

6

函数

6.1 引言

6.2 函数定义的一般形式与函数原型声明

6.3 函数调用

6.4 函数的递归调用

6.5 变量存储空间

6.6 内部函数和外部函数

6.1 引言

模块化程序设计

基本思想：将一个大的程序按功能分割成一些小模块

特点：

各模块相对独立、功能单一、结构清晰、接口简单

控制了程序设计的复杂性

提高元件的可靠性

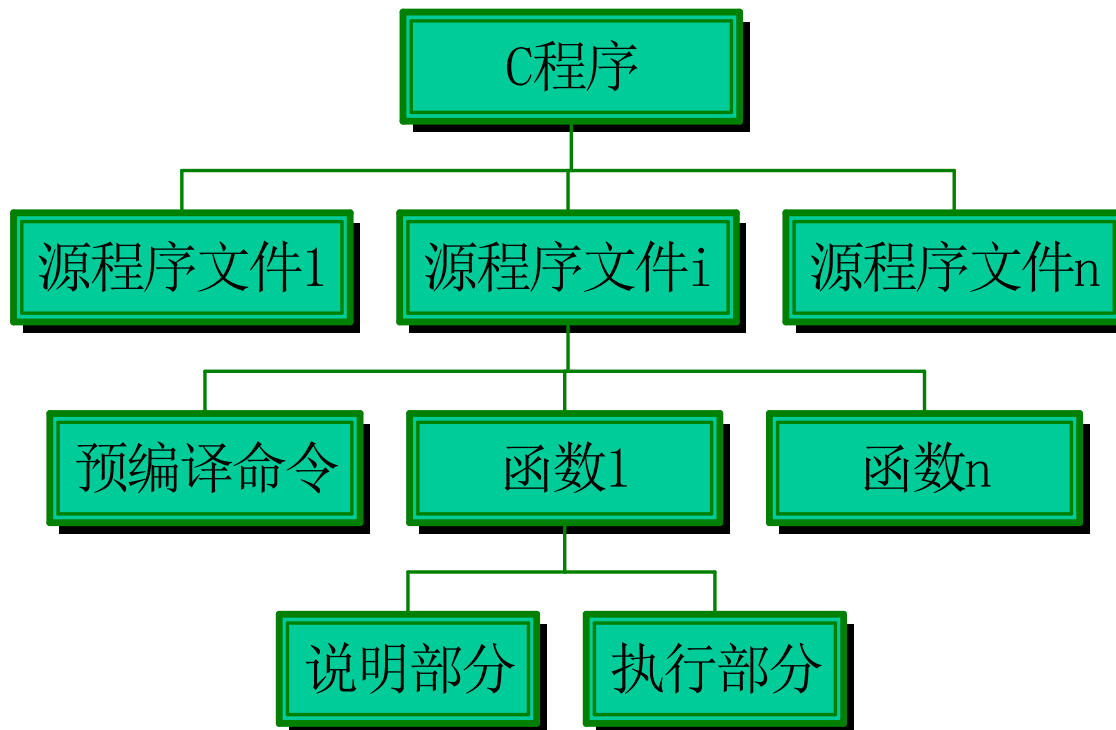
缩短开发周期

避免程序开发的重复劳动

易于维护和功能扩充

开发方法：自上向下、逐步分解、分而治之

- C是**函数式**语言
- 必须有且只能有一个名为**main**的主函数
- C程序的执行总是**从main函数开始，在main中结束**
- 函数**不能嵌套定义,可以嵌套调用**



6.2 函数定义的一般形式与函数原型声明

合法标识符

一般形式

函数类型 函数名 (形参类型说明表)
{

函数返回值类型
缺省int型
无返回值void

函数的操作对象 (数据) 定义说明部分
语句部分

函数体

```
int factorial(int n){ //计算n的阶乘
    int product,i;
    product=1;
    for(i=1;i<=n;i++)
        product*=i;
    return(product);
}
```

★ 函数声明

❖ 对被调用函数要求：

- 必须是已存在的函数
- 库函数: `#include <*.h>`
- 用户自定义函数: 函数类型说明

❖ 函数说明

- 一般形式：`函数类型 函数名(形参类型1 [形参名1],.....);`
或 `函数类型 函数名(形参类型1, 形参类型2,);`
- 作用：告诉编译系统函数类型、参数个数及类型，以便检验
- 函数定义与函数说明不同
- 函数说明位置：程序的数据说明部分（函数内或外）
- 被调用函数定义出现在主调函数之前，可不作函数说明
- 库函数声明包括在头文件（*.h）里，只需将头文件*.h作#include预处理即可。

例 函数说明举例

```
#include<stdio.h>
main()
{
    int a,b;
    int c;
    scanf("%d,%d",&a,&b);
    c=max(a,b);
    printf("Max is %d\n",c);
}
max(int x, int y)
{
    int z;
    z=x>y?x:y;
    return(z);
}
```

function declaration*/

函数出现在主调函数
前，不必函数说明

add();

int型函数可不作函数说明

6.3 函数调用

调用形式：函数名(实参表);

说明：

- 实参与形参个数相等，类型一致，按顺序一一对应
- 实参表求值顺序，因系统而定（VC++6.0自右向左）

【6.1】 计算并输出三个电阻的串联和并联值，分别由函数series()和parallel()实现

```
#include<stdio.h>
float series(float a1,float a2,float a3){    //计算串联阻值
    return(a1+a2+a3);
}
float parallel(float b1,float b2,float b3){    //计算并联阻值
    float rp,rr;
    rr=1.0/b1+1.0/b2+1.0/b3;
    rp=1.0/rr;
    return(rp);
}
int main()
{
    float r1,r2,r3,rs,rp;
    printf("Enter the value fo r1,r2,r3\n");
    scanf("%f%f%f",&r1,&r2,&r3);
    rs=series(r1,r2,r3);
    printf("The series values is %f\n",rs);
    rp=parallel(r1,r2,r3);
    printf("The parallel values is %f\n",rp);
}
```

例 参数求值顺序

```
#include<stdio.h>
main()
{
    int i=2,p1,p2;
    p1=f1(i,++i);
    p2=f2(i,i++);
    printf("p1=%d,p2=%d\n",p1,p2);
}

int f1(int a, int b)
{
    int c;
    if(a>b) c=1;
    else if(a==b) c=0;
    else c=-1;
    return(c);
}

int f2(int a, int b)
{
    int c;
    if(a>b) c=1;
    else if(a==b) c=0;
    else c=-1;
    return(c);
}
```

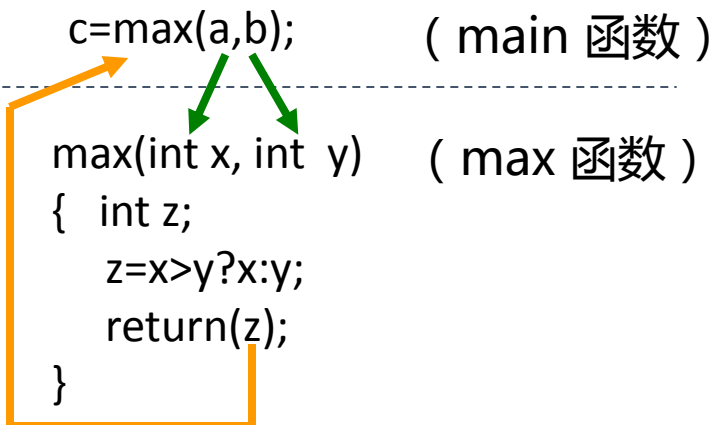
根据顺序不同会产生不同结果

函数参数及传递方式

★ 形参与实参

- ❖ 形式参数：定义函数时函数名后面括号中的变量名
- ❖ 实际参数：调用函数时函数名后面括号中的表达式

例 比较两个数并输出大者



```
main()
{
    int a,b,c;
    scanf("%d,%d",&a,&b);
    c=max(a,b);
    printf("Max is %d",c);
}

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

实参

形参

说明：

- 实参必须有确定的值，可以是**常量、变量或表达式**，但必须有**确定的值**
- 形参必须指定类型，各个形参之间要用**逗号**隔开
- 形参与实参**类型一致，个数相同**
- 若形参与实参类型不一致，自动按形参类型转换——**函数调用转换**
- 形参**不能在定义的同时进行初始化**
- 形参在函数**被调用前不占内存**，函数调用时为形参分配内存，**调用结束，内存释放**

值拷贝传递机制

【例6.3】 字符串加密程序encrypt.c的实现

```
#include<stdio.h>
#define STEPS 3
#define LEN_INFO 16

char encrypt(char origin){
    char i='A'+(origin+STEPS-'A')%26;
    return i;
}

int main()
{
    int i;
    char c, [LEN_INFO]="SIX AM,RUN WEST";
    printf("origin is:\t%s\n",s);
    printf("encrypt is:\t");
    for(i=0;i<LEN_INFO-1;i++){
        c=encrypt(s[i]);
        putchar(c);
    }
    printf("\n");
}
```

值传递方式

- 函数调用时,为形参分配单元,并将实参的值**复制**到形参中；调用结束，形参单元**被释放**，实参单元仍保留并维持原值

特点：

- 形参与实参占用**不同**的内存单元
- **“单向”** 传递

【例6.4】 交换两个数

```
#include <stdio.h>
main()
{
    int x=7,y=11;
    printf("x=%d,y=%d\n",x,y);
    printf("swapped:\n");
    swap(x,y);
    printf("x=%d,y=%d\n",x,y);
}
swap(int a,int b)
{
    int temp;
    temp=a; a=b; b=temp;
}
```

调用前：

x: 7 y: 11

调用：

x: 7 y: 11
↓ ↓
a: 7 b: 11

swap:

x: 7 y: 11

a: 11 ← b: 7
 ↘ ↗
 temp

调用结束：

x: 7 y: 11

地址传递方式

```
#include <stdio.h>
main()
{
    void swap(int);
    int a[2]={1,2};
    printf("a[0]=%d,a[1]=%d\n",a[0],a[1]);
    printf("swapped:\n");
    swap(a);
    printf("a[0]=%d,a[1]=%d\n",a[0],a[1]);
}
void swap(int b[])
{
    int temp;
    temp=b[0]; b[0]=b[1]; b[1]=temp;
}
```


地址传递方式

- 函数调用时，将数据的**存储地址**作为参数传递给形参

特点：

- 形参与实参占用**同样**的存储单元（此时形参本质是“**指针**”变量）
- **“双向”**传递
- 实参是地址常量或变量，形参是指针变量

函数的返回值

返回语句

形式： `return (表达式) ;`

`return 表达式 ;`

`return ;`

功能：使程序控制从被调用函数返回到调用函数中，同时把返回值带给调用函数.

说明：

- 函数中可有多多个return语句.
- 若无return语句，遇 `}` 时，自动返回调用函数.
- 若函数类型与return语句中表达式值的类型不一致，按前者为准，自动转换-
-----**函数调用转换**.
- **void型函数**.

6.4 函数的返回值

函数定义不可嵌套，但可以嵌套调用

递归：函数自己调用自己



吓得我抱起了

抱着抱着抱着我的小鲤鱼的我的我的我



```
#include <stdio.h>
void My_Small_Carp_Recursion(int depth)
{
    printf("抱着");
    if (depth==0) printf("我的小鲤鱼");
    else My_Small_Carp_Recursion(--depth);
    printf("的我");
}
int main()
{
    printf("吓得我抱起了\n");
    My_Small_Carp_Recursion(2);
    putchar('\n');
}
```

❖ 说明

- C编译系统对递归函数的自调用次数没有限制
- 每调用函数一次，在内存堆栈区分配空间，用于存放函数变量、返回值等信息，所以递归次数过多，可能引起堆栈溢出


例 求n的阶乘

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

```
#include <stdio.h>
int fac(int n)
{   int f;
    if(n<0) printf("n<0,data error!");
    else if(n==0 || n==1) f=1;
    else f=fac(n-1)*n;
    return(f);
}
main()
{   int n, y;
    printf("Input a integer number:");
    scanf("%d",&n);
    y=fac(n);
    printf("%d! =%15d",n,y);
}
```

例 编制一递归函数，将一个十进制正整数（如：15613）转换成八进制数形式输出。

思路：十进制整数转换成八进制整数的方法是**除8逆向取余**。

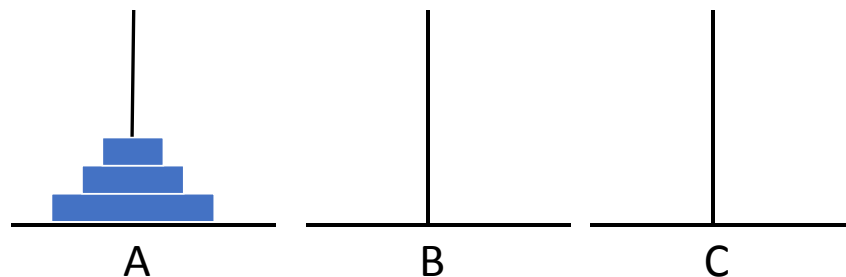
余数：		商：
$15613 \% 8 = 5$		$15613 / 8 = 1951$
$1951 \% 8 = 7$		$1951 / 8 = 243$
$243 \% 8 = 3$		$243 / 8 = 30$
$30 \% 8 = 6$		$30 / 8 = 3$
$3 \% 8 = 3$		$3 / 8 = 0$
结果： 36375		

```
#include <stdio.h>
void dtoo(int x)
{ int m;
  m=x%8;    x=x/8;
  if(x!=0) dtoo(x);
  printf("%d",m);
}
main( )
{ int num;
  printf("input num:");
  scanf("%d",&num);
  printf("\n%d的八进制是:", num);
  dtoo(num);
  printf("\n");
}
```


汉诺塔问题:

有三个柱和 n 个大小各不相同的盘子，开始时，所有盘子以塔状叠放在柱A上，要求按一定规则，将柱A上的所有盘子借助于柱B移动到柱C上。移动规则如下：

- (1) 一次只能移动一个盘子。
- (2) 任何时候不能把盘子放在比它小的盘子的上面。



汉诺塔问题递归过程的函数描述

有n个盘子的汉诺塔问题的函数：hanoi(n,'A','B','C');

当 $n=1$ 时, 直接从 A 移到 C, 问题结束。移动过程用如下函数描述：

move('A','C');

若 $n>1$ 时, 则必须经过如下三个步骤：

第一步：按照移动规则，把A上面的 $n-1$ 个盘子，移到B，此时C为中间柱。

hanoi(n-1,'A','C','B');

第二步：将A上仅有的一只盘子（当前最大的一只）直接移到柱B上。

move('A','C');

第三步：用第一步所述方法, 将B柱上的 $n-1$ 个盘子移到C柱上，此时A为中间柱。

hanoi(n-1,'B','A','C');

```
void move(char getone, char putone)
{
    printf("%c--->%c\n", getone, putone);
}

void hanoi(int n, char one, char two, char three)
{
    if(n==1)    move(one, three);
    else
    {
        hanoi(n-1, one, three, two);
        move(one, three);
        hanoi(n-1, two, one, three);
    }
}

main()
{
    int m;
    printf("Input the number of disks:");
    scanf("%d", &m);
    printf("The steps to moving %3d disks:\n", m);
    hanoi(m, 'A', 'B', 'C');
}
```

6.5 变量存储空间

变量是对程序中数据的存储空间的抽象

❖ 变量的属性

- 数据类型：变量所持有的数据的性质（操作属性）
- 存储属性
 - 存储器类型：寄存器、静态存储区、动态存储区
 - **生存期**：变量在某一时刻存在-----静态变量与动态变量
 - **作用域**：变量在某区域内有效-----局部变量与全局变量

❖ 变量的存储类型

- auto ----- 自动型
- register ----- 寄存器型
- static ----- 静态型
- extern ----- 外部型

❖ 变量定义格式：[存储类型] 数据类型 变量表;

局部变量与全局变量

局部变量（内部变量）

定义：**在函数内定义，只在本函数内**

说明：

- main中定义的变量只在main中有效
- **不同函数中同名变量，占不同内存**
- 形参属于局部变量
- 可定义在复合语句中有效的变量
- 局部变量可用存储类型：**auto**

例 不同函数中同名变量

```
main()
{   int a,b;
    a=3;
    b=4;
    printf("main:a=%d,b=%d\n",a,b);
    sub();
    printf("main:a=%d,b=%d\n",a,b);
}

sub()
{   int a,b;
    a=6;
    b=7;
    printf("sub:a=%d,b=%d\n",a,b);
}
```

运行结果：
main:a=3,b=4
sub:a=6,b=7
main:a=3,b=4

全局变量---外部变量

定义：**在函数外定义，可为本文件所有函数共用**

说明：

- 有效范围：从**定义变量的位置开始到本源文件结束**，及有**extern**说明的其它源文件
- 外部变量说明：extern 数据类型 变量表；
- 外部变量定义与外部变量说明不同

应尽量少使用全局变量，因为：

- ☆ 全局变量在程序全部执行过程中占用存储单元
- ☆ 降低了函数的通用性、可靠性，可移植性
- ☆ 降低程序清晰性，容易出错

```
float  max,min;
float  average(float  array[], int n)
{
    int i; float  sum=array[0];
    max=min=array[0];
    for(i=1;i<n;i++)
    {
        if(array[i]>max) max=array[i];
        else if(array[i]<min) min=array[i];
        sum+=array[i];
    }
    return(sum/n);
}
main()
{
    int i;
    float ave,score[10];
    ave=average(score,10);
    printf("max=%6.2f\nmin=%6.2f\n
           average=%6.2f\n",max,min,ave);
}
```

max

min

作用域

扩展后
c1,c2
的作用范围

扩展后
c1,c2
的作用范围

```
int p=1,q=5;  
extern char c1,c2;  
float f1(int a)  
{ int b,c;  
  extern char c1,c2;  
  .....  
}  
int f3()  
{.....  
}  
char c1,c2;  
char f2(int x,int y)  
{ int i,j;  
  .....  
}  
main()  
{ int m,n;  
  .....  
}
```

p,q的作用范围

c1,c2的作用范围

例 外部变量与局部变量

```
int a=3,b=5;  
max(int a, int b)  
{ int c;  
  c=a>b?a:b;  
  return(c);  
}  
main()  
{ int a=8;  
  printf("max=%d",max(a,b));  
}
```

运行结果：max=8

例 外部变量副作用

```
int i;  
main()  
{ void prt();  
  for(i=0;i<5;i++)  
    prt();  
}  
void prt()  
{ for(i=0;i<5;i++)  
  printf("%c",'*');  
  printf("\n");  
}
```

运行结果：*****

动态变量与静态变量

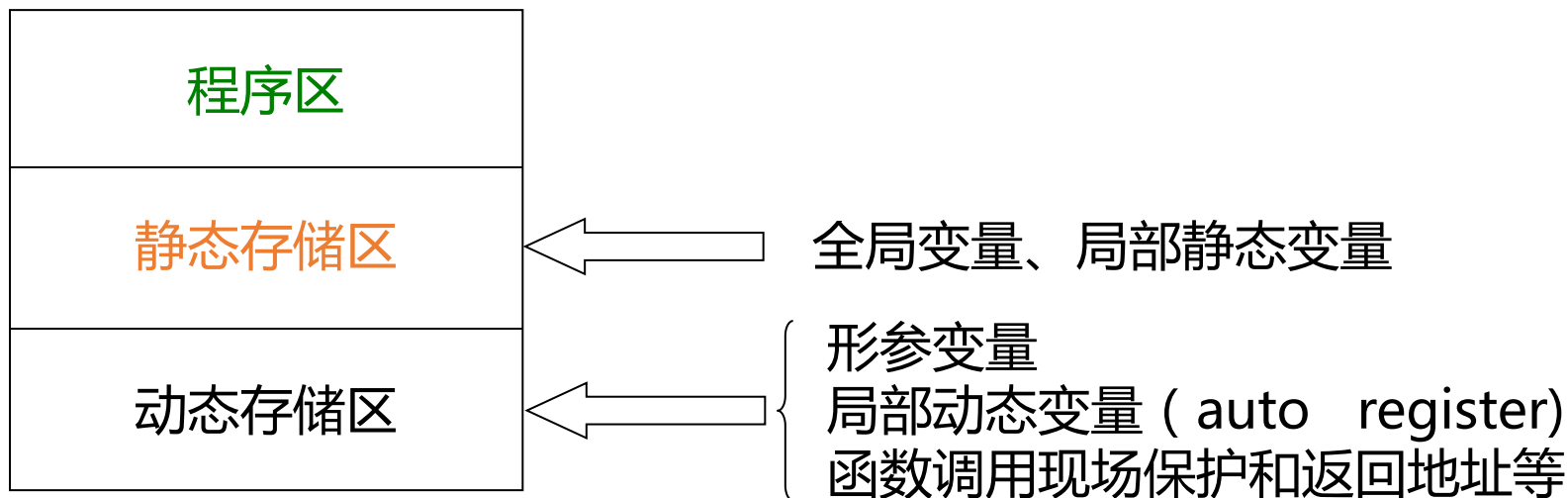
❖ 存储方式

- 静态存储：程序运行期间分配固定存储空间
- 动态存储：程序运行期间根据需要动态分配存储空间

❖ 内存用户区

❖ 生存期

- 静态变量：从程序开始执行到程序结束
- 动态变量：从包含该变量定义的函数开始执行至函数执行结束



例 a、b、c的作用域和生存期

int a;

main()

{

.....
f2;

.....
f1;

.....

}

f1()

{ auto int b;

.....
f2;

.....

}

f2()

{ **static int c;**

.....

}

main → f2 → main → f1 → f2 → f1 → main

a生存期: |-----|

b生存期: |-----|

c生存期: |-----|

a作用域

b作用域

c作用域

例 变量的作用域

```
main()
{   int x=1;
    void prt(void);
    {
        int x=3;
        prt();
        printf("2nd x=%d\n",x);
    }
    printf("1st x=%d\n",x);
}

void prt(void)
{   int x=5;
    printf("3th x=%d\n",x);
}
```

运行结果：
3th x=5
2nd x=3
1st x=1

例 局部静态变量值具有可继承性

```
main()
{ void increment(void);
  increment();
  increment();
  increment();
}
void increment(void)
{ int x=0;
  x++;
  printf("%d\n",x);
}
```

运行结果：1
1
1

```
main()
{ void increment(void);
  increment();
  increment();
  increment();
}
void increment(void)
{ static int x=0;
  x++;
  printf("%d\n",x);
}
```

运行结果：1
2
3

例 变量的寿命与可见性

```
#include <stdio.h>
int i=1;
main()
{
    static int a;
    register int b=-10;
    int c=0;
    printf("-----MAIN-----\n");
    printf("i:%d a:%d b:%d c:%d\n",i,a,b,c);
    c=c+8;
    other();
    printf("-----MAIN-----\n");
    printf("i:%d a:%d b:%d c:%d\n",i,a,b,c);
    i=i+10;
    other();
}

other()
{
    static int a=2;
    static int b;
    int c=10;
    a=a+2;    i=i+32;    c=c+5;
    printf("-----OTHER-----\n");
    printf("i:%d a:%d b:%d c:%d\n",i,a,b,c);
    b=a;
}
```

```
-----MAIN-----
i:1 a:0 b:-10 c:0
-----OTHER-----
i:33 a:4 b:0 c:15
-----MAIN-----
i:33 a:0 b:-10 c:8
-----OTHER-----
i:75 a:6 b:4 c:15
```

6.5 内部函数和外部函数

函数**本质上是全局的**,因为一个函数要被另外的函数调用,但是,也可以指定函数不能被其他文件调用,根据函数能否被其他源文件调用,将函数区分为**内部函数**和**外部函数**。

◆ 外部函数

`extern int fun (int a, int b)`

- 外部函数fun可以为其他文件调用。
- C语言规定,如果在定义函数时省略extern,则默认为外部函数。本书前面所用的函数都是外部函数。

◆ 内部函数

`static int fun(int a,int b)`

- 如果一个函数只能被本文件中其他函数所调用,它称为内部函数。在定义内部函数时,在函数名和函数类型的前面加static。

使用内部函数,可以使函数只局限于所在文件,如果在不同的文件中有同名的内部函数,互不干扰。这样不同的人可以分别编写不同的函数,而不必担心所用函数是否会与其他文件中函数同名,通常把只能由同一文件使用的函数和外部变量放在一个文件中,在它们前面都static使之局部化,其他文件不能引用。