09 函数

内容提要

- 函数概述
- 递归
- 多源代码文件程序的编译
- 地址运算符
- 指针
- 关键概念

复习函数

1. 函数概述

- 函数(function)，用于完成特定任务的独立程序代码单元
  - 执行某些动作。如printf()把数据打印到屏幕上
  - 返回结算结果。如strlen()把指定字符串的长度返回给程序
  - 一般而言，函数可以同时具备以上两种功能

```c
#include <stdio.h>
#define SIZE 50
int main(void)
{
    float list[SIZE];
    readlist(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);
    bargraph(list, SIZE);

    return 0;
}
```

函数概述

- 为什么要使用函数？
  - 代码/功能重用
    - 可以省略重复代码的编写，调用即可
    - 使用他人成果
  - 模块化设计
    - 系统设计需要使用层级方式。程序设计的层级结构逻辑上对应着模块和函数
    - 提高了程序代码可读性。方便代码修改和维护

```
1.  #include <stdio.h>
2.  #define SIZE 50
3.  int main(void)
4.  {
5.      float list[SIZE];
6.      readlist(list, SIZE);
7.      sort(list, SIZE);
8.      average(list, SIZE);
9.      bargraph(list, SIZE);
10.
11.     return 0;
12. }
```

- 如何了解函数？
  - 如何正确地定义函数
  - 如何调用函数
  - 如何建立函数间的通信

1.1 创建并使用简单函数

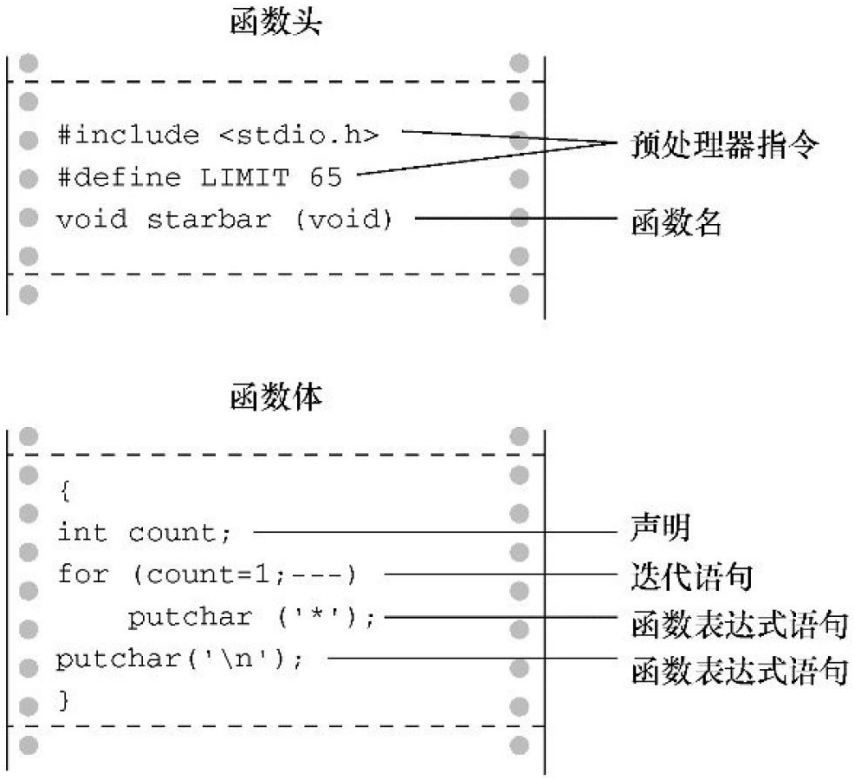
- 程序清单9.1 lethead1.c
- 在一行打印40个星号
- 3处使用了starbar标识符
- 函数原型（function prototype）
  - 告诉编译器函数starbar()的类型
  - 函数原型指明函数返回值类型和函数接受参数类型
- 函数调用（function call）
  - 表明在此处执行函数
- 函数定义（function definition）
  - 明确地指定函数要做什么

```
1.  #define NAME "GIGATHINK, INC."
2.  #define ADDRESS "101 Megabuck Plaza"
3.  #define PLACE "Megapolis, CA 94904"
4.  #define WIDTH 40
5.  void starbar(void);  /* prototype the function */
6.  int main(void){
7.      starbar();
8.      printf("%s\n%s\n%s\n", NAME, ADDRESS, PLACE);
9.      starbar();  /* use the function       */
10.     return 0;
11. }

12. void starbar(void) {  /* define the function    */
13.     int count;
14.     for (count = 1; count <= WIDTH; count++)
15.         putchar('*');
16.     putchar('\n');
17. }
```

void starbar(void);

- 函数原型（function prototype）
  - 告诉编译器函数starbar()的类型
- 函数调用（function call）
  - 表明在此处执行函数
- 函数定义（function definition）
  - 明确地指定了函数要做什么
- starbar()函数中的局部变量
  - 变量count
  - 局部变量（local variable），意思是该变量只属于starbar()函数



1.3 函数参数

- 程序清单9.2 lethead2.c
- 一个更通用的函数
  - void show_n_char(char ch, int num)
  - 使用内置的值来显示字符和重复的次数

```
1.  #define NAME "GIGATHINK, INC."
2.  #define ADDRESS "101 Megabuck Plaza"
3.  #define PLACE "Megapolis, CA 94904"
4.  #define WIDTH 40
5.  #define SPACE ' '
6.  void show_n_char(char ch, int num);
7.  int main(void){
8.      int spaces;
9.      show_n_char('*', WIDTH); //constants as arguments
10.     putchar('\n');
11.     show_n_char(SPACE, 12); //constants as arguments
12.     printf("%s\n", NAME);
13.     spaces = (WIDTH - strlen(ADDRESS)) / 2;
14.     show_n_char(SPACE, spaces);//variable argument
15.     printf("%s\n", ADDRESS);
16.     show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
17.     /* an expression as argument    */
18.     printf("%s\n", PLACE);
19.     show_n_char('*', WIDTH);
20.     return 0;
21. }
22. void show_n_char(char ch, int num){}
```

1.3 函数参数

- 程序清单9.2 lethead2.c
- 写一个更通用的函数

```
1.  #define NAME "GIGATHINK, INC."
2.  #define ADDRESS "101 Megabuck Plaza"
3.  #define PLACE "Megapolis, CA 94904"
4.  #define WIDTH 40
5.  #define SPACE ' '
6.  void show_n_char(char ch, int num);
7.  int main(void){
8.      int spaces;
9.
10.     show_n_char('*', WIDTH);/* using … */
11.     putchar('\n');
12.     show_n_char(SPACE, 12);/* using … */
13.     printf("%s\n", NAME);
14.     spaces = (WIDTH - strlen(ADDRESS)) / 2;
15.     /* Let the program calculate    */
```

- void show_n_char(char ch, int num)
- 使用内置的值来显示字符和重复的次数

```
16.     /* how many spaces to skip      */
17.     show_n_char(SPACE, spaces);/* use … */
18.     printf("%s\n", ADDRESS);
19.     show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
20.     /* an expression as argument    */
21.     printf("%s\n", PLACE);
22.     show_n_char('*', WIDTH);
23.     putchar('\n');
24.
25.     return 0;
26. }

27. /* show_n_char() definition */
28. void show_n_char(char ch, int num){}
```

1.3 函数参数

```c
#include <stdio.h>
#include <string.h>             /* for strlen() */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' ‘
void show_n_char(char ch, int num);

int main(void){
    int spaces;

    /* using constants as arguments */
    show_n_char('*', WIDTH);
    putchar('\n');
    /* using constants as arguments */
    show_n_char(SPACE, 12);
    printf("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS)) / 2;
    /* Let the program calculate    */
    /* how many spaces to skip      */
    /* use a variable as argument   */
    show_n_char(SPACE, spaces);
    printf("%s\n", ADDRESS);
    show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
    /* an expression as argument    */
    printf("%s\n", PLACE);
    show_n_char('*', WIDTH);
    putchar('\n');

    return 0;
}

/* show_n_char() definition */
void show_n_char(char ch, int num){
    int count;

    for (count = 1; count <= num; count++)
        putchar(ch);
}
```

1.4 定义带形式参数的函数

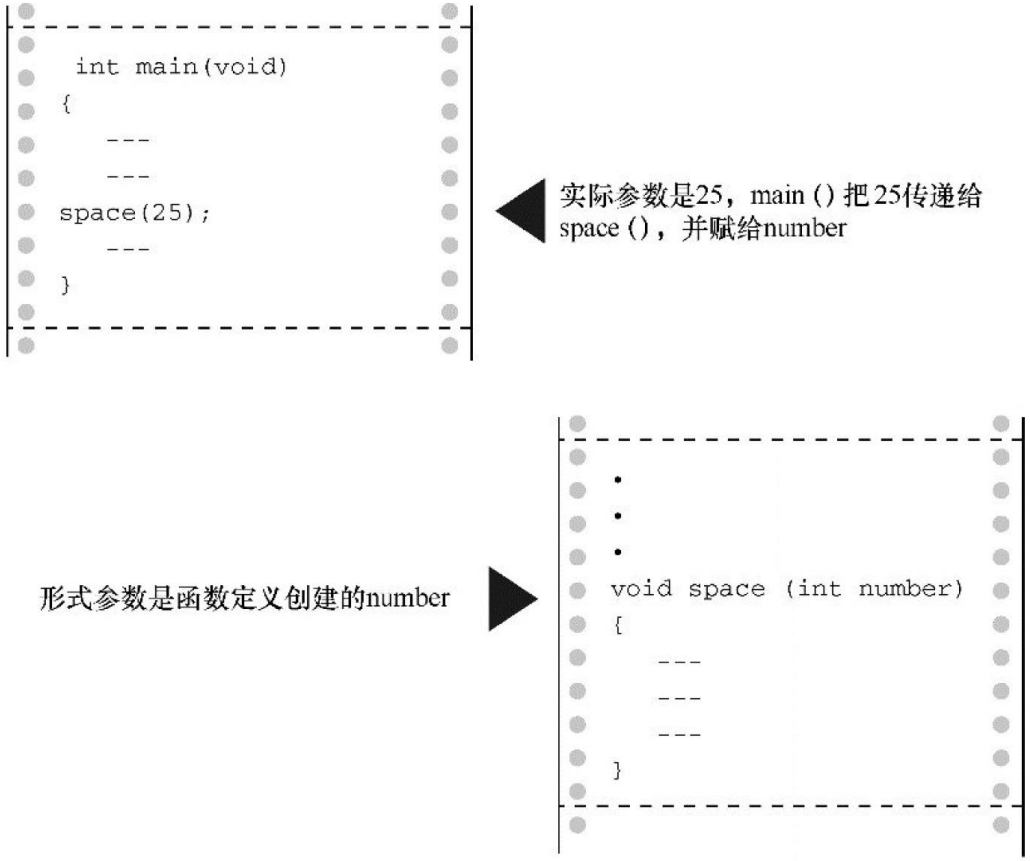
- void show_n_char(char ch, int num)
- 形式参量（formal parameter ，简称形参）
  - ch是char类型，num是int类型
  - 多个参数用逗号分隔，在函数原型和函数头中同时指定
  - 【和定义在函数中变量一样】形式参数是局部变量，该函数私有，意味着
    - 其他函数使用同名变量不会引起名称冲突
    - 每次调用函数，会给这些变量赋值
- 实际参数（actual argument，简称实参）
  - show_n_char(' ' , 12) //实际参数是空格和12，赋值给形参ch ， num
  - 对于函数定义中的每个参数（形参），必须对应于函数调用的一个实参
- 形式参数是被调函数（called function）中的变量，实际参数是主调函数（calling function）赋给被调函数的具体值

1.5 声明带形式参数函数的原型

- 在使用函数之前，要用ANSI C形式声明函数原型
  - `void show_n_char(char ch, int num);`
- 可以省略变量名
  - `void show_n_char(char, int);`
  - 在原型中使用变量名并没有实际创建变量，char仅代表了一个char类型的变量
  - 原型的作用是申明和“占位”，用于编译
    - 后续有链接，即，接入具体的函数

1.6 调用带实际参数的函数

- 函数调用中，实数提供了ch和num的值
- show_n_char(SPACE, 12);
  - 实参是空格字符和12。这两个值被赋给show_n_char()中相应的形参：变量ch和num
- 简而言之，形参是被调函数中的变量，实参是主调函数（calling function）赋给被调函数的具体值



1.7 函数的黑盒视角

- 黑盒特点
  - 关心接口（功能，输入，输出），不关心细节
- show_n_char()
  - 待显示的字符和显示的次数是输入。结果是打印指定数量的字符。输入以参数的形式被传递给函数
  - 这些信息表明了如何在 main()中使用该函数，也可以作为编写该函数的设计说明
- 黑盒方法的核心部分ch、num和count
  - 都是show_n_char()私有的局部变量
  - 如在main()中使用同名变量，那么它们相互独立，互不影响。
  - 如main()有一个count变量，那么改变它的值不会改变show_n_char()中的count，反之亦然
  - 黑盒里发生了什么对主调函数不可见

1.8 使用return从函数中返回值

- 程序清单9.3 lesser.c
- 函数的返回值把信息从被调函数传回主调函数
  - lesser = imin(n,m);
  - answer = 2 * imin(z, zstar) + 25;
- 用于测试函数的程序，有时被称为驱动程序（driver），该驱动程序调用一个函数
  - 如通过测试，则可在一个更重要的程序中使用
- 函数中的return
  - 任意位置（行）出现

```
1.  int imin(int, int);
2.  int main(void){
3.      int evil1, evil2;
4.      printf("A pair of integers (q to quit):\n");
5.      while (scanf("%d %d", &evil1, &evil2) == 2) {
6.          printf("The lesser of %d and %d is %d.\n",
7.                  evil1, evil2, imin(evil1,evil2));
8.          printf("A pair of integers (q to quit):\n");
9.      }
10.     return 0;
11. }

12. int imin(int n,int m){
13.     int min;
14.     if (n < m) min = n;
15.     else min = m;
16.     return min;
17. }
```

1.9 函数类型

- 声明函数时必须声明函数的类型
  - 带返回值的函数类型，应该与其返回值类型相同
  - 没有返回值的函数，声明为void类型
- 类型声明是函数定义的一部分
- 函数类型指的是返回值的类型，不是函数参数的类型
- 函数的前置声明放在主调函数外面。也可以放在主调函数里面

ANSI C函数原型

2 ANSI C函数原型

- 程序清单9.5 proto.c
- ANSI C要求在函数声明时还要声明变量的类型
  - 即，使用函数原型（function prototype）来声明函数的返回类型、参数的数量和每个参数的类型
  - 否则，各编译器的函数调用传参方式不一致！
- 一个不支持ANSI C的编译器会假定用户没有用函数原型来声明函数，它将不会检查参数.为了表明函数确实没有参数，应该在圆括号中使用void关键字
- 函数定义也相当于函数原型。对于较小的函数，这种用法很普遍

```
1.  /* proto.c -- uses a function prototype */
2.  #include <stdio.h>
3.  int imax(int, int);         /* prototype */
4.  int main(void)
5.  {
6.      printf("The maximum of %d and %d is %d.\n",
7.             3, 5, imax(3));
8.      printf("The maximum of %d and %d is %d.\n",
9.             3, 5, imax(3.0, 5.0));
10.     return 0;
11. }

12. int imax(int n, int m)
13. {
14.     return (n > m ? n : m);
15. }
```

递归

3 递归

- 递归（recursion）
  - 函数调用它自己的过程
- 递归函数
  - 直接或者间接调用自己的函数

- 可以使用循环的地方通常都可以使用递归
  - 有时用循环解决问题比较好，但有时用递归更好
  - 递归方案更简洁，但效率却没有循环高

3 递归

- 基本属性
  - 1．每级函数调用都有自己的变量
  - 2．每次函数调用都会返回一次。当函数执行完毕后，控制权将被传回上一级递归。程序必须按顺序逐级返回递归
  - 3．递归函数中位于递归调用之前的语句，均按被调函数的顺序执行
  - 4．递归函数中位于递归调用之后的语句，均按被调函数相反的顺序执行
  - 5．函数每级递归都有自己的变量，但是并没有拷贝函数的代码
  - 6．函数必须包含可以终止递归调用的语句
- 优点：递归为某些编程问题提供了最简单的解决方案
- 缺点：一些递归算法会快速消耗计算机的内存资源。另外，递归不方便阅读和维护

3.1 演示递归

- 程序清单9.6 recur.c
- up_and_down()调用自己

```
1.  /* recur.c -- recursion illustration */
2.  #include <stdio.h>
3.  void up_and_down(int);

4.  int main(void){
5.      up_and_down(1);
6.      return 0;
7.  }

8.  void up_and_down(int n)
9.  {
10.     printf("Level %d: n location %p\n", n, &n); // 1

11.     if (n < 4)
12.         up_and_down(n+1);

13.     printf("LEVEL %d: n location %p\n", n, &n); // 2
14. }
```

3.2 递归的基本原理

- 第1，每级函数调用都有自己的变量
- 第2，每次函数调用都会返回一次。当函数执行完毕后，控制权将被传回上一级递归
- 第3，递归函数中位于递归调用之前的语句，均按被调函数的顺序执行
- 第4，递归函数中位于递归调用之后的语句，均按被调函数相反的顺序执行
- 第5，虽然每级递归都有自己的变量，但是并没有拷贝函数的代码
- 第6，递归函数必须包含让递归调用停止的语句

变量	n	n	n	n
第1级调用后	1			
第2级调用后	1	2		
第3级调用后	1	2	3	
第4级调用后	1	2	3	4
从第4级调用返回后	1	2	3	
从第3级调用返回后	1	2		
从第2级调用返回后	1			
从第1级调用返回后	（全部结束）			

3.3 尾递归

- 递归调用置于函数的末尾，在return语句之前。这种形式的递归被称为尾递归（tail recursion）
  - 递归调用在函数的末尾
  - 尾递归是最简单的递归形式，它相当于循环

程序清单9.7 factor.c

```c
long fact(int n){      // loop-based function
    long ans;
    for (ans = 1; n > 1; n--) ans *= n;
    return ans;
}
long rfact(int n){  // recursive version
    long ans;
    if (n > 0) ans= n * rfact(n-1);
    else ans = 1;
    return ans;
}
```

```c
long fact(int n);
long rfact(int n);
int main(void){
    int num;
    printf("Value range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1){
        if (num < 0) printf("Input >= 0, please.\n");
        else if (num > 12) printf("Input < 13.\n");
        else{
            printf("loop: %d factorial = %ld\n", num, fact(num));
            printf("re: %d fact = %ld\n", num, rfact(num));
        }
        printf("Value range 0-12 (q to quit):\n");
    }
    return 0;
}
```

3.4 递归和倒序计算

```c
/* binary.c -- prints integer in binary form */
#include <stdio.h>
void to_binary(unsigned long n);

int main(void){
    unsigned long number;
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");

    return 0;
}
```

```c
/* recursive function */
void to_binary(unsigned long n)
{
    int r;

    r = n % 2;
    if (n >= 2)
        to_binary(n / 2);
     putchar(r == 0 ? '0' : '1');

    return;
}
```

处理倒序：二进制

程序清单9.8 binary.c

3.5 递归的优缺点

- 递归
  - 优点，递归为某些编程问题提供了最简单的解决方案
  - 缺点，一些递归算法会快速消耗计算机的内存资源。另外，递归不方便阅读和维护

- 在程序中使用递归要特别注意，尤其是效率优先的程序

多源代码文件程序的编译

4 多源代码文件程序的编译

- 因操作系统而异
  - gcc file1.c file2.c
  - .sln
- 使用头文件
  - hotel.h：符号常量和 hotel.c 中所有函数的原型
  - hotel.c ：函数支持模块
  - usehotel.c ：带有main函数的控制模块（使用者）

地址运算符&

5 地址运算符&

- 一元运算符&
  - 取得变量的存储地址
  - 地址运算符的操作数必须是左值，不能用于常量或者结果不是引用的表达式。

- 例如：

```
int y = 5;
int * yPtr; //定义指针
yPtr = &y; //yPtr指向y
*yPtr = 8; //yPtr指向的内存赋值，也就是y的值编程8
```

程序清单9.12 loccheck.c

```c
/* checks to see where variables are stored  */
#include <stdio.h>
void mikado(int);          /* declare function  */
int main(void)
{
    int pooh = 2, bah = 5;  /* local to main()   */
    printf("pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("bah = %d and &bah = %p\n", bah, &bah);
    mikado(pooh);
    return 0;
}

void mikado(int bah){     /* define function   */
    int pooh = 10;    /* local to mikado() */
    printf("pooh = %d &pooh = %p\n", pooh, &pooh);
    printf("bah = %d and &bah = %p\n", bah, &bah);
}
```

改变调用函数中的变量

6 改变调用函数中的变量

- 9.13 swap1.c
- 9.13 swap2.c

- 函数调用
  - 按值传递，实参复制给形参，函数中形参进行运算，因此修改形参的值不影响实参
  - 通过指针，可以通过间接方式修改实参相关的信息【如修改实参指向内存的值】

```c
#include <stdio.h>
void interchange(int u, int v);
int main(void){
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x , y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);
    return 0;
}

void interchange(int u, int v){
    int temp;
    printf("Originally u = %d and v = %d.\n", u , v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

指针

7 指针

- 指针（pointer）
  - 存储内存地址的变量（或数据对象）
  - 指针变量的值是地址
- 一个指针变量名是ptr，指向pooh
  - ptr = &pooh; //把pooh的地址赋给ptr，也可以说ptr“指向”pooh
  - 通过赋值，指针还可以指向别处
    - ptr = &bah; // 把ptr指向bah

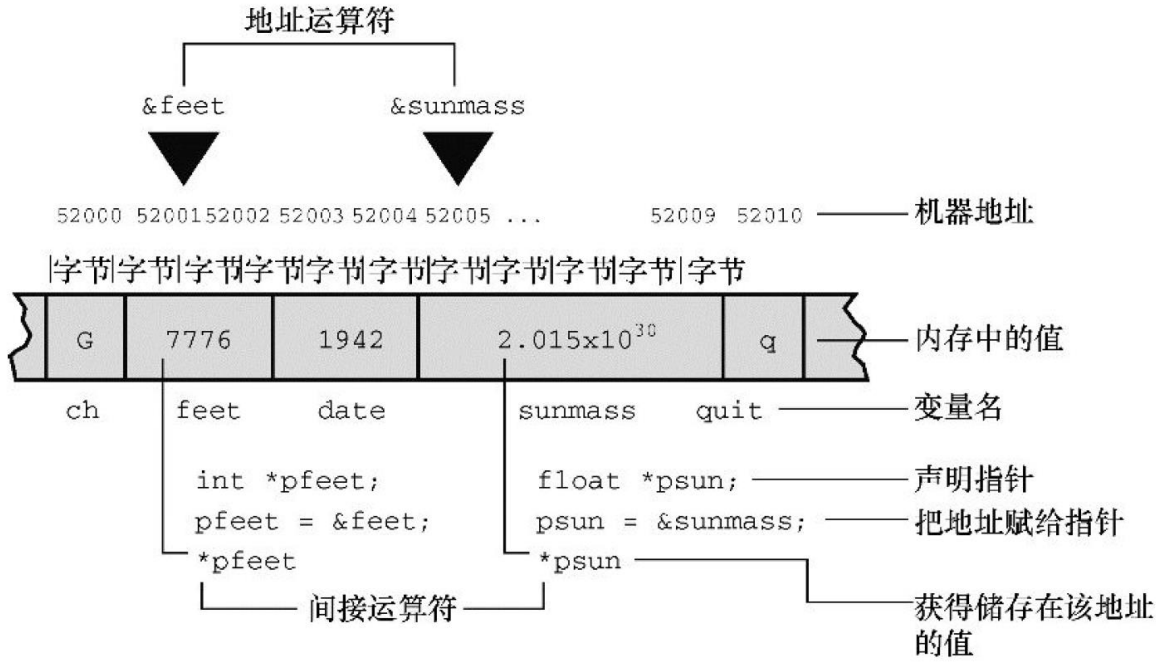
7.1 间接运算符：*

- * 间接运算符（indirection operator）
  - 一元运算符
  - 当后接一个指针名或地址时，*给出储存在被指向地址中的数值

- 使用间接运算符*找出存储在bah中的值，该运算符有时也称为解引用运算符（dereferencing operator）

7.2 声明指针

- 声明指针变量时，必须指定指针所指向变量的类型
- 指针声明
  - 需要声明指针变量的类型
  - 带*表示变量是指针
  - int * pi; //pi是指向整数变量的指针
  - 指针可在声明中初始化成0或nullptr或地址



7.3 使用指针在函数间通信

- 程序清单9.15 swap3.c

- 该函数传递的不是x和y的值，而是它们的地址
  - 出现在interchange()原型和定义中的形式参数u和v是地址
  - 使用指针和*运算符，该函数可以访问存储在这些位置的值并改变它们

```c
/* swap3.c -- 使用指针解决交换函数的问题 */
#include <stdio.h>
void interchange(int * u, int * v);
int main(void){
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(&x, &y); // 把地址发送给函数
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int * u, int * v){
    int temp;
    temp = *u; // temp获得 u 所指向对象的值
    *u = *v;
    *v = temp;
}
```

指针的用法9.15 swap3.c程序

- 地址运算符：&
  - 后跟一个变量名时，&给出该变量的地址。
- 间接运算符：*
  - 后跟一个指针名或地址时，*给出储存在指针指向地址上的值。

- `ptr = &bah;`
- `val = *ptr; // 找出ptr指向的值`

- 使用指针解决交换函数的问题

关键概念

8 关键概念

- 用函数模块学会分而治之
  - 通过小的，简单的部件构件程序；学会软件重用，用现有的函数作为构建块来创新新的程序
- 使用参数向函数传递数值，return让函数返回一个数值
- 在函数中操作调用函数的变量，可以使用地址及指针
- 递归就是C函数可以调用自身
  - 不过递归会浪费内存和花费时间