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

```
int maxmin (int x, int y)
```

```
{ int z;
```

```
min=(x<y)?x : y;
```

```
z=(x>y)? x : y ;
```

```
return z;
```

函数值为4

```
}
```

```
int main (void)
```

```
{ int a=4,b=1,max;
```

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

```
printf("The max is %d\nThe min is %d",max,min);
```

```
return 0;
```

```
}
```

The max is 4

The min is 1

定义点前或其它文件引用： 需用**extern**作引用声明

```
    srand(int x)
    {
        extern int seed;
        seed=x;
    }
    int seed=C;
    int rand( )
    {
        seed = (A*seed+B) % M ;
        return seed;
    }
```

引用在前，定义在后

文件file1.c中的内容为:

```
int a;
```

```
int main(void)
```

```
{ extern int fun (int);
```

```
  int b=3, c, d, m;
```

```
  ...
```

```
  c=a*b;
```

```
  d= fun(m);
```

```
  ...
```

```
}
```

外部变量定义

外部变量说明

引用本文件外
定义的全局变量

文件file2.c中的内容为:

```
extern int a;
```

```
int fun (int n )
```

```
{ int i, y=1;
```

```
  ...
```

```
  y*=a;
```

```
  return y;
```

```
}
```



5.5.3 存储类型static

关键字**static**有两个重要而截然不同的用法:

- (1) 用于定义局部变量, 称为静态局部变量。
- (2) 用于定义外部变量, 称为静态外部变量。


存储分配方式:静态分配方式

生命周期:永久的,

缺省初值: 0, 只执行一次

静态局部变量和自动变量有根本性的区别。

静态外部变量和外部变量区别: 作用域不同。



```
srand(int x)
{
    seed=x; // seed在此无作用
}
```

```
int rand( )
{
    int seed=D; // 自动变量，存于栈，每次调用初始化为D
    seed = (A*seed+B) % M ;// 不能产生随机系列
    return seed;
}
```




```
srand(int x)
```

```
{  
    seed=x; // seed在此无作用，用户无法改变种子参数  
}
```

```
int rand( )
```

```
{  
    static int seed=D; // 静态局部变量，存于静态区  
                        // 初始化只执行1次  
    seed = (A*seed+B) % M; // 可以产生随机系列  
    return seed;  
}
```



```
int seed=D; // 外部变量，存于静态区， 初始化只执行1次
srand(int x)
{
    seed=x; // seed在此有作用， 用户可改变种子参数
}
```

```
int rand( )
{
    seed = (A*seed+B) % M ; // 可以产生随机系列
    return seed;
}
```



1. 静态局部变量

- 作用域：自动变量一样

【例5.7】 编程计算 $1!+2!+3!+4!+\dots n!$

- 将求阶乘定义成函数，要求使用**static**使计算量最小。


```
scanf("%d",&n);
for(i=1;i<=n;i++)
    sum+=fac(i);
printf("1!+2!+...+%d!=%ld\n",n,sum);
return 0;
```

```
-}
```

```
/******
```

函数名称: fac

函数功能: 利用静态局部变量的特性求一个整数的阶乘。

函数参数: 形参n是int型

函数返回值: 返回n的阶乘, 类型long

```
*****/
```

```
long fac(int n)
```

```
-{
```

```
    static long f=1;    /* 静态局部变量 */
```

```
    f *=n;
```

```
    return f;
```

```
}
```

2. 静态外部变量

```
static int seed=D;
srand(int x)
{
    seed=x;
}
int rand( )
{
    seed = (A*seed+B) % M ;
    return seed;
}
```

静态外部变量

只能作用于定义它的文件，
其它文件中的函数不能使用



2. 静态外部变量

- 静态外部变量和外部变量的区别是作用域的限制。
- 静态外部变量只能作用于定义它的文件，其它文件中的函数不能使用，
- 外部变量用**extern**声明后可以作用于其它文件。
- 使用静态外部变量的好处在于：当多人分别编写一个程序的不同文件时，可以按照需要命名变量而不必考虑是否会与其它文件中的变量同名，**保证文件的独立性**。
- 和局部变量能够屏蔽同名的外部变量一样，一个文件中的静态外部变量能够屏蔽其他文件中同名的外部变量。在静态外部变量所在的文件中，同名的外部变量不可见。

5.5.4 存储类型register

- 用来定义局部变量, **register** **建议**编译器把该变量存储在计算机的高速硬件寄存器中, 除此之外, 其余特性和自动变量完全相同。
- 使用**register**的目的是为了提高程序的执行速度。程序中最频繁访问的变量, 可声明为**register**。

```
register int i;           /* 等价于register i; */  
for (i=0;i<=N;i++) {    ...}
```

- 不可多, 必要时使用。
- 无地址, 不能使用**&**运算。



5.4.1 递归概述

- 递归是一种函数在其定义中直接或间接调用自己的编程技巧。递归策略只需少量代码就可描述出解题过程所需要的多次重复计算，十分简单且易于理解。
- 递归调用：函数直接调用自己或通过另一函数间接调用自己的函数调用方式
$$f() \{ \dots f(); \dots \}$$
- 递归函数：在函数定义中含有递归调用的函数



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

```
// 输出 n ~ 1
```

```
void prn_int(int n)
```

```
{ if (n>0) {
```

```
    printf ("%d ",n);
```

```
    prn_int(n-1);
```

```
}
```

```
}
```

```
int main(void)
```

```
{ prn_int(5);
```

```
    return 0;
```

```
}
```

递归函数：在定义中含有递归调用

递归调用：调用自己



【例5.10】 用递归法计算阶乘n!

- 阶乘的计算是一个典型的递归问题。**n!** 定义为:

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

- 这是递归定义式，对于特定的**k**，**k!**只与**k**和**(k-1)!**有关，上式的第一式是递归结束条件，对于任意给定的**n!**，将最终求解到**1!** 或**0!**。

$n!$ 的递归实现



```
/*
```

函数功能：递归法求一个整数的阶乘。

函数参数：参数 n 为`int`,表示要求阶乘的数。

函数返回值： n 的阶乘值，类型为`long`。 `*/`

```
long factorial(int n)
```

```
{
```

```
    if (n==0||n==1)          /* 递归结束条件 */
```

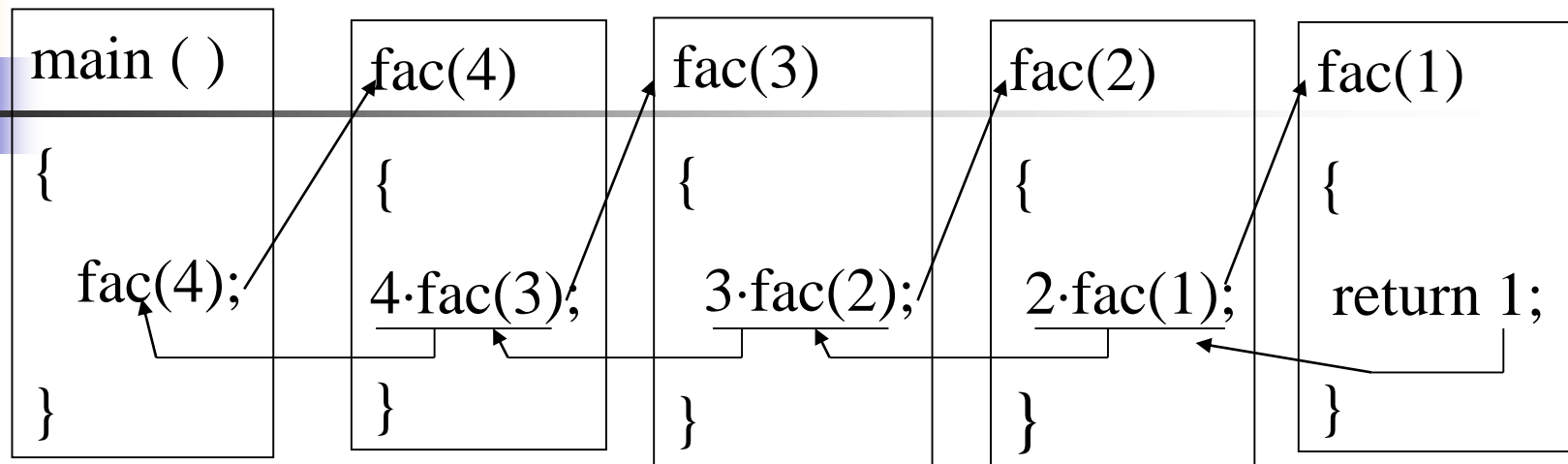
```
        return 1;
```

```
    else
```

```
        return (n*factorial(n-1)); /* 递归调用 */
```

```
}
```


4! 的递归执行过程



- ◆ 把求解问题转化为规模较小的子问题，通过多次递归一直到可以得出结果的最小解，然后通过最小解逐层向上返回调用，最终得到整个问题的解。
- ◆ 将递归概括为一句话：“能进则进，不进则退”
- ◆ 简化算法（易于理解）、但不节省存储空间，运行效率也不高(运行时开销大，效率低)

$n!$ 的迭代实现



```
/* 迭代法计算n! */
```

```
long factorial_iteration( int n )
```

```
{
```

```
    int result = 1;
```

```
    while( n>1 ) {
```

```
        result *=n;
```

```
        n--;
```

```
    }
```

```
    return result;
```

```
}
```



对比

◆ 两种方式的比较

两种实现方式都非常简单易懂，在实际项目中，为了效率，应该优先选择迭代。

◆ 什么情况下使用递归

用递归能容易编写和维护代码，且运行效率并不至关重要



递归算法的特点

- 递归算法的运行效率较低，耗费的计算时间较长，占用的存储空间也较多。
 - 递归算法结构紧凑、清晰、可读性强、代码简洁。
 - 大多数的简单递归函数都能改写为等价的迭代形式。
- 什么情况下使用递归呢？如果用递归能容易编写和维护代码，且运行效率并不至关重要，那么就使用递归。例如，像二叉树这样的数据结构，由于其本身固有的递归特性，特别适合于用递归处理；像回溯法等算法，一般也用递归来实现。



递归的要素

(1) 每次调用在规模上有所缩小。

(2) 递归结束条件

当子问题的规模足够小时，必须能够直接求出该规模问题的解。

例 编写一个递归函数计算**Fibonacci**数列的第**n**项。

```
/* Recursive fibonacci : Compute the n item */
```

```
long fibonacci (long n )
```

```
{
```

```
    if ( n == 1 || n == 2) return 1;
```

```
    else
```

```
        return fibonacci(n-1)+fibonacci(n-2);
```

```
}
```



5.4.3 递归函数设计

- 递归是一种强大的解决问题的技术，其基本思想是将复杂问题逐步转化为稍微简单一点的类似问题，最终将问题转化为可以直接得到结果的最简单问题。在较高级的程序中，递归是一个很重要的概念。

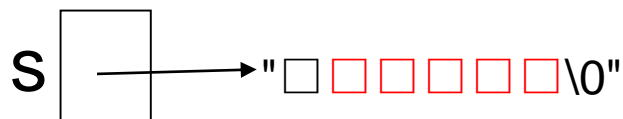


字符串的递归处理

- 字符串是以空字符（'\0'）结尾的字符序列。因此，可以把字符串看成“一个字符后面再跟一个字符串”，或者仅有一个空字符组成的空串。这个字符串的定义说明字符串是一种递归的数据结构，可以用递归的方法对一些基本的处理字符串函数进行编写。

【例1】 用递归实现标准库函数strlen(s)

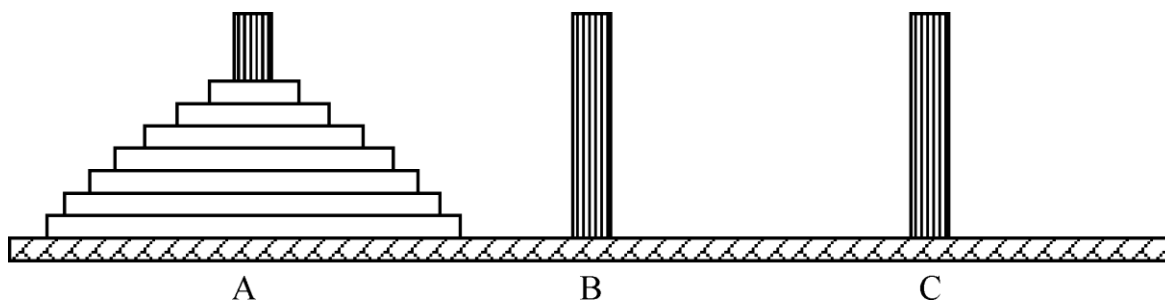
- 字符串看成 “一个字符后面再跟一个字符串”



```
int strlen(char s[ ])  
{  
    if(s[0]=='\0')  
        return 0;  
    else  
        return(1+strlen(s+1));  
}
```

汉诺塔问题

- 问题：木桩A上有64个盘子，盘子大小不等，大的在下，小的在上。把木桩A上的64个盘子都移到木桩C上，条件是一次只允许移动一个盘子，且不允许大盘放在小盘的上面，在移动过程中可以借助木桩B。



【例5.13】 设计一个求解汉诺塔问题的算法。

- 这是一个典型的用递归方法求解的问题。要移动 n 个盘子，可先考虑如何移动 $n-1$ 个盘子。分解为以下3个步骤：
 - (1) 把A上的 $n-1$ 个盘子借助C移到B。
 - (2) 把A上剩下的盘子（即最大的那个）移到C。
 - (3) 把B上的 $n-1$ 个盘子借助A移到C。
- 其中，第（1）步和第（3）步又可用同样的3步继续分解，依次分解下去，盘子数目 n 每次减少1，直至 n 为1结束。这显然是一个递归过程，递归结束条件是 n 为1。

函数move(n,a,b,c)

/ 将n个盘从a借助b，移至c */*

```
void move(int n,int a,int b,int c )
{
    if (n==1) printf(" %c-->%c\n ", a, c);
    else {
        move (n-1, a,c, b);
        printf(" %c-->%c\n ", a, c);
        move (n-1, b, a, c);
    }
}
```

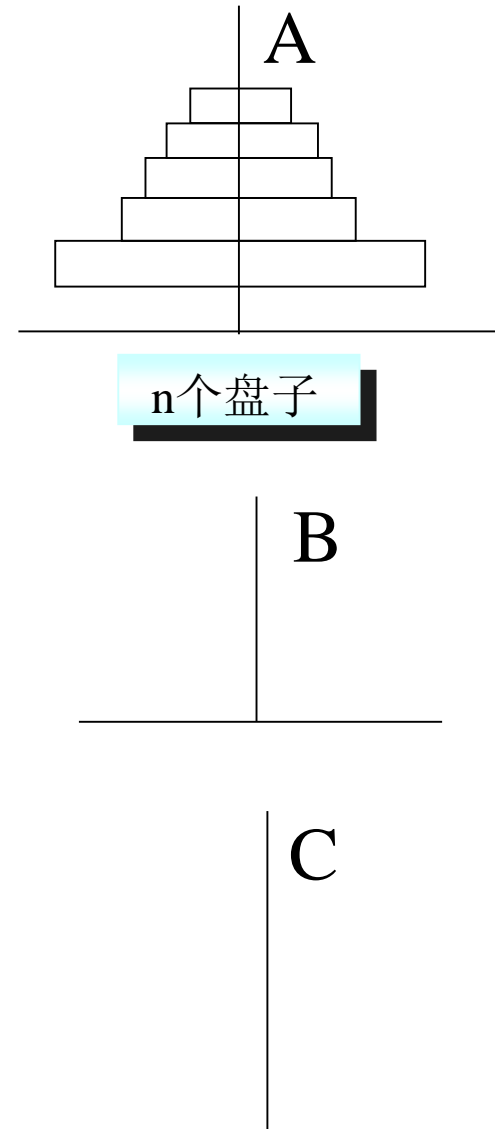
移动次数：1个盘子：移动1次

2个盘子：移动 $2*1+1=3$ 次

3个盘子：移动 $2*3+1=7$ 次

n个盘子：移动 (2^n-1) 次

64个盘子：移动 $(2^{64}-1)$ 次





5.4.5 分治法与快速排序

- **分治法**：将一个大问题划分成若干互相独立的子问题，这些子问题与原问题相同但规模更小。
- **递归求解子问题**：分治与递归像一对孪生兄弟，经常同时应用在算法设计之中。



12.3 分治法与快速排序

- 分治法的基本思想是，将一个大问题划分成若干子问题，这些子问题互相独立且与原问题相同。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型一致而其规模却不断缩小，最终使子问题缩小到很容易直接求出其解。这自然导致递归过程的产生。分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

quick排序算法

- quick排序法是C.A.R.Hoare于1962年发明的。
- 基于分治策略

quick排序算法

(1) 分解：将数组分为两部分

给定数组 $a[\text{left}] \sim a[\text{right}]$ ，从中选择一个元素（称为分区元素），并把其余元素划分为两个子集合：

$a[\text{left}] \sim a[\text{split}-1]$ 、 $a[\text{split}]$ 、 $a[\text{split}+1] \sim a[\text{right}]$

左边部分的所有元素都比右边部分的元素小。

(2) 递归求解

对两个子集合递归应用同一过程，当某个子集合中的元素个数小于2时，递归结束。



函数QuickSort

```
/* quick排序法 */
void QuickSort(int a[ ],int left,int right)
{
    int split;    /* 分区位置 */
    if(left<right) /* 待排序数组的元素个数至少为2 */
    {
        split=partition(a,left,right); /* 将数组元素分成两部分*/
        QuickSort(a,left,split-1);    /* 对左边部分递归排序 */
        QuickSort(a,split+1,right);    /* 对右边部分递归排序 */
    }
}
```

- 选择中间元素作为切分元素

$v[] = \{ 5 \ 2 \ 6 \ 4 \ 7 \ 3 \ 1 \ 8 \}$

(1) 移切分元素到最左边位置

4 5 2 6 7 3 1 8

left last i i right

(2) 分区

4 2 5 6 7 3 1 8

last i i i i

4 2 3 6 7 5 1 8

last i i

4 2 3 1 7 5 6 8

last i i

(3) 恢复切分元素到中间

1 2 3 4 7 5 6 8

last i

/* 将数组a中的元素a[left]至a[right]分成左右两部分，
返回切分点的下标。 */

int partition_v1(int a[],int left,int right)

```
{ int i, last;    /*last为左边部分最后一个元素的位置*/  
  int split=(left+right)/2; /*选择中间元素作为切分元素 */  
  swap(a, left, split); /* 移切分元素到最左边位置 */  
  last = left;    /* 初始化last */  
  for (i = left + 1; i <= right; i++) /* 分区： 从左至右扫描 */  
    if (a[i] < a[left]) /* 小的数移到左边*/  
      swap(a, ++last, i);  
  swap(a, left, last); /* 将切分元素移到两部分之间*/  
  return last;  
}
```

```
/* swap: 交换 v[i] and v[j] */  
void swap(int v[], int i, int j)  
{  
    int temp;  
    temp = v[i];  
    v[i] = v[j];  
    v[j] = temp;  
}
```

- **stdlib.h** 中有 函数 **qsort** 能对任意类型的对象排序。

分区

- 选择中间元素作为切分元素

**i 从左至右扫描，
找到比切分元素大的数停止**

j从右至左扫描，
找到比切分元素小的数停止

$V[] = \{ 5 \quad 2 \quad 6 \quad 4 \quad 7 \quad 3 \quad 1 \quad 8 \}$

① 移切分元素到最左边位置

(2) 分区

	4	5	2	6	7	3	1	8
		↑					↑	↑
		j					j	

交换a[j]和a[j]

(2) 分区

4 1 2 6 7 3 5 8

4 1 2 i i 7 6 j 5 8

扫描相遇结束

(3) 恢复切分元素到中间

3 1 2 4 7 6 5 8

↑ j ↑ i

函数partition

```
int partition(int a[ ],int left,int right )
{
    int i=left,j=right+1;
    int split=(left+right)/2; /* 选择中间元素作为切分元素 */
    swap(a,left,split); /* 将切分元素移到数组的开头 */
    for ( ; ; )
    {
        while(a[++i]<=a[left] && i <= right); /*从左至右扫描 */
        while(a[--j]> a[left]); /* 从右至左扫描 */
        if(i>=j) break; /* 扫描相遇（或交叉）结束循环 */
        swap(a,i,j); /* 交换左右两边的元素 */
    } /* j 是切分元素的位置 */
    swap(a,left,j); /* 将切分元素重新移到中间 */
    return j; /* 返回切分元素的下标 */
}
```



切分元素的选择

- 选择切分元素有很多种策略，最简单的方法是选用数组的第一个元素，该法对随机排列的数组很好，如果数据基本有序，则执行效率很差。上述程序中的方法可极大提高对有序或基本有序数组排序的效率。更加完善的策略是选择中间值，或至少是介于最大值和最小值之间的数值。
- 编写测试主函数用于输出排序结果，具体代码如下。
- **\源程序\ex12_5.c**