chapter 5

函数

武汉光电国家研究中心

李春花

主要内容

1. 模块化程序设计思想
2. 函数定义、函数声明、函数调用
3. 变量的存储类型与作用域
4. 函数间的数据传递
5. 递归函数

5.1 模块化程序设计

基本思想：把一个复杂问题，逐步细化“分解”成若干子问题，分别设计其算法，再将其“组合”从而获得问题的解决方案。

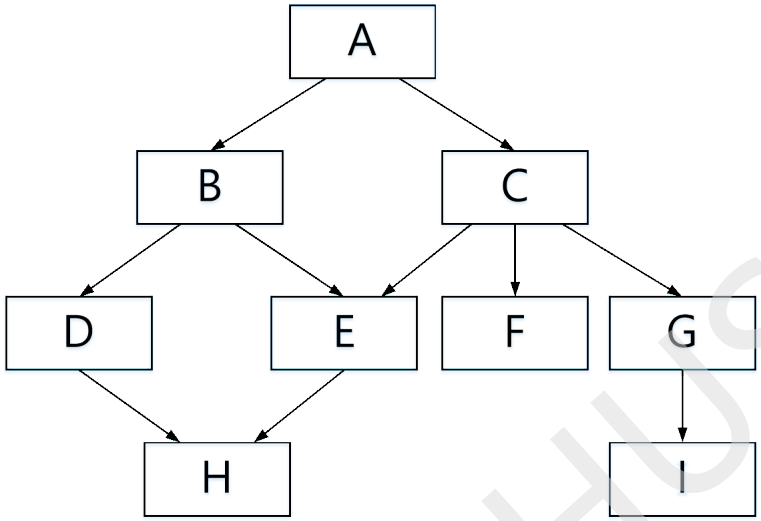
子问题的算法对应的程序称为子程序。

子程序 { 过程(没有返回值的子程序)；函数(具有返回值的子程序) } 函数

“组合” 是通过函数(子过程)之间调用方法实现。函数之间数据交换可以采用参数、返回值和全局变量3种方式。

模块化设计的优越性

模块化软件示意图

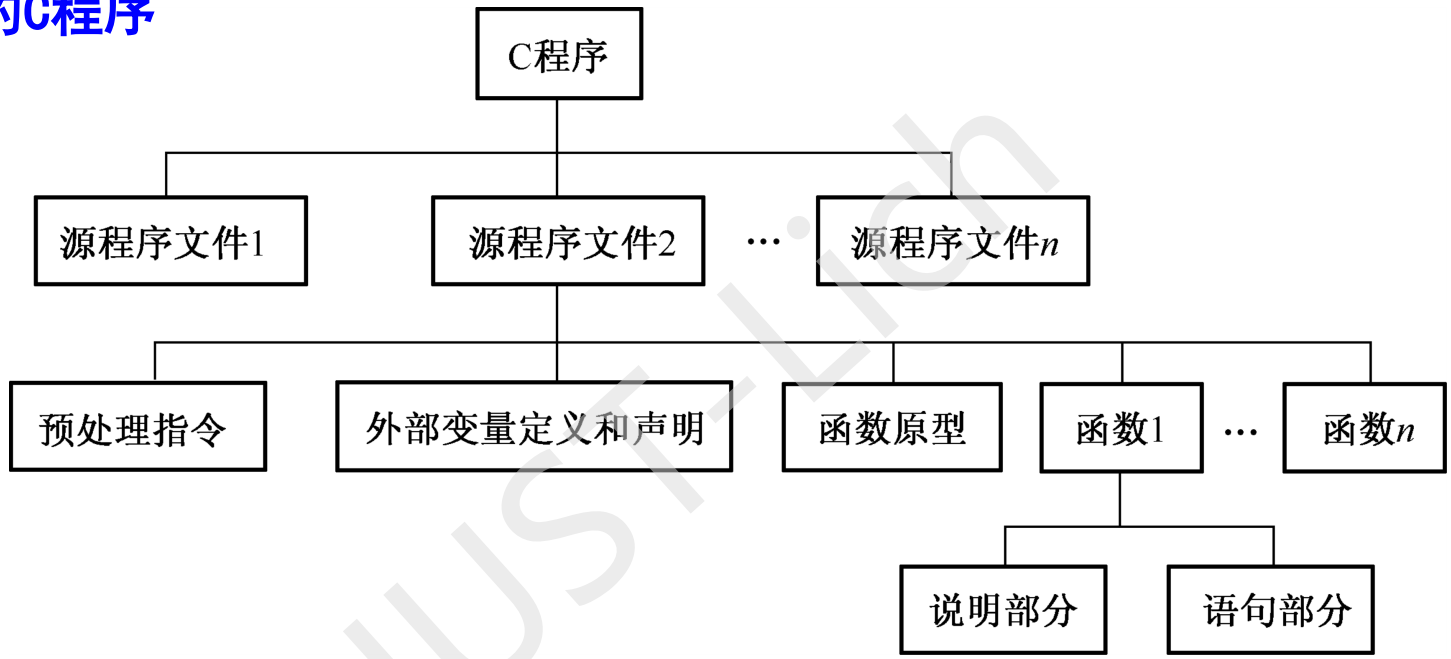


- ✓ 使程序编制方便，易于管理、修改和调试。
- ✓ 增强了程序的可读性、可维护性和可扩充性，方便于多人分工合作完成程序的编制。
- ✓ 函数可以公用，避免在程序中使用重复的代码。
- ✓ 提高软件的可重用性，软件的可重用性是转向面向对象程序设计的重要因素。

从软件工程上看，**可靠性、效率、可维护性**是软件质量的主要评价指标。因此，模块化软件能够成为高质量的软件。

C语言程序的结构

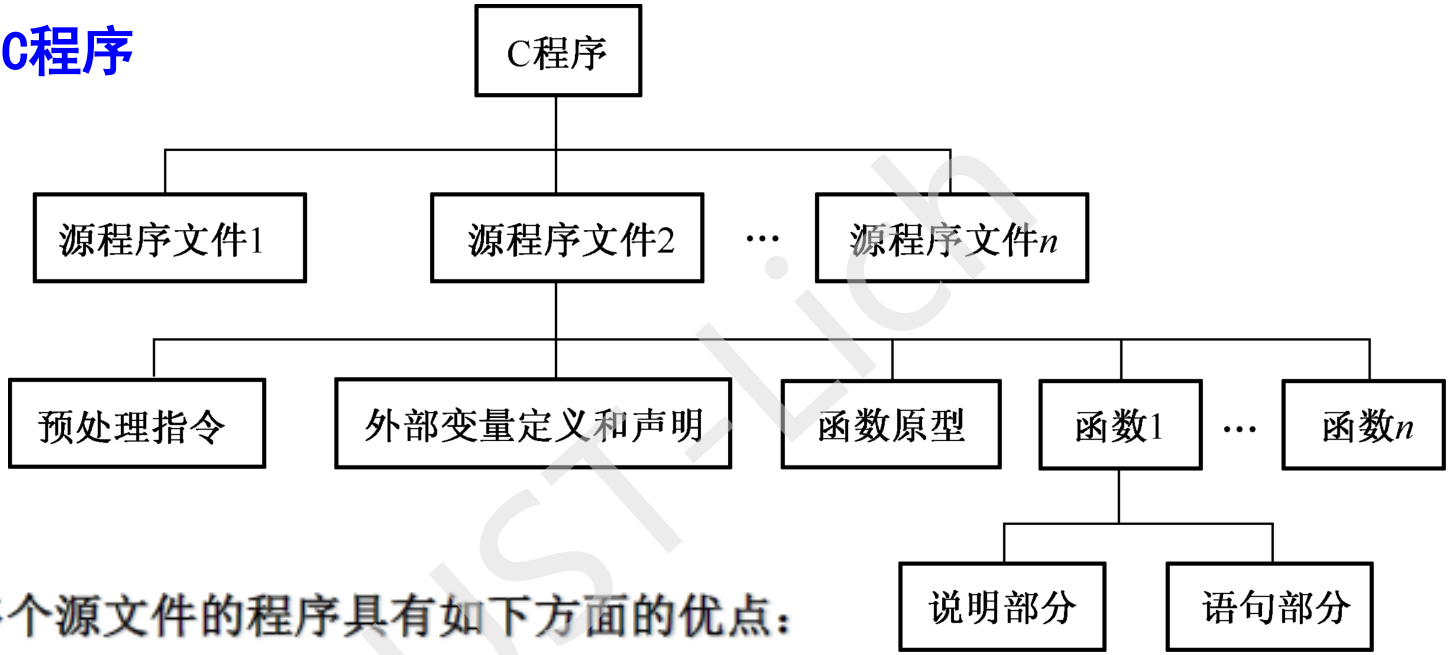
多文件的C程序



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。

C语言程序的结构

多文件的C程序



建立包含多个源文件的程序具有如下方面的优点：

(1) 能够提高软件的可重用性和良好的软件工程。

(2) 修改某个文件时不必重新编译其它文件。

(3) 便于多人分别编写与调试程序。

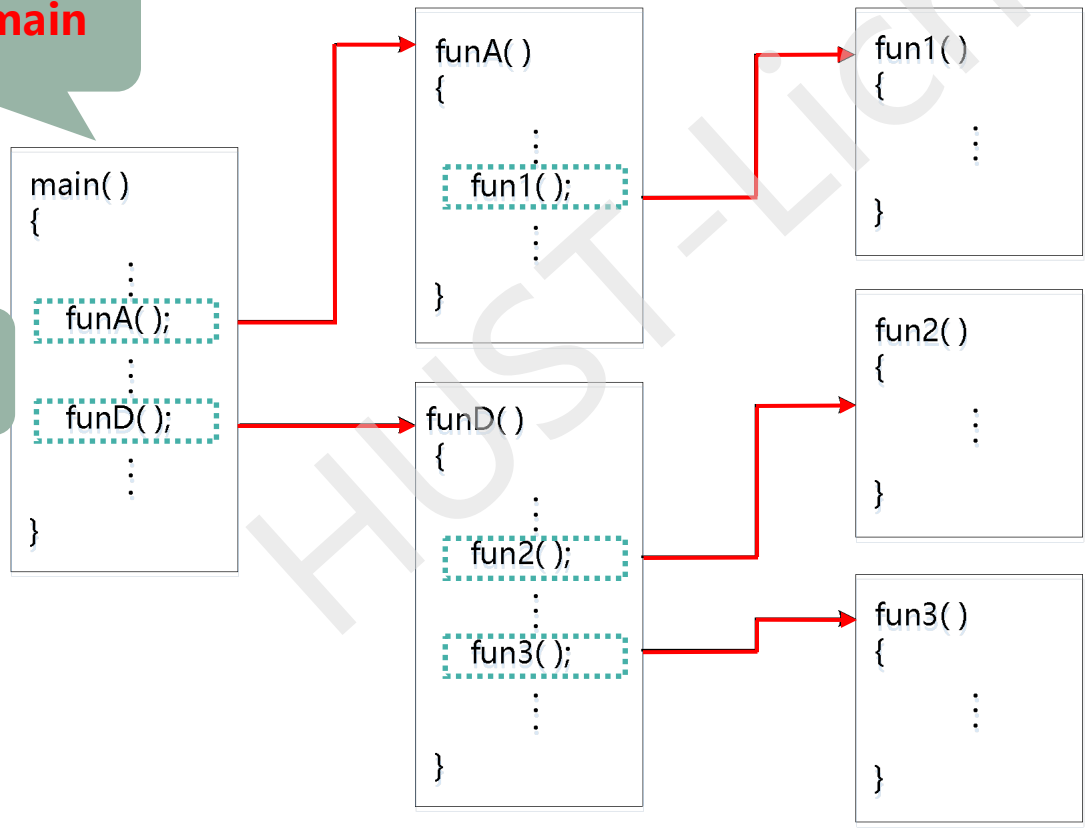
在IDE环境下建立多文件程序，其方法是创建一个工程文件，并使用静态函数和外部函数声明。

C语言程序的结构

函数调用关系

有且仅有一个入口函数称为主函数main()

函数调用语句



程序的执行总是从主函数开始。主函数中的所有语句按先后顺序执行完，则程序执行结束。

例 模块化设计

编写一个显示从整数1到10的2~5幂表的程序

分析：问题可以分解成如下子问题，每个子问题单独设计成函数；之后，这些函数通过main函数调用方式，完成问题的实现。

(1)显示标题。prn_banner()

(2)显示各列上部的标题部分。prn_headings()

(3)显示1到10的2至5次幂。其中，计算m^n。power(m,n)

```
**************************************
*            A TABLE OF POWE         *
**************************************
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
```

例 模块化设计

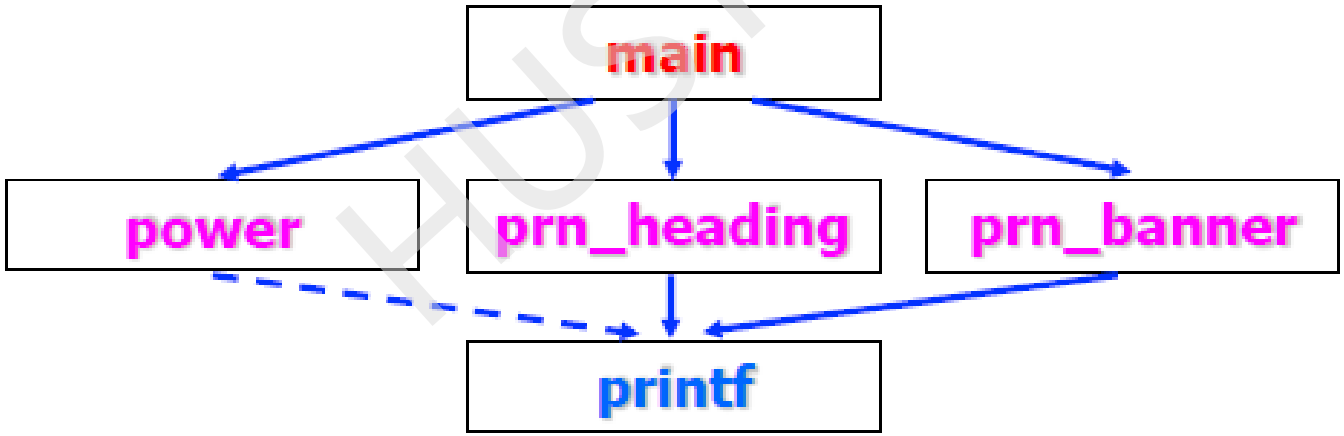
编写一个显示从整数1到10的2~5幂表的程序

分析：问题可以分解成如下子问题，每个子问题单独设计成函数；之后，这些函数通过main函数调用方式，完成问题的实现。

(1)显示标题。prn_banner()

(2)显示各列上部的标题部分。prn_headings()

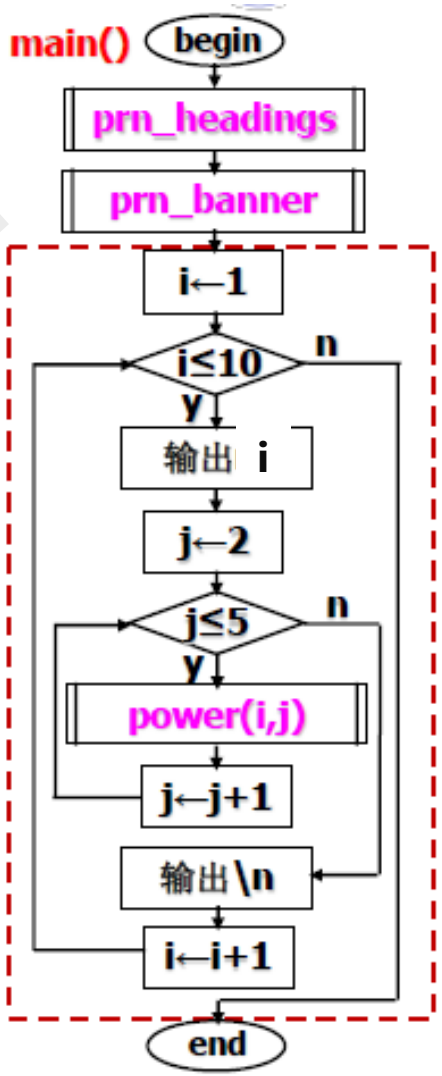
(3)显示1到10的2至5次幂。其中，计算m^n。power(m,n)



例 模块化设计

编写一个显示从整数1到10的2~5幂表的程序.

```
#include<stdio.h>
void  prn_banner(void);
void  prn_headings(void);
long  power(int x,int n);
int  main(void){
   int  i,j;
   prn_banner();  /*显示标题*/
   prn_headings();  /*显示表头*/
   for(i=1;i<=10;i++) {
      printf("%-8d",i);
      for(j=2;j<=5;j++)
         printf("%8-ld", power(i,j));
      printf("\n");
   }
   return 0;
}
```

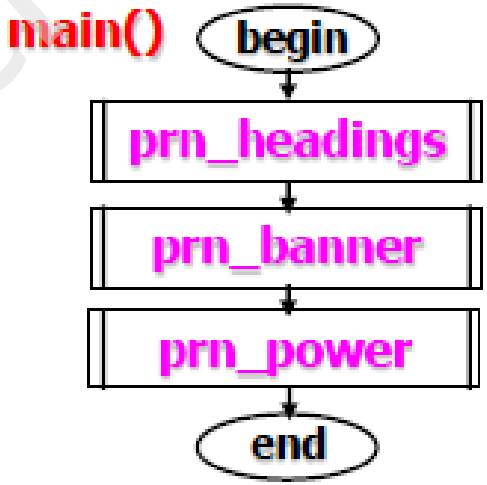


prn_power();

例 模块化设计

编写一个显示从整数1到10的2~5幂表的程序.

```
#include<stdio.h>
void  prn_banner(void);
void  prn_headings(void);
void  prn_power(void);
int  main(void){
   prn_banner() ;  /*显示标题*/
   prn_headings();  /*显示表头*/
   prn_power();
   return 0;
}
```



例5.2 猜数字游戏

计算机产生一个1到1000之间的随机数，游戏者猜直到正确为止，同时计算猜数的次数。为了帮助游戏者一步一步得到正确答案，程序会不断地发出信息“Too high”或“Too low”。最后，程序向游戏者显示游戏结果。

任务分解成以下两个子任务：

（1）计算机产生一个1到1000的随机数；

（2）游戏者猜数，直至猜对。

例 猜数字游戏

主程序结构

任务分解成以下两个子任务：

（1）计算机产生一个1到1000的随机数；

（2）游戏者猜数，直至猜对。

```
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);

猜数字游戏

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');
```

例 猜数字游戏

子任务1：GetNum( ) //产生一个1到1000的随机数

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。

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();                             /* 调用标准库函数rand产生一个随机数 */
    x= x % MAX_NUMBER + 1;                /* 将这个随机数限制在1～MAX_NUMBER之间 */
    return(x);
}
```

rand是接口stdlib.h中的一个函数，
它返回一个<=RAND_MAX的正数，
RAND_MAX的值取决于系统。
windows：32767 (2^{15}-1)
Linux：2147483647 (2^{31}-1)

例 猜数字游戏

子任务2：GuessNum( ) //游戏者猜数，直至猜对

```
for(;;) {
    输入猜测的数
    if（猜对了） 结束
    else if（小了） 输出太小的提示
    else  输出太大的提示
}
```

```
for(;;) {
   输入猜测的数
   if（猜对了） 结束
   else if（小了） 输出太小的提示
   else  输出太大的提示
}
```

```c
void GuessNum(int x)
{
    int guess,counter = 0;
    for (;;)
    {
        printf("guess it: ");
        scanf("%d", &guess);
        counter++;                  /*   统计猜的次数   */
        if (guess == x)
        {                   /*   猜对    */
            printf("You guessed the number by %d times!\n\n", counter);
            return;      // break; 行吗
        }
        else if (guess < x) /*   猜小了   */
        {
            printf("Too low. Try again.\n");
        }
        else                        /*   猜大了   */
        {
            printf("Too high. Try again.\n");
        }
    }
}
```

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 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;
}
```

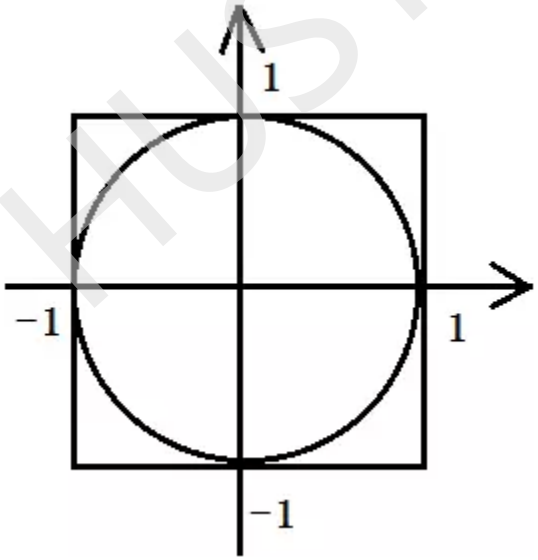
```
do {
        magic = GetNum( );   /* 产生
       GuessNum(magic);     /* 猜数
      printf("Play again? (Y/N) ");
      scanf("%1s", &cmd);
} while (cmd == 'y' || cmd == 'Y');
```

初始化语句的位置

```c
int main(void)
{
    char command;
    int magic;
    printf("This is a guessing game\n\n");
    srand(time(NULL)); /*用系统时间初始化随机数生成器 */
    do
    {
        magic = GetNum( ); /*调用GetNum产生随机数供猜测*/
        GuessNum(magic);   /* 调用GuessNum猜出这个数 */
        printf("Play again? (Y/N) ");
        scanf("%1s", &command);     /* 问是否继续 */
    } while (command == 'y' || command == 'Y');
    return 0;
}
```

练习----蒙特卡罗法求圆周率近似值

- 有一个如图所示正方形及其内切圆，往正方形内扔飞镖n次，当n足够大，比值“落在圆内的次数/落在正方形内的次数n”将无限接近“圆的面积/正方形的面积”，即π/4。n越大，π的近似结果越精确。



自定义函数

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定义函数：按语法规则编写完成指定任务的函数;

② 在调用函数之前要进行函数声明(有些情况);

③ 在需要使用函数时调用函数

5.2 函数的定义

函数定义的一般格式：

形式参数(简称形参)
无参数：void或空

```
类型名  函数名(参数列表)
{
    说明语句;
    执行语句;
}
```

函数头

函数体

无返回值：void

⚠ 函数名符合标识符构词规则；

⚠ 类型名是函数返回值的数据类型(简称函数类型)，没有返回值时使用关键字**void**；

⚠ 参数列表是说明函数参数的名称、类型和个数，如果没有参数时使用关键字**void** 或空。

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：无返回值函数　*/

  （2）return 表达式；/* 非void函数：有返回值函数 */

- 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。
- 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。
- return表达式值的类型，应与函数定义的返回值类型一致，如不同，则将把表达式值的类型转换为函数的类型 。

例 return语句

例： 写一个函数is_prime，判断整数n是否素数。如果n是素数，则返回1；如果n不是素数，则返回0。

是否结构化程序?

```
int is_prime(int n){
   int k,limit;
   if (n==2) return 1;
   if (!(n % 2)) return 0;
   limit=n/2 ;
   for (k=3;k<=limit;k+=2)
      if (!(n % k)) return 0;
   return 1;
}
```

模块结构化：单入口单出口流程。

《程序设计方法学》

建议：

1.避免使用goto；

2.尽量少用break，continue；

3.函数中尽量只有一个return语句

如何修改为结构化程序?

函数的声明：函数原型

以下两种情况需先声明函数原型，再调用函数，否则编译出错

- **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**

函数声明方法：

```
类型名 函数名（参数类型表）；  // 只包含函数头
```

```
long term(int i);   等价   long term(int);
```

⚠ 一个函数的函数定义与其函数原型，函数名、函数类型和参数表必须一致。

⚠ 函数原型中的参数可以缺省参数名，仅使用参数类型。

例 先声明后使用

```
int GetNum (void);          /* 函数原型 */
void GuessNum(int x)        /* 函数定义 */
{
  ......
}

int main(void)
{  ......
  do
   {
      magic = GetNum( );    /* 函数调用 */
      GuessNum(magic);      /* 函数调用 */
      ......
   } while (command == 'y' || command == 'Y');
   return 0;
}

int  GetNum( )     /* 函数定义 */
{
   ......
}
```

函数外声明

```
void GuessNum(int x)     /* 函数定义 */
{
  ......
}

int main(void)
{  ......
   int GetNum (void);          /* 函数原型 */
   void GuessNum(int) ; /* 函数原型，也可以 voi
   do {
      magic = GetNum( );    /* 函数调用 */
      GuessNum(magic);      /* 函数调用 */
      ......
   } while (command == 'y' || command == 'Y');
   return 0;
}

int  GetNum( )     /* 函数定义 */
{
   ......
}
```

函数内声明

两种声明方式，建议采用左侧的函数外声明方式

函数调用与参数传递

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

⚠ 实参必须与函数定义或原型中的形参之个数、次序和类型保持一致。

例如

```
putchar(c);          // 作为表达式语句
c=getchar();         // 作为表达式的操作数
s += term(i);
printf("%10.0f",sqrt(x)); // 作为实参
GuessNum(GetNum());  // 作为实参
```

函数调用的三种方式

函数调用的执行过程

遇到函数调用时，其执行过程如下：

(1) 传递参数
(2) 执行函数
(3) 返回调用处

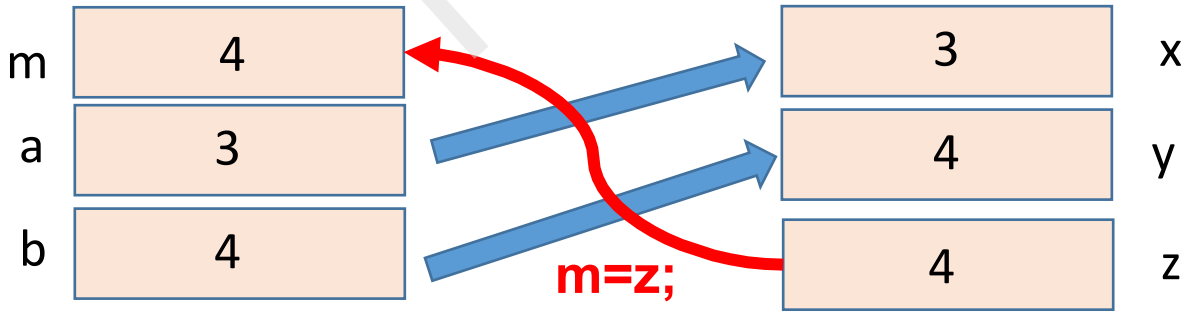
```
#include<stdio.h>
int max(int,int);   //函数声明
int main(void)
{
    int m,a=3,b=4;
    m = max(a,b); //函数调用
    printf("%d",m);
    return 0;
}
```

(1) 函数调用，实参初始化形参：
int x=a; int y=b;

(3) 返回调用处return z；m = z;

```
int max(int x,int y) //函数定义
{
    int z= (x>y) ? x: y;
    return z;
}
```

(1)　(2)　(3)



参数值的传递

函数调用时，将实参传送给形参称为参数传递。

在C中参数的传递方式是“值传递”，即计算每一个实参值，传递给相应的形参。

如果实参是一个变量x，形参也是x，则实参x和形参x各自有不同的存储单元，被调用函数不能改变实参x的值。

如果函数返回时需要将计算结果传回给主函数，如何实现呢？

方法一：函数返回值

方法二：通过参数传地址，如scanf(“%d”, &x);

方法三：用全局变量

例 值传递示例

“值传递”，即单向传递。在内存中实参、形参分占不同的单元。

例 下面的程序说明了值传递概念。

```
#include<stdio.h>
long fac(int);

int main(void){
   int  n=4;
   printf("%d\n",n);                  /* 输出4 */    ①
   printf("%ld\n",fac(n));            /* 输出24 */   ③
   printf("%d\n",n);                  /* 输出4 */    ④
}

long fac(int n){      /* 计算n的阶乘 */
   long  f = 1;
   for( ; n>0 ; --n ) f *= n ;  /* 函数main中的n值未改变 */
   printf("%d\n", n);                 /* 输出0 */ ②
   return(f);
}
```

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 .**

传地址（引用）调用

- 将变量的地址值传递给函数，函数既可以调用，也可以改变该变量的值。
- 标准库函数scanf就是一个引用调用的例子。
- int x;

  scanf("%d",&x);

5.3 变量的存储类型

C语言中的变量具有两种属性：

（1）根据变量所持有数据的性质不同而分为各种数据类型。

（2）根据变量的存储方式不同而分为各种存储类型。

变量的数据类型决定了该变量所占内存单元的大小及形式；

变量的存储类型规定了该变量所在的存储区域，因而规定了该变量作用时间的长短，即寿命的长短，这种性质又称为"存在性"。

变量在程序中说明的位置决定了该变量的作用域，即在什么范围内可以引用该变量，"可引用"又称为"可见"，所以这种性质又称为"可见性"。

——几个基本概念

- **作用域**：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。
- **变量的生存期**：变量从定义开始到它所占有的存储空间被系统收回为止的这段时间。
- **变量的可见性**：在某个程序段，可以对变量进行访问操作，则称该变量在该区域为可见的，否则为不可见的。**多重嵌套时，同名外层变量不可见。**
- **按照作用域，变量分为：**

  局部变量：在一个函数内部或复合语句内部定义的变量。

  全局变量：在任何函数外定义的变量，又称外部变量。

局部变量和全局变量

- **局部变量**：在函数内部定义的变量，其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;      // p、q为全局变量
float  f1( int  a)  // 形参a为局部变量
{  int  b,c;        // b、c为局部变量
   .....
}
```

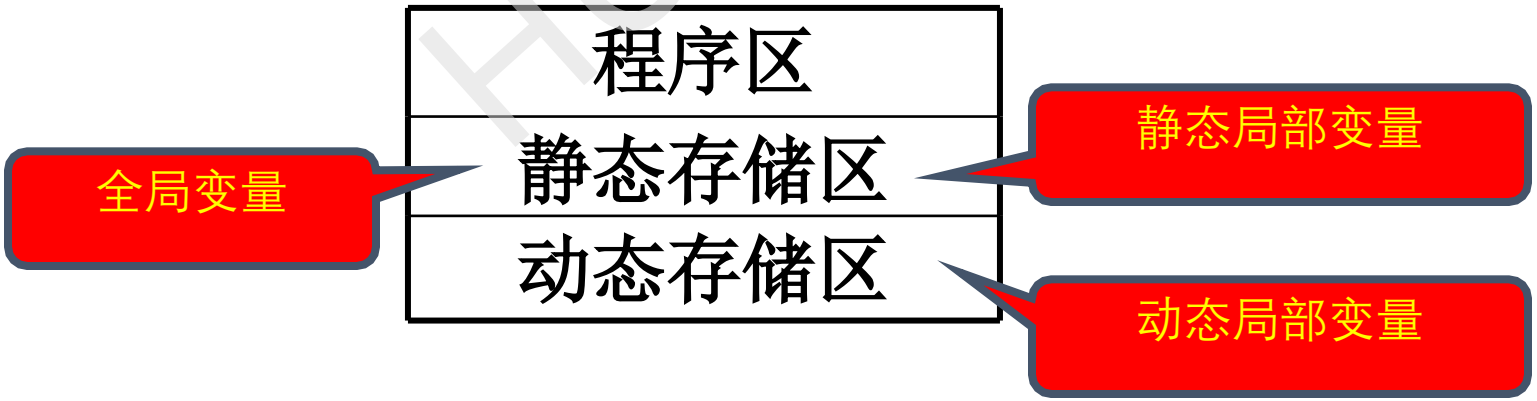
——几个基本概念

动态存储的变量、静态存储的变量

静态存储：在程序运行期间有固定的存储空间，直到程序运行结束。永久性占用内存　（全局变量属于静态存储）

动态存储：在程序运行期间根据需要分配存储空间，所在块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。临时动态分配存储空间

（局部变量有动态存储和静态存储）



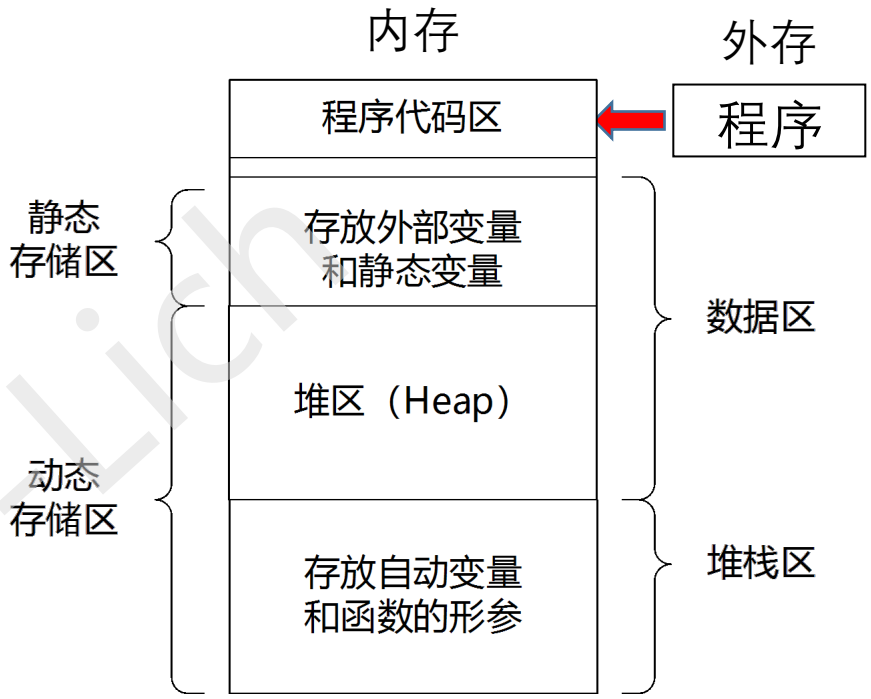
——使用内存的情况

一个正在运行的程序可将其使用内存的情况分为如下三类:

程序代码区：存放程序的指令代码。

静态存储区：静态存储变量存放区，包括全局变量和静态变量。

动态存储区：存放局部变量、函数形参、函数调用时的现场保护和返回地址等。



定义四种存储类型的关键字：

auto(自动变量)、**extern**(外部变量)

static(静态变量)、**register**(寄存器变量)

变量的存储类型决定：

作用域

存储分配方式

生命周期

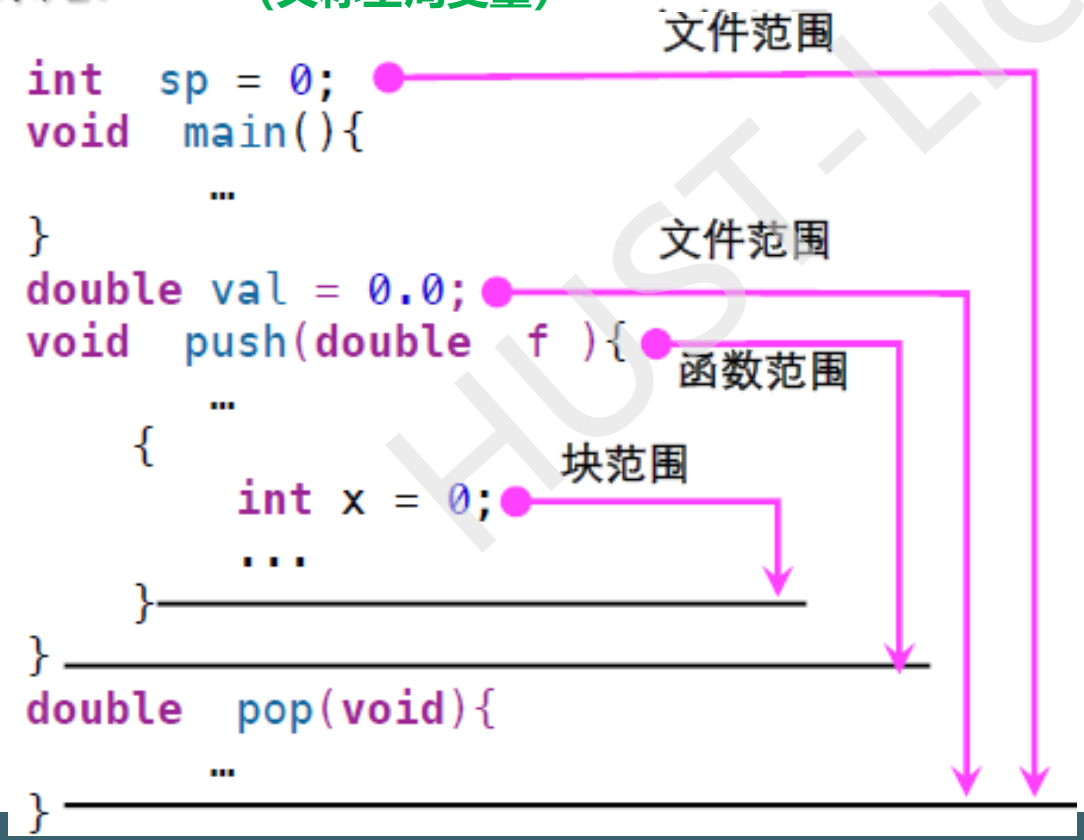
初始化方式

——作用域

程序中可以使用标识符的程序区域，称为标识符的作用域。

在函数内定义的变量称为局部变量，其作用域是定义该变量的程序块。

在函数外定义的变量称为外部变量，其作用域从其定义处开始一直到其所在文件的末尾。 （又称全局变量）



1. 表达式范围
2. 块范围
3. 函数范围
4. 文件范围

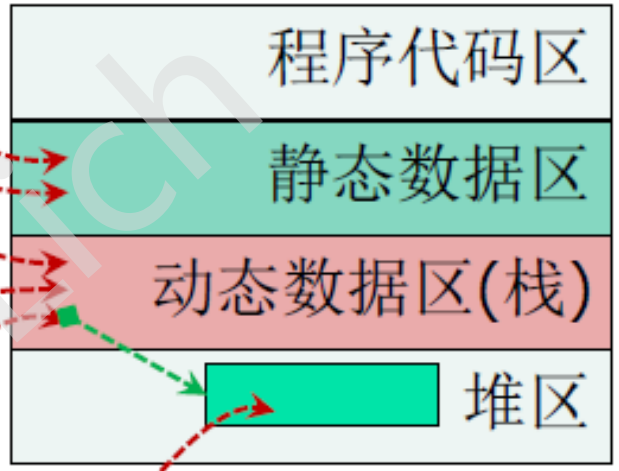
同名覆盖：

内层块变量与外层同名时，外层变量暂时失去可见性。

存储类型：存储区域

```
int   sp = 0;
void  main(){
        …
}
double val = 0.0;
void  push(double  f){
        …
    {
        int x = 0;
        ...
    }
}
double  pop(void){
        …
}
int  creatnode(void){
        …
    int *p = (int *)malloc(sizeof(int)*8)
}
```

全局变量属于静态存储



局部变量有动态和静态存储

动态存储区的变量，其作用域结束后立即释放空间

自动变量

- 局部变量的缺省存储类型是auto，又称自动变量

  auto int a; /*等价于int a; 也等价于auto a; */

- 作用域：局限于定义它的块。
- 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：定义该变量时，若没有显示初始化，其初值是不确定的；若显式初始化，则每次进入块时都要执行一次赋初值操作。

自动变量

例 自动型变量的作用域

```
#include <stdio.h>
 void  main( )
{
    int x=5;                          //等价于auto  int x=5;   (1)
    printf("x=%d\t",x);
    if(x>0)
    {
      int x=10;                       //等价于 auto int x=10;  (2)
      printf("x=%d\t",x);
    } // x所在块结束后立即释放x空间

    printf("x=%d\n",x+2);
}//释放外层变量 x 空间
```

运行结果：

x=5 x=10 x=7

自动型变量

例 下面的例子说明了自动变量的特性

```
#include<stdio.h>
void func( )
{
    auto int a = 0;
    printf(" a of func( ) = %d\n",++a);
}
int main( )
{
    int a = 10 ;
    func( );
    printf(" a of main( ) = %d\n",++a);
    func( );
    func( );
    printf(" a of main( ) = %d\n",++a);
    return 0;
}
```

调用func( )函数

函数在程序中被多次调用，函数内定义的局部变量每次都会被重新初始化，每次初始化时分配的存储单元有可能不同

运行结果：

```
a of func( )=1
a of main( )=11
a of func( )=1
a of func( )=1
a of main( )=12
```

自动型变量

例 下面的程序说明自动变量的初始化和作用域

```
#include<stdio.h>
int n;      /* 外部变量,等价于int  n =0 */
void show( );
void show( )
{
  auto int i=3;
  n++;
  i++;
  printf("the value: n=%d  i=%d\n", n, i);
  {
      auto int  i=10;
      i++;
      printf("now the value i=%d \n",i);
   }
  printf("then the value i=%d\n",i);
}
```

```
int main( )
{
    auto int i;
    auto int n=1;
    printf("at first n=%d\n",n);
    for(i=1 ; i<3 ; i++)
    {
        show( );
     }
    printf("at last n=%d",n);
    return 0;
}
```

运行结果：

```
at first n=1
the value: n=1  i=4
now the value i=11
then the value i=4
input the value: n=2  i=4
now the value i=11
then the value i=4
at last n=1
```

外部变量

- 外部变量是全局变量，但定义时不用存储类关键字。
- 作用域：从定义之后直到该源文件结束的所有函数，若用extern做声明，外部变量的作用域可以扩大到整个程序的所有源文件。
- 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：存在于整个程序的执行期间。
- 初始化方式：定义时若没有显式初始化，初值默认为0；若有显式初始化，只执行一次赋初值操作。

extern声明

```
srand(int x)
{
   seed=x;
}
int seed=10;   // seed作用域开始
int rand( )
{
   seed =（A*seed+B）% M ;
   return seed;
}
```

引用出错：无定义
引用在前，定义在后

外部变量的定义
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=10;
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

定义性声明和引用性声明

- 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
  - ✓ 可出现多次
  - ✓ 不能初始化

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

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，函数间通过形参进行通信更好。这样有助于提高函数的通用性，降低副作用。

/*file1.cpp*/

```
int maxmin(int x,int y);
extern int min,max;//引用性声明
int main (void)
{
    maxmin (4 , 1) ;
    printf("The max is %d\n",max);
    printf("The min is %d\n",min);
   return 0;
}
```

/*file2.cpp*/

```
int  min, max;//定义性声明
void  maxmin (int x, int  y)
{
    int m;
    min=(x<y)?x : y;
    max= (x>y)? x : y ;
    return;
}
```

外部变量的使用

外部变量的引用性声明

```
/*f1.cpp*/
#include “f3.cpp”
extern int x;
int main( )
{
  x++;
 printf("f1:x=%d",x);
 return 0;
}
```

```
/*f2.cpp*/
#include “f3.cpp”
extern int x;
void fun1 ( )
{
  x+=3;
 }
```

```
/*f3.cpp*/
int x=0;
void fun2( )
{
  printf("f3:x=%d",x);
 }
```

外部变量的定义

file1.cpp和file2.cpp中的extern int x； 告诉编译程序X是外部参照变量，应在本文件之外去寻找它的定义。所以上面的x虽在两个源文件中，但它们是同一个变量。在文件之外的file3.cpp中，定义了int x=0，即为它们调用的变量。

外部变量的使用

例 用extern声明外部变量

文件f1.cpp中的内容为:

```
#include <stdio.h>
#include "f2.cpp"
int a;
int m;
extern int power( );  // 外部函数可不引用
int main( )
{
   int b=3,c,d;
   printf("input the number a and its\
          power m:\n");
   scanf("%d,%d",&a,&m);
   c = a*b;
   printf("%d*%d=%d\n",a,b,c);
   d = power();
   printf("%d**%d=%d",a,m,d);
   return 0;
}
```

通过外部变量a、m传递数据

文件f2.cpp中的内容为:

```
extern int a;
extern int m;
int power( )
{
    int i,y=1;
    for ( i=1 ; i<=m ; i++)
    {
        y*=a;
    }
    return y;
}
```

本程序的作用是给定b的值，输入a和m，求a*b，和a的值。

静态变量

静态变量：关键字static修饰的变量, 如

```
static int k=0;
```

存储分配方式: 静态分配, 只初始化1次

生命周期: 永久的，

缺省初值: 0

两种static变量：

(1) 静态局部变量，用于定义局部变量

(2) 静态外部变量，用于定义外部变量。

两者作用域不同

静态局部变量

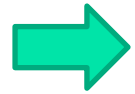
- 作用域：与自动变量一样
- 编程计算1! + 2! + 3! + 4! + … + n!
- 要求：将求阶乘定义成函数，使计算量最小。

```
long fac(int n){
    long  f = 1;
    for( ; n>0 ; --n ) f *= n ;
    printf("%d\n", n);
    return(f);
}
```

静态局部变量

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

```
long fac(int n){
    long  f = 1;
    for( ; n>0 ; --n ) f *= n ;
    printf("%d\n", n);
    return(f);
}
```



```
long  fac(int  n)
{
  static  long f=1;
  f *=n;
  return f;
}
```

两个函数的功能区别？

静态局部变量

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

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

seed在此无作用，
用户无法改变种子参数

静态局部变量
存于静态区，
初始化只执行1次

可以产生随机系列

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
作用域：限于定义它的文件，其它文件中不能使用

静态外部变量

与外部变量的区别：作用域不同。

- 静态外部变量只能在定义它的文件中使用，其它文件中的函数不能使用，
- 外部变量用extern声明后可以在其它文件中使用

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，可按需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。

- 一个文件中的静态外部变量会屏蔽来自其他文件的同名的外部变量。在静态外部变量所在的文件中，同名的外部变量不可见。

寄存器变量

- 寄存器变量：用register定义的局部变量
- 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。

静态变量

例 考察静态变量的值

```
#include <stdio.h>
int a = 2;   /* 全局变量*/
int f( )
{
      auto int b=0;
      static int c=3; /* 静态局部变量
*/
      b++;
      c++;
      return(a+b+c) ;
 }
void main( )
{
     int i;
     for(i=0;i<3;i++)
     {
          printf("%d\n",f());
      }
 }
```

静态数据区

a 2
c 3

动态数据区

i	main()
b 0	f()

运行结果：

7

8

9

静态变量

例 说明外部静态变量和外部变量的区别

```
//file1.cpp
#include<stdio.h>
#include  "file2.cpp"
extern int f2( );  //外部变量引用声明
static int x;  // 静态外部变量
int y ;        //全局变量
int f1( )
{
    return(x*x);
}
int main( )
{
    x=5;
    y=2;
    printf("f1=%d,\n f2=%d\n", f1( ), f2( ));
    return 0;
}
```

```
extern int y;  //外部变量引用声明
float f2( )
{
 return(y*y);
}
```

静态外部变量的作用域仅限于当前文件!

在file1.cpp 中定义的外部静态变量 x，在file2.cpp中是无效的!

存储类型小结

变量类型	存储类型	初值	可作用域	存储区	生命周期
局部变量	auto	无	当前定义块	动态数据区	小于程序运行期
外部变量 (全局变量)	extern	0	1）当前定义文件，从定义开始 2）通过extern声明到所有文件	静态数据区	等于程序运行期
静态局部变量	static	0	当前定义块	静态数据区	等于程序运行期
静态外部变量	static	0	当前定义文件，从定义开始	静态数据区	等于程序运行期
寄存器变量	register	无	当前定义块	动态数据区/register	小于程序运行期

对于函数，extern和static决定的是函数的作用域。

static函数作用域：定义该函数的文件，不可作用于其它文件

函数在定义时也可以使用extern

extern函数通过extern声明作用到其它文件或者函数（可省略extern）

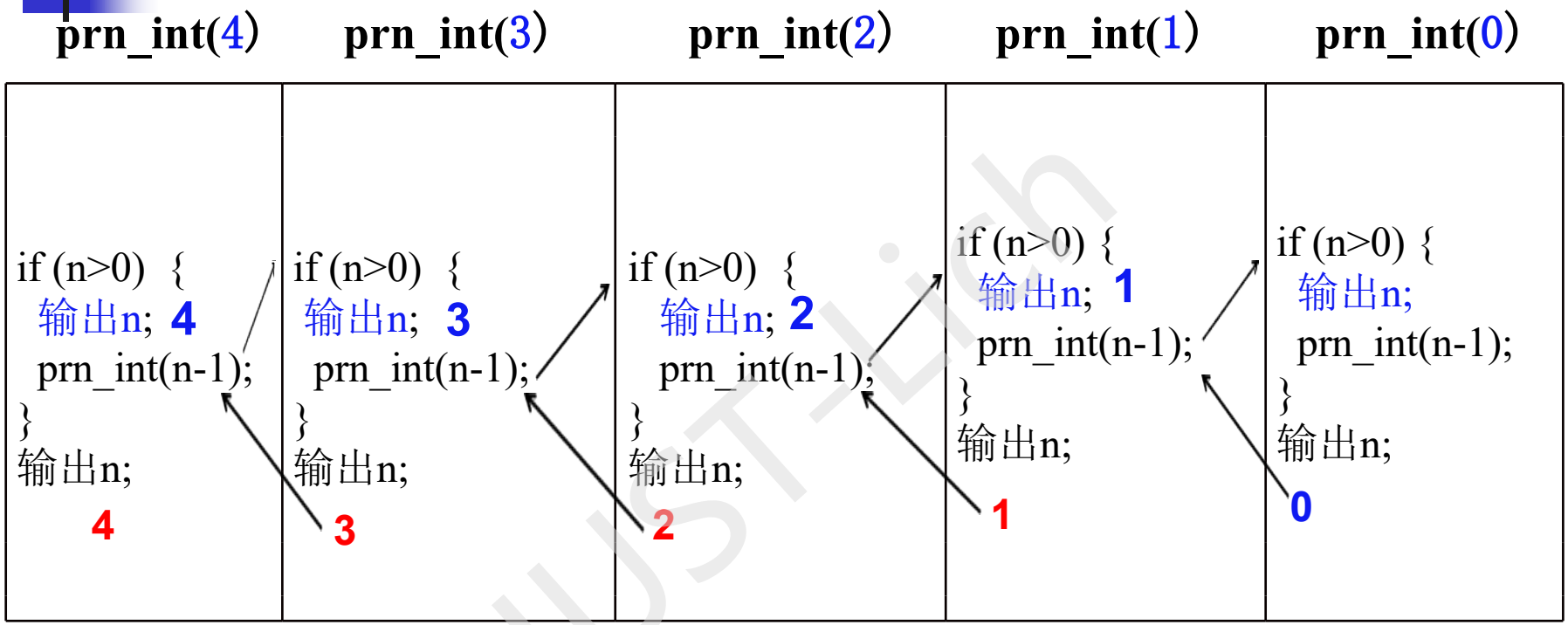
5.4 递归函数

```
void  prn_int(int n)
{
   if (n>0)  {
      printf ("%d ",n);
      prn_int(n-1);
    }
    printf ("%d ",n);
 }
int main(void)
{
   prn_int(4);
   return 0;
 }
```

递归函数：
在定义中含有递归调用

递归调用：调用自己

执行过程




```
void  prn_int(int n)
{
   if (n>0)  {
      printf ("%d ",n);
      prn_int(n-1);
    }
    printf ("%d ",n);
 }
```

```
int main(void)
{
   prn_int(4);
   return 0;
}
```

输出：

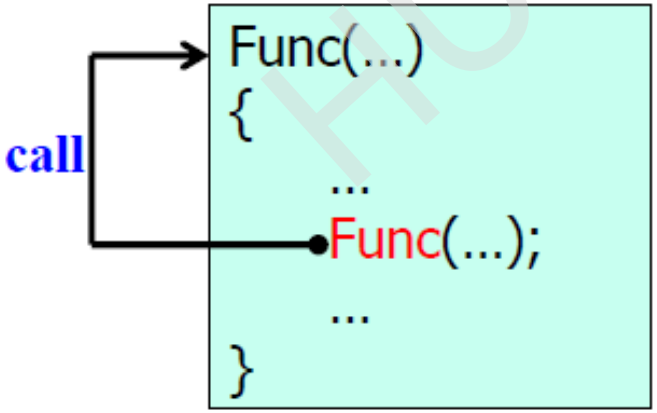
4 3 2 1 0 1 2 3 4

5.4 递归概述

递归函数，又称为自调用函数，即：在函数内部直接或间接地自已调用自已。

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。递归过程就是使问题空间逐步缩小的过程。

在递归函数中，由于存在着自调用的过程，故程序控制将反复地进入它的函数体。为了防止自调用过程无休止地继续下去，在函数内必须设置某种条件终止调用过程，并使程序控制逐步从函数中返回。



递归结束条件称为递归出口

递归定义中必须存在递归出口

递归法求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！。

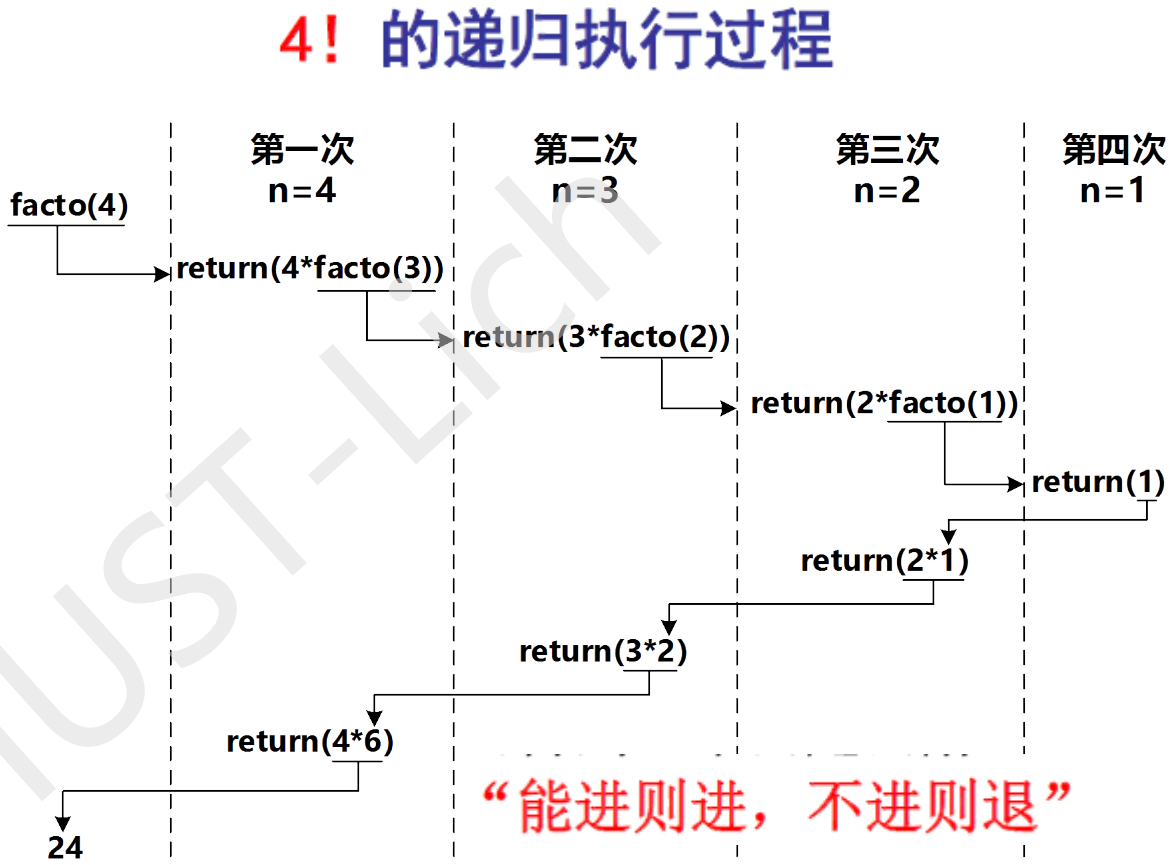
n！的递归实现

```
/*************************************************
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 fact(int n)
{
  if (n==0||n==1)          /*  递归结束条件 */
          return 1;
  else
          return ( n*fact(n-1) );  /* 递归调用 */
}
```

递归函数

例 阶乘的递归函数

```
long facto(int n)
{
    if(n == 1||n==0)
            //递归条件结束
      return 1;
    else
      return(n*facto(n - 1));
                  //递归调用
}
```



执行特点：

递次调用，直到递归终止条件；之后沿路回归。

n! 的两种实现方式

递归：
运行效率低、计算开销大、占用空间多；
结构紧凑、逻辑清晰、可读性强、代码简洁

```
/*********************************************
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 fact(int n)
{
 if (n==0||n==1)        /* 递归结束条件 */
        return 1;
 else
        return (n*fact(n-1)); /* 递归调用 */
}
```

实际项目中，为提高效率，
应该优先选择迭代

```
/* 迭代法计算n！ */
long factorial_iteration( int n )
{
   int result = 1;
   while( n>1 ) {
      result *=n;
      n--;
   }
   return result;
}
```

什么情况下使用递归呢？

用递归能容易编写和维护代码，且运行效率并不至关重要时

编写递归的两个要点 P142-P143

（1）能找到正确的递归算法（每次调用在规模上有所缩小）

（2）能确定递归的结束条件

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。

注意：

（1）每次递归调用都有一次返回。

（2）每一次递归调用都使用自己的私有变量，包括函数参数、函数内的局部变量！

递归求Fibonacci数列的第n项

```
/*  Recursive fibonacci : Compute the n item  */
long long fibo (long n )
{
   if ( n == 1 || n == 2)  return 1;
   else
       return  fibo(n-1)+fibo(n-2);
 }
```

$a_n=a_{n-1}+a_{n-2}$ $(a_1=1,a_2=1)$

这种递归方式有损于运行效率——如何优化？

优化1---- 记忆化搜索

$a_n=a_{n-1}+a_{n-2}$ $(a_1=1,a_2=1)$

```
long long fibo (long n )
{
    static long long f[1000] = {0,1,1};
                /* 用来储存每一个计算过的值, 第n项值储存于 f[n]中。 */
   if( f[n] )    /* 已计算过,直接返回，不算 */
   {
      return f[n];
   }
   f[n] = fibo(n-1) + fibo(n-2);
   return f[n];
}
```

1,1,2,3,5,8,13,…

优化2---- 不重复求解

$a_n=a_{n-1}+a_{n-2}$ $(a_1=1,a_2=1)$

1,1,2,3,5,8,13,…

```
long long fibo(long n)
{
    static long long a=1;  //  前二项
    long long b,c;       //  b: 前一项，c：当前项
    if(n ==1 || n== 2)
    {
        return 1;
    }
    b=fibo(n-1) ;  // 计算前一项
    c=b+a;          // 计算当前第n项
    a=b;           // 前一项成为前二项
    return c;  //   当前项成为前一项
}
```

优化3---- 尾递归

```
long long  fibo_tail_rec(int n, long long first, long long second)
{
   if (n <= 1)     return first;
  else
       return fibo_tail_rec(n-1,second,first+second);
}
long long fibo (long n )
{
    fibo_tail_rec(n,1,1);
}
```

$a_n=a_{n-1}+a_{n-2}$ $(a_1=1,a_2=1)$

1,1,2,3,5,8,13,…

函数的最后一步是调用自己，精髓：尾递归就是把当前的运算结果放在参数里传给下层函数，**避免爆栈**.

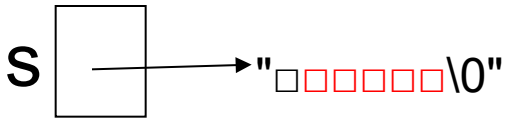
递归函数设计

- 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【例5.12】

- 字符串是以空字符（'\0'）结尾的字符序列
- 可以把字符串看成：一个字符后面再跟一个字符串或者空串
- 这是递归定义，可以用递归的方法对一些基本的字符串处理函数进行编写。

递归实现标准库函数strlen(s)

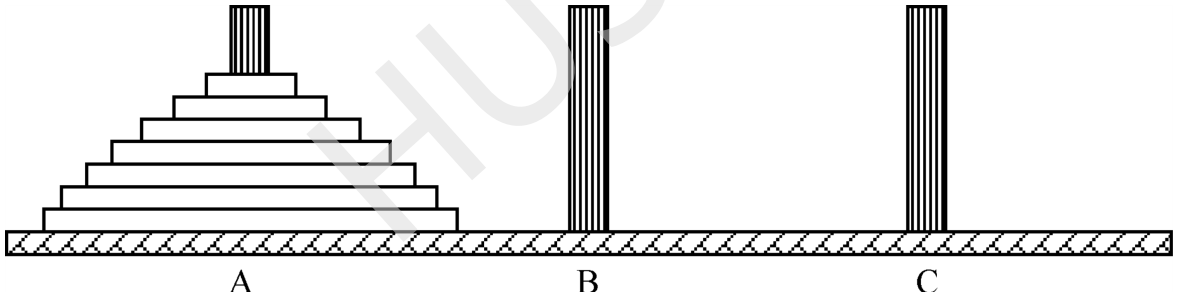



```
int strlen(char s[ ])
{
    if(s[0]=='\0')
        return 0;
    else
        return(1+strlen(s+1));
}
```

&s[1]

汉诺塔（游戏）问题 【例5.13】

- 问题描述：开始木桩A上有64个盘子，盘子大小不等，大的在下，小的在上。
- 游戏目标：把木桩A上的64个盘子都移到木桩C上
- 约束条件：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。



设计一个求解汉诺塔问题的算法，输出盘子在各个木桩之间的移动顺序。

汉诺塔问题的递归算法

move(n,a,b,c)

- 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。 move (n–1, a, c, b);

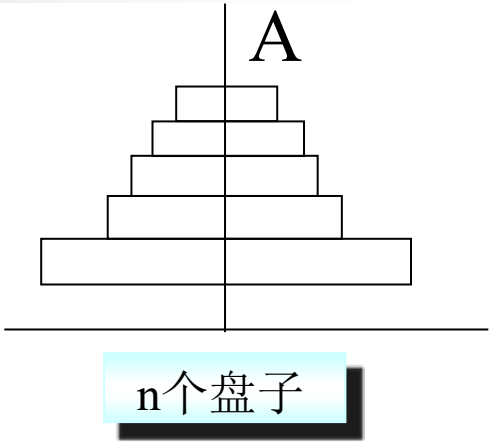
  （2）把A上剩下的盘子（即最大的那个）移到C。 a→ c

  （3）把B上的n-1个盘子借助A移到C。 move (n–1, b, a, 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 )
{      如何记录一共移动的次数呢？
  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);
   }
}
```



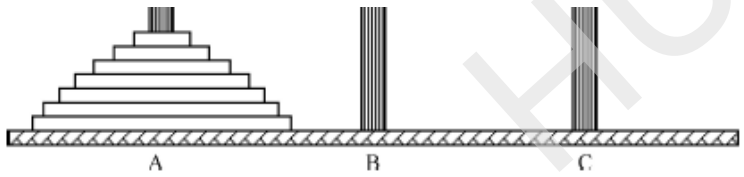
（1）把A上的n-1个盘子借助C移到B。

（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。

汉诺塔问题

```
void move(int n, int a, int b, int c)
{
    static int i=1;  //累计移动步骤
    if (n==1)   /* 递归结束条件：a上只有1个盘子 */
        printf("step %d: %c-->%c\n",i++,a,c);  /* 把a上盘子移到C */
    else
    {
        move(n-1,a,c,b);   /* 递归调用：把a上的n-1个盘子借助c移到b */
        printf("step %d: %c-->%c\n",i++,a,c);/* 把a上剩下的盘子移到C */
        move(n-1,b,a,c);    /* 递归调用：把b上的n-1个盘子借助a移到c */
    }
}
```



（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。

自学例【5.14】约瑟夫问题

递归函数设计

[程序说明] 以下函数climb求解“上楼梯问题”，假设楼梯有n阶台阶，上楼可以一步上1阶，也可以一步上2阶，求所有不同走法的数量。请将下面程序中划线处应该完善的内容填写划线处。

```
int climb(int n)
{
    if (n == 0||n == 1||n == 2)
            return n;
    else
            return climb(n-1) + climb(n-2);
}
```

递归练习

1、用递归实现reverse函数：将输入的非负长整数逆序输出。如reverse(123)，则输出321。函数reverse的原型为：

```
void reverse(long m);
```

2、正反序相同的字符串被称为“回文字符串”。例如LeveL就是一个回文字符串。使用递归实现is_palindrome函数：判断一个串是否回文，是，返回1；不是，返回0。函数is_palindrome的原型为：

```
int  is_palindrome(char str[ ],int n);
```

整数的分划问题

- 就是把正整数M写成一系列正整数之和的表达式。 注意，分划与顺序无关，即5+1和1+5是同一种分划。另外，M本身也算是一种分划。例如，M=6时，可以分划为：

  6

  5+1

  4+2， 4+1+1

  3+3， 3+2+1， 3+1+1+1

  2+2+2， 2+2+1+1， 2+1+1+1+1

  1+1+1+1+1+1

- 输入正整数m，输出m的所有分划。

递归原则

1.先设定分划m的第0位置的值（i），再从余下的m-i设定下一个位置的值，...，直至分划数为0。

2.由于分划与顺序无关，为避免重复，按照从大到小依次设定每一个分划值，即前面的值>=后面的值。

3. 将确定的分划值存入数组中

递归求解整数的分划问题

```
void split(int n, int cur ) // 将n从位置cur分划为一系列整数和
{
 if( !n)       /* 递归边界：已确定一种分划 */
       输出该分划方案
 else
       for(i = n; i >=1; i--)    /* 枚举 当前位置cur的分划值 */
      {
            if( i值可行）{
               a[cur]=i;          //  取 i 作为当前位置的分划值
              split(n-i,cur+1);    /* 递归确定下一位置的分划值*/
           }
       }
```

```
int a[100]={0};
void split(int n,int  cur)
{
    int i;
    if(n==0) out(cur);
    else
        for(i=n;i>=1;i--)/*依次选n至1作为当前位置的值*/
            if(cur==0||i<=a[cur-1])  {/*当前需要确定的是首元素,
                                       或者待选值不大于上一位置的值 */
                a[cur]=i;/*选i*/
                split(n-i,cur+1);/*递归确定下一位置的值*/
            }
}
```

递归练习

八皇后问题：在8X8格的棋盘上摆放8个皇后，使它们互不攻击，即任意2个皇后不能在同一行或同一列或同一对角线上。找出所有解。

int a[8];　记录可行解

a[i]表示第i行皇后放在第a[i]列

a={0,5,7,2,6,3,1,4}

从 k=0 开始递归穷举a[k]， 直至 k=8 结束

```
void find(int a[ ],int i )
{
 if(i==8)   /* 递归边界：已枚举每一行  */
   {    输出    }
 else
      for(j = 0; j <8; j++)   /* 将i行皇后放到 j 列 */
     {
            判断 是否互不攻击
            if( 互不攻击）{
               a[i]=j;
               find(a,i+1);   /* 枚举a[i+1] */
            }
      }
```

本章小结

本章需重点掌握的内容有：

- 了解C语言程序的结构特点与其优越性
- 熟练掌握函数定义与调用的方法
- 掌握变量的存储类型、变量的作用域
- 熟练掌握函数间数据传递的方法
- 了解递归函数的概念和使用方法
- 了解多文件C语言程序的构造

数据存储类型：

- auto
- extern
- static
- register

子程序数据交换方法：

- 函数值
- 参数传地址值
- 变量作用域