### 前言

C语言编程仍然是编程工作者必备的技能。本书的基础版本《C语言解惑》<sup>[1]</sup>通过比较编程中存在的典型错误,从而实现像雨珠打在久旱的沙滩上一样滴滴入骨的效果,使学习者更容易记住编程的要诀,并通过演示如何将一个能运行的程序优化为更好、更可靠的程序,使读者提高识别坏程序和好程序的能力。尽管如此,那本书仍然要照顾初学者并兼顾知识的完整性,所以讨论的深度有所限制。为此,我们决定推出它的提高版,并将讨论聚焦于函数设计。

本书将集中讨论C语言的核心部分——函数设计。函数设计涉及函数类型、函数参数及返回值,这就要求读者熟练掌握指针和数组的知识,此外,还要掌握多文件编程以及多文件之间的参数传递等知识。

因为本书要求读者已经学过C语言,所以我们可以完整、系统地论述各个部分的内容,无须赘述基础知识。本书的另一个特点是每一章之间都有知识交叉,进而达到讲透的目的。如果遇到不清楚的知识点,读者可以自行学习相应参考资料,也可以与《C语言解惑》配合学习。

本书的落脚点是实现C语言的结构化程序设计。为实现这一目标,本书专门选择了完整的设计实例。尤其是第10章,结合趣味游戏程序,综合讲解函数设计和多文件编程。

本书各个部分论述详细,涉及的知识面广,有些知识是传统教材中所没有的,所以它既可以作为从事教学的老师及工程技术人员的参考书,也可以作为常备手册。其实,它不仅对工程技术人员极有参 考价值,也能帮助在校生进行编程训练或作为毕业论文的参考资料。此外,本书对于初学者也大有帮助,他们可以将它作为课外读物,对目前看不懂的地方,可以等具备相关知识之后再来研究,彼时将收 获更大。总之,本书能帮助各类人群找到自己需要的知识并有所收获,而这也将拓宽本书的应用范围。

本书共分10章。第1章通过例子说明引入指针变量的必要性并简单介绍指针变量的基本性质。第2章通过实例解释指针的基本性质。第3章介绍数组及数组的边界不对称性。第4章介绍C语言中两个非常重要的概念——数组和指针。第5章介绍如何掌握函数设计和调用的正确方法。第6章介绍如何设计合理的函数类型及参数传递方式。第7章先讨论函数设计的一般原则,然后结合典型算法,用实例说明设计的具体方法,以便使读者进一步开阔眼界。第8章结合具体实例详细介绍头文件的编制、多个C语言文件及工程文件的编制等方法,以提高读者的多文件编程能力。第9章给出两个典型的多文件编程实例,一个使用链表,另一个使用数组。第10章中的游戏程序实例将加深读者对一个完整工程项目的理解。为了学习方便,本书提供全部程序代码。

本书的两位作者分别撰写各章的不同小节,然后逐章讨论并独立成章。刘燕君负责第1~6章,刘振安负责第7~10章,最后由刘振安统稿。参与本书工作的还有周淞梅实验师、苏仕华副教授、鲍运律教 授、刘大路博士、唐军高级工程师等。

在编写过程中,我们得到了中国科学院院士、中国科学技术大学陈国良教授的大力支持,特此表示感谢!对书中所引用资料的作者及网络作品的作者表示衷心感谢!

作者

zaliu@ustc.edu.cn

2016年6月

[1] 此书已于2014年由机械工业出版社出版,书号978-7-111-47985-7。

## 第1章 引入指针变量

指针在C语言中具有举足轻重的地位,也是编制C程序的基本功之一。本章将通过例子说明引入指针变量的必要性并简单介绍指针变量的基本性质。

## 1.1 变量的三要素

一个变量具有3个要素:数据类型、名字和存放变量的内存地址。本节将简要回顾变量的3个要素,以便为引入指针打下基础。

### 1.基本数据类型

数据类型是C语言中非常重要的一个概念,它将C语言所处理的对象按其性质不同分为不同的子集,以便对不同类型的数据规定不同的运算。void是无类型标识符,只能声明函数的返回类型,不能声明 变量,但可以声明指针。

本节只涉及基本数据类型,C语言的基本数据类型有如下4种。

- · char 字符型
- · int 整数型
- ·float 浮点数型 (又称为单精度数)
- · double 双精度浮点数型

另外还有用于整型的限定词short、long、signed和unsigned。short和long表示不同长度的整型量;unsigned表示无符号整型数(它的存放值总是正的);可以省略signed限定词。例如,可以将如下声明

short int x; unsigned int z;

中的说明符int省略。即它们与如下声明

short x; unsigned z; 是等效的。上述数据类型的长度及存储的值域也随编译器不同而变化,ANSI C标准只限定int和short至少要有16位,而long至少32位,short不得长于int,int不得长于long。表1-1是数据类型的长度及存储的值域表,表1-1中VC是Visual C++6.0的缩写。表1-2是加了限定词的数据类型及它们的长度和取值范围。

C语言提供一个关键字sizeof,用来求出对于一个指定数据类型,编译系统将为它在内存中分配的字节长度。例如,语句"printf("%d",sizeof(double));"的输出结果为8。

注意在表1-1中的标注,在VC中int使用4字节,这是本章计算的依据。

C语言定义的存储类型有4种:auto、extern、static和register,分别称为自动型、外部型、静态型和寄存器型。自动型变量可以省略关键字auto。存储类型在类型之前,即

存储类型 类型

例如auto int和static float等。可以省略auto,其他类型均不可以省略。

表1-1 数据类型的长度及存储的值域

| 类 型    | 位长度 | 字节数 | 值域                  | 备注         |
|--------|-----|-----|---------------------|------------|
| char   | 8   | 1   | 0 ~ 255             |            |
| int    | 16  | 2   | -32 768 ~ 32 767    | VC 使用 4 字节 |
| float  | 32  | 4   | 3.4E-38 ~ 3.4E+38   |            |
| double | 64  | 8   | 1.7E-308 ~ 1.7E+308 |            |

#### 表1-2 加限定词的数据类型及其长度和取值范围

| 类 型          | 位长度 | 字节数 | 值域                             | 备 注        |
|--------------|-----|-----|--------------------------------|------------|
| short int    | 16  | 2   | -32 768 ~ 32 767               |            |
| long int     | 32  | 4   | -2 147 483 648 ~ 2 147 483 647 |            |
| unsigned int | 16  | 2   | 0 ~ 65 535                     | VC 使用 4 字节 |

#### 2.变量的名字和变量声明

C语言中大小写字母是具有不同含义的,例如,name和NAME就代表不同的标识符。原来的C语言中虽然规定标识符的长度不限,但只有前8个字符有效,所以对定义为

dwNumberRadio dwNumberTV

### 的两个变量是无法区别的。

现在流行的为32位操作系统配备的C编译器已经能识别长文件名,不再受8位的限制。另外,在选取时不仅要保证正确性,还要考虑容易区分,不易混淆。例如,数字1和字母在一起,就不易辨认。在取名时,还应该使名字有很清楚的含义,例如使用area作为求面积函数的名字,area的英文含义就是"面积",这就很容易从名字猜出函数的功能。对一个可读性好的程序,必须选择恰当的标识符,取名应统一规范,以便使读者能一目了然。

在现在的编译系统中,内部名字中至少前31个字符是有效的,所以应该采用直观的名字。一般可以遵循如下简单规律。

- 1) 使用能代表数据类型的前缀。
- 2) 名称尽量接近变量的作用。
- 3) 如果名称由多个英文单词组成,每个单词的第一个字母大写。
- 4) 由于库函数通常使用下划线开头的名字,因此不要将这类名字用作变量名。
- 5) 局部变量使用比较短的名字,尤其是循环控制变量(又称循环位标)的名字。
- 6) 外部变量使用比较长且贴近所代表变量的含义。
- 7) 函数名字使用动词,如Get\_char(void)。变量使用名词,如iMen\_Number。

变量命名可以参考Windows API编程推荐的匈牙利命名法。它是通过在数据和函数名中加入额外的信息,既增进程序员对程序的理解,又方便查错。

所有的变量在使用之前必须声明,所谓声明即指出该变量的数据类型及长度等信息。声明由类型和具有该类型的变量列表组成。如:

int lower, upper; char c, name[15];

### 变量可按任何方式分布在若干个声明中,上述声明同样可以写成:

后一种形式会使源程序冗长,但便于给每个声明加注释,也便于修改。

变量的存储类型在变量声明中指定。变量声明的一般形式为:

存储类型 类型 变量名列表;

应该养成在声明时就为变量赋初值的习惯,但在某些特殊场合则只能声明,如头文件中对外部变量的声明,下面是一些典型的例子。

```
auto int a;
static float b, c;
extern double x;
register int i=0;
extern char szClassame[];
static int size=50;
const double PI=3.14159;
```

#### 3.变量的地址

内存地址由系统分配,不同机器为变量分配的地址大小虽然可以不一样,但都必须给它分配地址。

在C语言中,声明和定义两个概念是有区别的。声明是对一个变量的性质(如构成它的数据类型)加以说明,并不为其分配存储空间;而定义则是既说明一个变量的性质,又为其分配存储空间。定义 一个函数,也是为它提供代码。

### 1.2 变量的操作

从三要素可知,既可以通过名字对变量进行操作,也可以通过地址对存放在该地址的变量进行操作。

#### 1.左值和右值的概念

变量是一个指名的存储区域,左值是指向某个变量的表达式。"左值"来源于赋值表达式"A=B",其中左运算分量"A"必须能被计算和修改。左值表达式在赋值语句中既可以作为左操作数,也可以作为右操作数,例如"x=56"和"y=x",x既可以作为左值(x=56),又可以作为右值(y=x)。但右值"56"只能作为右操作数,而不能作为左操作数。由此可见,常量只能作为右值,而普通变量既可以作为左值,也可以作为右值。如下语句

```
const int a = 256;
```

定义的a,显然不能作为左值,只能作为右值。

由此可见,值可以作为右值,如整数、浮点数、字符串、数组的一个元素等。在C语言中,右值以单一值的形式出现。假设有字符数组a和b,则这两个字符数组的每个元素均可以作为右值,即 "a[0]=b[0]" 是正确的, "b[0]=a[0]" 也是正确的。需要注意的是,它们在 "="号左右两边的含义是不同的。以a[0]为例,在 "b[0]=a[0]" 中,它是作为值出现的,即a[0]是数组第1个元素的值;而在 "a[0]=b[0]" 中,它是作为变量出现的,即a[0]是数组的第1个元素的变量名,所以a[0]可以作为左值。即可以使用数组的具体元素作为左值和右值。

a和b都不是字符串的单个元素,所以都不能作为右值。而因为a和b可以作为数组首地址的值赋给指针变量,所以在这种情况下它们又都可以作为右值。

由此可见,在C语言中,左值是一个具体的变量,右值一定是一个具体类型的值,所以有些可以既可以作为左值,也可以作为右值,但有些只能作为右值。

### 2.对变量的基本操作

C语言使用地址运算符"&"来取变量存储在内存中的首地址。假设变量a=55,但不同机器和系统为它分配的地址是不一样的,这里也假设分配的十六进制地址是0x0012FF7C。如何从这个地址取出"55"呢?

C语言提供了"\*"运算符,用来取出地址里的值。"&a"代表地址,显然"\*&a"可以取出55。使用下面语句

```
printf("%d,%d\n",a,*&a,);
```

可以得到输出结果为"55,55",即证明a和\*&a是等价的。

## 1.3 指针变量

1.2节介绍了"\*"和"&"运算符,本节将通过具体的例子说明它们的用途,从而引入指针变量。

### 1.对有效地址进行操作

【例1.1】取地址里的值和取地址里存放的地址值的例子。

```
#include <stdio.h>
int main()
{
    int a = 65;
    int addr;
    addr = 0x0012ff7c;
    printf("0x%p, 0x%p, 0x%p\n", &a, addr, &addr);
    printf("%d, %d, 0x%p\n", a, *&a, *&addr);
    return 0;
}

#include <stdio.h>
// 将a的首地址存入变量addr
// 将a的首地址存入变量addr
// 输出3个地址
printf("%d, %d, 0x%p\n", a, *&a, *&addr);
// 输出变量及地址里的值

**The control of the control
```

语句 "int a=65; " 定义了整型变量a的值为65, VC使用4字节存储65。假设存放它的内存首地址为十六进制的"0x0012ff7c",则可以使用输出格式"%p"来输出这个地址。"0x"是标注它为十六进制地址,也可以简单地使用"%#p"输出地址。

一个变量具有地址和值,&是取地址值运算符。系统为整型变量a和adder分别分配地址"0x0012ff7c"和"0x0012ff78"。给整型变量addr赋十六进制数,这个数可以代表地址,但不一定是有效的地址(将计算机可以存取的地址称为有效地址)。已经验证0x0012ff7c是分配给变量a的地址,所以addr是被赋给一个有效地址。

将"0x0012ff7c"赋给变量addr,&addr是系统分给它的地址"0x0012ff78",这个地址与a的地址相差4字节,证明它们是连续存放的。现在这个地址里存放的是地址0x0012ff7c,也就是变量a的地址,因为\*&addr应该输出a的地址而不是a的值,所以要使用%p格式。程序输出结果也验证了如上分析。即输出为

既然addr存放的是有效地址,\*addr也应该能输出这个地址里的值,也就是变量a的地址。不过,&a虽然和addr的值是一个值,但它们对运算的反应并不一样。a是变量,其值为65,&a是存储地址, 所以\*&a是取地址里的值。类似的,\*addr应该输出它的存储内容,即地址"0x0012ff7c",而\*\*addr应该输出地址"0x0012ff7c"里的内容65。其实这是不行的,因为编译系统并不知道\*addr存储 的"0x0012ff7c"是地址,所以将它作为整数,因此编译系统会报错,当然使用\*\*addr也要出错。

但addr里面确实装的是地址,所以可以将这个整数强制转为地址。\*addr加上强制转换,应该是"(int\*) addr",它的内容是变量a的地址"0x0012ff7c"。

再对它使用\*运算符,即\*(int\*)addr,输出结果应该是存在这个地址里的变量a的值65。下面的例子验证了如上分析。

【例1.2】取地址里的整数值和取地址里存放的地址值。

```
#include <stdio.h>
int main()
{
   int a=65;
   int addr;
   addr=0x0012ff7c;
   printf("0x%p, 0x%p\n", &a, addr, (int*)addr);
   printf("%d, %d, %d\n",a,*&a, *(int*)addr);
   return 0;
}
```

#### 输出结果如下:

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C
```

#### 2.引入指针的概念

在【例1.2】中,要使用a的地址直接给addr赋值,必须事先知道这个地址。为了避免这个麻烦,可以直接将地址表达式"&a"赋给变量。即

addr=&a:

因为&a是地址值,addr是整型变量,所以会给出警告信息。不过,可以使用强制转换让警告信息"闭嘴"。即

addr=(int)&a;

这样一来,使用起来就方便多了。

【例1.3】直接将变量地址赋给另一个变量的例子。

```
#include <stdio.h>
int main()
{
   int a=65;
   int addr;
   addr=(int)&a;
   printf("0x$p, 0x$p, 0x$p\n", &a, addr, (int*)addr);
   printf("8d, %d, %d, \n",a, *&a, * (int*)addr);
   return 0;
}
```

### 程序运行结果如下:

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C 65,65,65,
```

运行结果完全吻合。如果想像对待变量一样对待addr,即使用"\*"和"&"运算符的结果与变量a一样,就必须定义新的变量类型。分析下述表达式:

```
addr=(int)&a;
(int*)addr
*(int*)addr
```

由此可见,如果定义一种变量,使它存储的数据类型是地址,问题就可以迎刃而解了。

要使用 (int\*) addr的addr存储地址,那么就要用 "int\*" 声明 "addr" ,即

int \* addr

这时 "addr=&a" 就无需转换,赋值顺理成章了。

"addr"输出地址,则"\*addr"输出地址里的值。这就与普通变量的使用方法完全一样了。

暂且将使用 "int\*" 定义的变量称为指针变量,下面编程验证一下这个设想。

【例1.4】使用新的数据类型(指针)的例子。

```
#include <stdio.h>
int main()
{
   int a=65;
   int *p;
   p=6a;
   printf("0x%p, 0x%p, 0x%p\n", &a, &p, p);
   printf("%d,%d,%d\n",a,*&a, *p);
   return 0;
}
```

### 输出结果如下:

输出结果与【例1.3】的完全一样。

这种类型称为指针类型,指针类型存储的是地址值。因为这里使用的地址值是另外一个变量的地址,所以是有效地址。要明确的是,地址值不一定是有效地址,所以说从指针的引入开始,也就暗示着它存在着无法预防的错误。

#### 3.引入字符指针再次验证

下面再使用字符来验证一下,看是否与整数的结论相同。

【例1.5】使用字符的例子。

```
#include <stdio.h>
int main()
{
    char a="B";
    int addr;
    addr=(int)&a;
    printf("0x8p, 0x8p, 0x8p\n", &a, addr, (char*)addr);
    printf("%c, %c, %c\n", a,*&a, *(char*)addr);
    return 0;
}
```

#### 程序输出结果如下:

```
0x0012FF7C, 0x0012FF7C, 0x0012FF7C
B, B, B
```

程序验证了" (char\*) addr"和"\* (char\*) addr"的作用,从而推知,可以定义字符类型的指针。

【例1.6】使用字符指针的例子。

```
#include <stdio.h>
int main()
{
    char c='B';
    char *p;
    p=&c;
    printf("0x%p, 0x%p, 0x%p\n", &c, &p, p);
    printf("%c, %c, %c\n", c, *&c, *p);
    return 0;
}
```

### 程序输出结果如下:

```
0x0012FF7C, 0x0012FF78, 0x0012FF7C B, B, B
```

## 4.声明指针类型的变量

由此可见,声明指针变量是用普通数据类型加"\*"号。例如:

至于"\*"号的位置,对于整型类型指针p,以下三个位置均可。

至于哪种写法好,也要根据实际情况,以不造成误会为准。下面是正确的使用实例。

```
int *a, p, d;
d = 45;
a=&d;
p=d;
```

在上面的声明中,只有a是指针变量,p和d都是整型变量。使用下面的声明就可以提高可读性。

```
int p, d, *a,
```

系统不管是何种数据类型的指针,一律分配4字节,即各种类型的指针所占内存的大小是一样的。

【例1.7】演示典型指针长度的例子。

下面再以整型指针p为例,说明指针的含义。

#### 【例1.8】演示整型指针的例子。

```
#include <stdio.h>
int main()

{
    int a=65, *p;
    p=6a;
    printf("a的值等于%d, a的音地址是%#p, p指向的地址是%#p。\n", a, &a, p);
    printf("a的值等于%d, 通过p内的地址等p使用%d. \n", a, p, *p);
    printf("p指向的地址为%#p, 存成p的地址是%#p。\n", p, &p);
    return 0;
}
```

#### 程序输出结果如下:

a的值等于65, a的首地址是0x0012FF7C, p指向的地址是0x0012FF7C。通过名字使用65, 通过p内的地址0x0012FF7C使用65。 p指向的地址为0x0012FF7C, 存放的地址是0x0012FF78。

可以画出变量p和a之间的关系如图1-1所示。

# 存放对象p的首地址0x0012FF78

指向关系

存放整型变量a的首地址0x0012FF7C

地址运算: &a=0x0012FF7C

0x0012FF7C

0X0012FF/C

p指向a的存储首地址 \*p指向变量a的值

地址运算: &p=0x0012FF78

地址运算: p=&a=0x0012FF7C

值引用: \*p的内容为65

赋值运算: a=65

65

图1-1 变量p和a之间的关系示意图

系统为指针变量分配地址0x0012FF78,一般不需要管它。它存储的地址是变量a的首地址,正是因为这种数据类型声明的变量代表指向另一个数据类型变量的存储首地址,所以得名为"指针"类型。 注意这句话:指向另一个数据类型变量的存储首地址。

读者有时会对使用如下方法在声明指针的同时初始化指针的方式感到困惑,即

int \*p=&a;

实际上,选择"int\*p;",认为"int\*"是一种指向整型的指针类型,用它声明指针变量p,p应该赋予a的地址,所以应是"p=&a"。声明指向整型的指针变量p并同时初始化,也就顺理成章为"int\*p=&a"。显然,称p为指针变量(存放的是变量的首地址),而不是称\*p为指针变量(\*p代表指针指向的地址单元所存放的值)。

由此可知,p的值是地址,虽然这个地址就是变量a在内存中的存储首地址,但并不直接说p的值是a的地址,而说成p指向a的存储首地址,简称p指向a的地址。

## 1.4 指针类型

假设已经知道变量的地址(NULL也算已知),现在将上一节的构造语法总结如下:

存储类型 数据类型\* 指针名; 指针名=变量地址;

### 或者采取直接初始化的方法:

存储举型 数据举型\* 指针名=变量抽址:

默认的存储类型为自动存储类型 (auto) ,目前也仅以自动存储类型为例,以后将通过例子进一步介绍存储类型。现在假设它们具有如图1-2所示的形式。由关联关系可知,\*p和a同步变化,即改变任何一个的值,它们的值保持一致。如果改变p的内容,如使用语句 "p=&b;",这使得\*p=66,它与b同步,不再与a有任何关系。

# 名字标识符

# 名字的含义

# 关联关系

变量的名字a

变量的内容为数值: a=55

变量的名字b

变量的内容为数值: b=66

指针的名字p

指针的内容为地址: p=变量a的地址

变量a的地址: &a

p=&a; \*p=a

图1-2 指针操作关系图

### 【例1.9】说明对p和\*p进行赋值操作含义的程序。

#### 程序输出如下:

```
&a:0x0012FF7C, &b:0x0012FF78, &p:0x0012FF74
55 66 &a:0x0012FF7C &p:0x0012FF74
88 88 &a:0x0012FF7C &p:0x0012FF74
88 66 &b:0x0012FF78 &p:0x0012FF74
```

在【例1.9】中,指针本身的地址不会变化,它反映了系统需要为指针p分配地址这一概念。正如使用a不要再考虑&a一样,以后也不再考虑&p。

【例1.10】下面是一个使用数组b的首地址作为右值的例子,该程序将数组a的内容复制到数组b中,然后输出两个数组的内容以便验证。

```
// 将数组的的首地d址作为右值的例子
#include <stdio.h>
void main ()d
{
    char a[]="We are here! Where are you?", b[28], *p,
    int i=0;
    p=b;
    while(p[i]=a[i]) // 数组的价首地址作为右值赋给左值p
    i++;
    printf(a); printf("\n");
    printf(b); printf("\n");
```

### 程序运行结果如下:

We are here! Where are you? We are here! Where are you?

## 第2章 指针基础知识

本章将通过实例,解释指针的基本性质以便为用好指针打下基础。

# 2.1 指针运算符

指针有两种专门的运算符: "\*" 和 "&"。它们都仅需要一个操作数,但作用不同。假定已经初始化整型变量a、b和整型指针变量p。另外,它有一个从 "(\*)."简化而来的 "->" 运算符,用于存取结构等类型的成员,还可以使用下标 "[]"进行指针操作。

### 1.&运算符

如前所述, "&" 仅仅返回这个操作数的地址。而语句

p = &a;

只表示把变量a的地址赋给p,它不改变a的值。假设变量a的值为56,它存放在内存中的首地址是0x0012FF7C,执行上述语句后,p的值为0x0012FF7C。

```
b = *p;
```

表示将a的值赋给b, b的值是56, 运算符"\*"理解为: b接收了在地址p中的值。

可以通过指针间接地存取目标。如上所述,单目运算符"\*"将它的操作数作为最终目标的地址来处理,存取的变量是该地址里的内容。

由此可见,如果b为一个整型变量,在执行语句

p = &a;

#### 之后,下面两条语句

```
b = *p;

b = a:
```

的功能是等价的,都是将p所指向的单元的内容赋给b,第一条语句实际上是对p的间接存取。

#### 3.->运算符

为了书写使用方便,又从"(\*)."运算符演化出"->"运算符。

### 【例2.1】使用结构指针的程序。

### 程序输出结果如下:

```
Hardness Marker Number
2 F 485
0 G 38
3 E 108
```

已经知道一个结构,例如结构数组pen[0],它的成员hardness的值可以通过

pen[0].hardness

取得。又知道,指针变量的值是它所指向的数据的地址,故用结构指针pen来求结构成员,例如hardness的值,可通过语句

(\*pen).hardness

来取得。这里的圆括号不能省略。因为"."的运算优先级高于"\*";而在这里首先要求出结构指针所指向的结构,然后再求这个结构的成员,故必须加圆括号。表达式

(\*pen).hardness

写起来很费事,可以将它表示成如下语句

pen -> hardness

其中->是由负号 "-" 和大于号 ">" 组成的。这种表示方法显得相对简单些。

这样, 求结构成员值的一般形式就是:

指向结构变量指针的名字 -> 成员名字

### 例如在上面程序的for循环中的打印语句参数表

(\*pen).hardness, (\*pen).marker, (\*pen).number);

## 就可以简化为如下形式:

```
pen -> hardness, pen -> marker, pen -> number);
```



对结构变量本身进行操作时,必须用"."运算符。但若使用结构指针,必须使用箭头运算符。

#### 【例2.2】对p使用下标进行操作的程序。

```
#include <stdio.h>
int main ()

{
    int a = 36, b = 63, c = 656, i = 0, *p = &a;
    for(i =0; i>-3; i--)
        printf ("%4d", p[i]);
    printf ("\n");
    p=6c;
    for(i =0; i<3; i++)
        printf ("%4d", p[i]);
    printf ("\n");
    return 0;
}</pre>
```

#### 程序输出结果如下:

```
36 63 656
656 63 36
```

使用下标时,p是从0开始计数的,即p[0]为起点。从p[0]可以正数(下标正序增加),也可以反数(下标按负数递减),即以p[0]为分界点往正负两个方向计数,下面给出一个例子。

【例2.3】对p使用正负下标进行操作的程序。

```
#include <stdio.h>
int main ()
{
    int a = 36, b = 63, c = 656, i=0;
    int *p = &b;
    printf ( "%d %d %d\n", p[1], p[0], p[-1]);
    for ( i=1; i>-2; i--)
        printf ( "%d ", p[i]);
    printf ( "\%d ", p[i]);
    printf ( "\n" );
    return 0;
}
```

### 程序输出结果如下:

36 63 656 36 63 656

## 2.2 指针移动

指针移动,就是对指针采取++和--操作。VC为整数分配4字节,所以p+1就是向高地址移动一个整数的长度,即4字节。字符只分配1字节,p+1就移动1字节。反之,--就代表地址减少。

1.顺序移动整数类型的指针

下面举例说明指针移动会带来的一些问题。

【例2.4】顺序移动指针的例子。

### 程序运行输出如下:

```
0x0012FF7C, 0x0012FF78, 0x0012FF74, 0x0012FF70, 0x0012FF6C
88 58 98 3 0012FF6C
0012FF6C
0 98 58 66 1245120 0012FF80 1245120
```

变量分配地址以a、b、c、i、p顺序降序分配: 0x0012FF7C~0x0012FF6C。使用p--方式,以降序0x0012FF7C~0x0012FF74顺序输出 "885898" 。下一个0x0012FF70是的地址,这时的i等于3,所以输出3。再往下是0x0012FF6C,这时p也指向这个地址,因此\*p就是0x0012FF6C。

可以验证一下, 将这个地址赋给变量a, 再取这个地址的内容, 验证结果正确。

现在从i开始往a移动(升序地址), a已经改为66, 所以依次输出

0 98 58 66

这时p出界了,输出是随机数据,这里是"1245120"。可以将a的地址加4赋给c来验证,这个地址内的内容也是"1245120"。

由此可见,用一个指针可以到处"跑",如果用法不正确,后患无穷。

【例2.5】用顺序移动指针说明危险性的例子。

#### 程序输出结果如下:

```
0x0012FF7C, 0x0012FF78, 0x0012FF74, 0x0012FF70, 0x0012FF78
88, 58, 98, 0, 58
0x0012FF7C, 0x0012FF78, 0x0012FF74, 0x0012FF70, 0x0012FF76
88, 58, 98, 0, 3801088
88, 0, 809042018, 0, 13579
```

#### 使用语句

p=(int \*)0x0012FF76;

虽然可行,但取出的内容是原来存储的数据,这里已经不是按分配的地址读取,所以是无意义的数据"3801088"。不过此时的变量b和c的内容还没有被破坏。

使用语句 "\*p=13579; "则完全破坏了变量b和c的内容。最后一行的输出验证了这个问题。b变成0,而c变成 "809042018",这条语句完全破坏了原来的存储内容。

【例2.6】演示使用字符指针存取整数的例子。

```
#include <stdio.h>
int main()
{
   int a=0x12345678, i=0;
   char *p;
   p=(char *)0x0012FF7C;
   printf("%#p, %#X\n", sa, p);
   for( i=0; i<4; i++, p++)
        printf("%#p, %#x\n", p, *p);
   return 0;
}</pre>
```

#### 程序输出结果如下:

```
0x0012FF7C, 0x12FF7C
0x0012FF7C, 0x78
0x0012FF7D, 0x56
0x0012FF7E, 0x34
0x0012FF7F, 0x12
```

系统为整型数据分配4字节、字符1字节,用字符指针逐个读取1字节的内容,可以发现,0x0012FF7C~0x00012FF7F存取0x78~0x12,即指针指向存储整数的低字节位置,具有连续的4字节。

所以说指针变量存储的是指向变量a的地址值,而不说是存储变量a的地址。

### 2.指针基本运算

本节所说的C语言的地址能进行某种运算,即指针可以与整数相加减。若p为指针,n为整数,则可以使用p+n或p-n。但这里必须弄清楚,编译程序在具体实现时并不是直接将n的值加到p上,而是要将n乘上一个"比例因子",然后再加上p。这是因为不同类型的数据实际存储所占的单元数不同,如char类型为1字节,int类型为2字节(VC为4字节),long和float类型为4字节,double类型为8字节等,这些数分别为它们的"比例因子",具体采用哪个作为比例因子,取决于p指向的数据是什么类型。对用户来说,不需要了解编译程序内部的实现,只要将p+n看成将指针p移动n个数(不必涉及每个数所占的字节数)的位置。下面通过实例来说明需要注意的几个问题。

### 【例2.7】演示指针及其运算概念的例子。

上面程序在VC中实现,为了更容易理解,表2-1给出VC在执行第1条语句之后,为各个变量分配的内存首地址。

为了更容易理解,在程序输出的右方给出输出前的操作过程及输出序号。

### 程序输出结果如下:

#### 表2-1 内存分配一览表

| 变 量 | 代表的值或地址    | 分配的首地址     | 备 注  |
|-----|------------|------------|------|
|     | 随机数        | 0x0012FF80 | 非程序区 |
| X   | 56         | 0x0012FF7C |      |
| у   | 65         | 0x0012FF78 |      |
| p   | 0x0012FF7C | 0x0012FF74 | 指针变量 |

#### 使用中需要注意如下问题:

- 1)系统根据变量x和y及指针的声明顺序,为它们分配一段连续的地址。内存首地址的关系及其所代表的含义如表2-1所示,第3列0x0012FF80是紧挨x上方的地址(x占4字节),内容为随机数;其他3个地址分别是第1行变量的存储首地址。第2列的值分别是第1列变量的值,其中0x0012FF7C为存储变量x的首地址,即p的值。
  - 2) 将p改为指向y,这就改变了p和\*p的值,p指向y的地址而\*p=y。当然&p是不会改变的,如第3行输出所示。
  - 3) 使用"\*p=66;"语句也同步改变了y的值,但p的指向不变,见第4行输出。
  - 4)对p进行--p操作时,因为本程序的y和p顺次存放,所以就使得p指向自己,即第5条输出语句中的&p和\*p均与p一样,都是输出0x0012FF74。
  - 5) 当对p进行++p操作时,使指针从指向p变为指向y的内存存放首地址,\*p也随之变化并为y的值,操作产生的影响见第6行的输出。
  - 6) 再次对p进行++p操作时,使指针从指向y变为指向x的内存存放首地址,\*p也随之变化并为x的值,操作产生的影响见第7行的输出。
- 7)如果这时继续执行++p操作,指针指向非程序区的地址0x0012FF80,其中的内容为随机数。第8行的输出证实了这一点。此时p所指向的地址虽然唯一,但已经不是所需内容。如果这个内容很重要,又不慎将它修改,就会造成灾难性后果。这就是使用指针的危险之处。
  - 8) 执行--p使p退回安全区并指向x的地址,这时第9行的输出就与第2行的一样。
- 9)对p进行运算,相应的\*p为p指向地址的内容。也可以不改变p,将相对p的地址的内容取出,这就是使用\*(p±n)。程序演示了使用\*(p-1)输出6,但这并没有改变\*p的内容,这可从第10行输出\*p的结果得到证实。最后一句使用语句"\*p=\*(p-1);"将p指向的内容改变为66,但p仍然指向x。因此要正确区别(p±n)和\*(p±n)操作。

可使用下标"[]"描述连续的指针p,这里不再赘述。

### 3.指针永远指向一个地址

迄今为止都是将指针进行初始化了,在讨论中也是假定已经将指针初始化了。下面看一个简单的例子。

### 【例2.8】指针没有赋初值的例子。

```
#include <stdio.h>
int main ()
{
    int *p;
        *p=65;
        printf ( "%d\n", p );
        return 0;
}
```

编译给出信息 "warning C4700: local variable'p'used without having been initialized" 。

指针没有指向一个地址,即产生运行时错误。有人认为改为"p=65;"是正确的,因为这时的输出结果为65。其实不对,这时输出的是地址,"p=65;"是将一个十进制数65作为地址,如果将打印语句改为

```
printf ( "%#p\n", p );
```

则输出0x00000041,这就是十六进制地址,41代表十进制65。这种做法就是将一个无效地址赋给指针,将会产生灾难性的后果。

由此可见,编译系统给出警告时,首先应该采取有效措施来消除这个警告。

因为指针变量存放的是地址,所以必须有具体指向。最常见的错误是声明了指针,没有为指针赋值。没有赋值的指针含有随机地址。可以将上面的赋值语句去掉,直接输出p以验证这一点。因为指针的破坏性很大,所以尽可能在声明时同时初始化指针,这种习惯能避免指针的遗漏赋值。由此可见,不仅要为指针赋一个地址,而且这个地址应是有效地址。



指针应该指向一个有效的地址。

## 2.3 指针地址的有效性

### 1.地址的有效性

计算机的内存地址是有一定构成规律的。能被CPU访问的地址才是有效的地址,除此之外都是无效的地址。

假设有一个指针变量p。可以随便将一个地址赋给p,只要转换匹配一下即可,p是"来者不拒",并不"考虑"给它赋的是什么值,更不"考虑"其后果。声明一个指针,必须赋给它一个合理的地址值,请看下面的例子。

#### 【例2.9】演示给指针赋有效和无效地址的例子。

```
#include <stdio.h>
int main()
{
    char *p,a='A',b='B';
    p=6a;
    printf("0x*p, %c\n", p,*p);
    p=(char *)0x0012FF74;
    printf("0x*p, %c\n", p,*p);
    p=(char *)0x0012FF78;
    printf("0x*p, %c\n", p,*p);
    p=(char *)0x0012FF7C;
    printf("0x*p, %c\n", p,*p);
    p=(char *)0x012FF7C;
    printf("0x*p, %c\n", p,*p);
    p=(char *)0x1234;
    printf("0x*p\n", p);
    printf("%c\n", *p);
    return 0;
}
```

#### 编译正确,但运行时出现异常。下面是除最后一条输出语句之外的输出结果。

```
0x0012FF78, A
0x0012FF74, B
0x0012FF78, A
0x0012FF7C, |
0x00001234
```

当指针被赋予字符A的地址时,指针地址不仅有效,且\*p具有确定的字符A。当将p改赋地址0x0012FF74时,这个地址恰恰是系统分给字符B的地址,这个地址不仅有效,且\*p具有确定的字符B。有时地址有效,但内容不一定确定,如0x0012FF7C是有效地址,但程序没有使用这个地址,所以决定不了它的内容,输出字符"|"是无法预知的。地址0x1234虽然能被指针p接受,也能输出这个地址,但这个地址是无效的,所以执行语句

```
printf("%c\n", *p);
```

时出错,产生运行时错误。也就是当赋一个无效的地址给p时,就不能对\*p进行操作。



使用指针必须对其初始化,并且给指针赋予有效的地址。

#### 2.指针本身的可变性

编译系统为变量分配的地址是不变的,为指针变量分配的地址也是如此,但指针变量所存储的地址是可变的。

#### 【例2.10】有如下程序:

### 假设运行后,第6和第7两行给出如下输出信息。

```
0x0012FF7C, 0x0012FF78, 0x0012FF74
0x0012FF6C, 0x0012FF7C, 0x0012FF7C, 15
```

### 请问能分析出程序后面的输出结果吗?

【解答】因为地址0x0012FF80里存储的值不是由程序决定的,所以这个输出值不能确定。除此之外,其他的输出值均可以根据这两行的输出结果,写出确定的输出结果。

为了便于分析,首先要清楚所给上述两行输出结果的含义。

- 1) 从第一行的输出可知,依次是分配给变量a、b和c的地址。
- 2) a的地址是0x0012FF7C。注意第2行的输出中,第2个和第3个的值与它相等。
- 3)第2行第1个0x0012FF6C对应"&p",是编译系统为指针分配的地址,用来存放指针p。因为已经给指针变量赋值(p=&a),所以"\*&p"就是输出指针地址0x0012FF6C里的内容0x0012FF7C。它就是p指向a的地址,即也输出0x0012FF7C。也就是说,\*&p、p和&a的值相同。
  - 4) "\*p" 就是输出指针p所指向地址0x0012FF7C里所存储的变量a的值,即15。

要分析输出,需要掌握如下操作含义。

- 1)编译系统为声明的变量a分配存储地址,运行时可以改变a的数值,但不会改变存储a的地址,即&a的地址值不变。同理,为声明的指针变量p分配一个存储地址,p指向的地址值可以变化,但&p的地址不会变化。
  - 2)可以对指针变量p做加、减操作。由第1行输出结果知,p=p-1(可记作--p),则p指向的地址是0x0012FF78,\*p输出38,再执行p--,则\*p输出35。如果再执行p++,则\*p输出38。这时,对p操

作后,不仅p指向的地址有效,其地址中存储的内容也正确。

3) 如果p的操作超出这三个变量的地址, 就无法得出输出结果。

按照上述提示, 预测如下。

- 1) 第8~10行中的for语句就是输出三个变量的值(153835),输出之后,可以预测p指向地址为0x0012FF70,但不能预测\*p的内容。在运行过程中&p保持为0x0012FF6C。
- 2) 第11~13行中的for语句是反向输出三个变量的值(353815),输出之后,可以预测p指向地址为0x0012FF80,但不能预测\*p的内容(假设它的值为1245120),当然&p仍为0x0012FF6C。
- 3) 第14~17行中的for语句也是输出三个变量的值(153835),第14行将p调整指向存储a的地址,循环语句中使用"\*(p-i)",因为只是使用p做基准,用i做偏移量,所以p的值不变,输出之后,p不变,仍为0x0012FF7C,\*p=15,&p不变。
  - 4) 第18~20行中的for语句是反向输出三个变量的值(353815),循环语句也使用p做基准,即"\*(p-2+i)"。输出之后,p不变,仍为0x0012FF7C,\*p=15,&p不变。

由此可见,要小心对p的操作,以免进入程序非使用区或无效地址。如果使用不当,严重时会使系统崩溃,这是使用指针的难点之一。

程序实际运行结果如下。

```
0x0012FF7c,0x0012FF78,0x0012FF74
0x0012FF6C,0x0012FF7C,0x0012FF7C,15
15 38 35
3,0x0012FF70,0x0012FF6C
35 38 15
15,0x0012FF80,0x0012FF6C // 1245120是不可预测的值
15 38 35
15,0x0012FF7C,0x0012FF6C
35 38 15
15,0x0012FF7C,0x0012FF6C
```

#### 3.没有初始化指针和空指针

#### 【例2.11】没有初始化指针与初始化为空指针的区别。

```
#include <stdio.h>
int main()
{
    int a=456;
    int "p;
    printf("指针没有初始化: \n", p,&p);
    printf("指针没有初始化: \n", p,&p);
    p=NULL;
    printf("8材没有初始化为NULL: \n", p,&p);
    printf("8材没有初始化为NULL: \n", p,&p);
    printf("0x%p,0x%p\n", p,&p);
}
```

#### 运行结果如下。

```
指针没有初始化:
0xCCCCCCC, 0x0012FF78指针初始化为NULL:
0x0000000, 0x0012FF78
```

显然,在这两种情况下,不管如何初始化指针,&p分配的地址是一样的,区别是指针变量存放的值。

指针在没有初始化之前,指针变量没有存储有效地址,如果对"\*p"进行操作就会产生运行时错误。当用NULL初始化指针时,指针变量存储的内容是0号地址单元,这虽然是有效的地址,但也不允许使用"\*p",因为这是系统地址,不允许应用程序访问。

为了用好指针,应养成在声明指针时就予以初始化。既然初始化为NULL也会产生运行时错误,何必要选择这种初始化方式呢?其实,这是为了为程序提供一种判断依据。例如申请一块内存块,在使用之前要判断是否申请成功(申请成功才能使用)。

```
int *p=NULL;
p=(int*)malloc(100);
if(p=NULL);{
printf("内存分配错误! \n");
exit(1); // 结束运行
}
```

注意正确地包含必要的头文件,下面给出一个完整的例子。

### 【例2.12】判断空指针的完整例子。

```
#include <stdlib.h>
#include <stdio.h>
void main()
{
    char *p;
    if( (p = (char *)malloc(100) ) == NULL) {
        printf("内存不够! \n");
        exit(1);
    }
    gets(p);
    printf("%\n", p);
    free(p);
}
```

其实,只要控制住指针的指向,在使用中就可避免出错。

将它与整型变量进行对比,就容易理解指针的使用。整型变量存储整型类型数据的变量,也就是存储规定范围的整数。指针变量存储指针,也就是存储表示地址的正整数。由此可见,一个指针变量的 值就是某个内存单元的地址,或称为某个内存单元的指针。可以说,指针的概念就是地址。

由此可见,通过指针可以对目标对象进行存取(\*操作符),故又称指针指向目标对象。指针可以指向各种基本数据类型的变量,也可以指向各种复杂的导出数据类型的变量,如指向数组元素等。

指针初始化,就是保证指针指向一个有效的地址。这有两层含义,一是保证指针指向一个地址,二是指针指向的地址是有效的。

#### 1.数值表示地址

为了更深入地理解这一点,首先要记住指针是与地址相关的。指针变量的取值是地址值,但如何用数值来代表地址值呢?请看下面的例子。

【例2.13】演示表示地址值的例子。

```
#include <stdio.h>
int main()
{
   int a=256,b=585;
   printf ("$#p, $#p\n", 6a, 6b);
   printf ("$d, $d\n", *aa, *ab);
   printf ("$d, $d\n", * (int *) 0x0012FF7C, * (int *) 0x0012FF78);
   return 0;
}
```

#### 程序输出结果如下.

```
0x0012FF7C,0x0012FF78
256,585
256,585
```

"&a"表示变量a的地址,"\*&a"表示存入地址里的值,也就是变量a的值。既然&a代表的是存储变量a的十六进制地址0x0012FF7C,是否可以使用"\*0x0012FF7C"呢?

答案是否定的,编译器无法解释"\*0x0012FF7C"。要想让"0x0012FF7C"表示地址,必须显式说明,即让它与指针关联起来,可使用"int\*"将这个十六进制数字强制转换成地址。这就是通常说的"地址就是指针"的含义。

由此可见,直接使用数值很麻烦。即使对于验证过的地址,换一台机器可能就不行了,不具备可移植性,这种赋值乃是不得已而为之。

#### 2.赋予有效和无效地址

【例2.13】不仅演示了如何使用地址值,还演示了有效地址的概念。该例是在确定变量地址之后才使用它们,所以是有效的。一般认为,所谓地址有效是指在计算机能够有效存取的范围内。由此可见,这个有效并不能保证程序正确,指针超出本程序使用的地址范围可能带来不可估计的错误。

为了保证赋予的地址有效,避免像上面例子那样直接使用强制转换,而是直接使用变量地址赋值。例如:

```
int a=256,b=585,*p=&b;
```

#### 或者使用语句

```
int a=256,b=585,*p;
p=&b;
```

由此可见,对于一个指针,需要赋给它一个地址值。上面的例子在赋给指针地址时,不是随意的,而是经过挑选的。如果随便选一个地址,可能是计算机不能使用的地址,也就是无效的地址。

【例2.9】演示了无效地址的例子。对于无效的地址,虽然编译没问题,但却产生运行时错误。由此可见,使用指针的危险性就是赋予它一个无效地址,如果有效地避免了这一点,就可以运用自如。

### 3.无效指针和NULL指针

编译器保证由0转换而来的指针不等于任何有效的指针。常数0经常用符号NULL代替,即定义如下:

```
#define NULL 0
```

当将0赋值给一个指针变量时,绝对不能使用该指针所指向的内存中存储的内容。NULL指针并不指向任何对象,但可以用于赋值或比较运算。除此之外,任何因其他目的而使用NULL指针都是非法的。因为不同编译器对NULL指针的处理方式不相同,所以要特别留神,以免造成不可收拾的后果。

如上所述,将指针初始化为NULL,就是用0号地址初始化指针,而且这个地址不允许程序操作,但可以为编程提供判别条件。尤其是在申请内存时,假如没有分配到合适的地址,系统将返回NULL。

C编译程序都提供了内存分配函数,最主要的是malloc和calloc,它们是标准C语言函数库的一部分,功能都是为要写的数据在内存中分配一个安全区。一旦找到一个大小合适的内存空间,就分配给它们,并将这部分内存的地址作为一个指针返回。malloc和calloc的主要区别是:calloc清除所分配的内存中的所有字节,即将所有字节置零;malloc仅分配一块内存,但所有字节的内容仍然是被分配时所含的随机值。

malloc和calloc所分配的内存空间都可以用free函数释放。这3个函数的原型在文件stdlib.h中,但很多编译器又放在头文件malloc.h中,注意查阅手册。

在目前所提供的最新C编译程序中,malloc和calloc都返回一个void型的指针,也就是说,返回的地址值可以假设为任何合法的数据类型的指针。这个强制转换可以在声明中进行,如将它们声明为字符型、整型、长整型、双精度或其他任何类型。

### 【例2.14】找出程序中的错误并改正。

```
#include <stdlib.h>
#include <stdlio.h>
int main ()
{
    char *p;
    *p = malloc(200);
    gets(p);
    printf(p);
    free(p);
    return 0;
}
```

malloc所返回的地址并未赋给指针p,而是赋给了指针p所指的内存位置。这一位置在此情况下也是完全未知的。下面语句

```
char *p;
*p = ( char *) malloc(200);
```

```
char *p;
p = malloc(200);
```

的方式,则是可以的,但它也有另外一个更为隐蔽的错误。如果内存已经用完了,malloc将返回空(NULL)值,这在C语言中是一个无效的指针。正确的程序应该将对指针的有效性检查加入其中,并及时释放不用的动态内存空间。下面是一个正确而完整的实例。

```
#include <stdlib.h>
#include <stdlib.h>
#include <stdlio.h>
int main ()

{
    char *p;
    p = (char *) malloc(200);
    if(p==NVLL) {
        printf ("内存分配错误! \n");
        exit(1);
    }
    gets(p);
    printf(p);
    free(p);
    return 0;
}
```

在设计C程序时,对指针的初始化有两种方法:使用已有变量的地址或者为它分配动态存储空间。好的设计方法是尽可能地早点为指针赋值,以免遗忘造成使用未初始化的指针。

对于上述程序, 如果设置

```
p =NULL;
```

则程序执行if语句"{}"里的部分,输出"内存分配错误!",然后退出程序。在程序设计中,有时正好利用"NULL"作为判别条件。下面就是一个典型的例子。

#### 【例2.15】完善下面的程序。

```
#include <stdio.h>
char s1[16];
char *mycopy(char *dest,char *src) {
    while (*dest++=*src++);
    return dest;
}
void main () {
    char s2[16]="how are you?";
    mycopy(s1,s2);
    printf(s1);
    printf("\n");
}
```

这个程序编译没有错误,但不够完善。这主要是因为mycopy函数中没有采取措施预防指针为NULL(又称0指针)的情况。解决的方法很多,下面是简单处理的例子。

```
char *mycopy(char *dest,char *src)
{
    if(dest == NULL || src == NULL)
        return dest;
    while (*dest+=*src++);
    return dest;
}
```

由此可见,使用已有变量的地址初始化指针能保证地址总是有效的。如果使用分配动态存储空间的方法来初始化指针,确保地址有效的方法是增加判断地址分配是否成功的程序段。

## 2.5 指针相等

假设有两个指针\*p1和\*p2, 一定要理解语句

```
*p1=*p2;
P1=p1;
```

的含义。为了说明这个问题,先介绍大端存储和小端存储的概念。

### 1.大端存储和小端存储

在CPU内部的地址总线和数据总线是与内存的地址总线和数据总线连接在一起的。当一个数从内存中向CPU传送时,有时是以字节为单位,有时又以字(4字节)为单位。传过来是放在寄存器里(一般是32字节),在寄存器中,一个字的表示是右边应该属于低位,左边属于高位,如果寄存器的高位和内存中的高地址相对应,低位和内存的低地址相对应,这就属于小端存储。反之则称为大端存储。大部分处理器都是小端存储的。

因为十六进制的2位正好是1字节,所以选十六进制0x0A0B0C0D为例,如图2-1所示,对小端存储,低位是0x0D,应存入低位地址,所以存入的顺序是

0x0D 0x0C 0x0B 0x0A

### 反之,对于大端存储则为

0x0A 0x0B 0x0C 0x0D

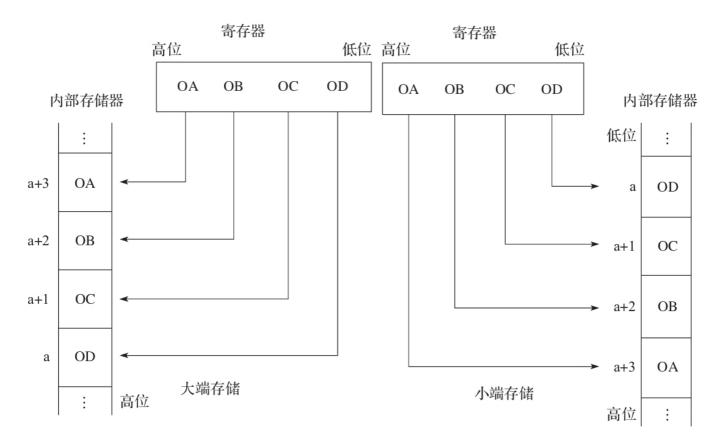


图2-1 图解大端和小端存储

下面利用union的成员共有地址的性质,用一个程序来具体说明小端存储。

#### 【例2.16】演示小端存储的程序。

```
#include <stdio.h>
union s{
    int a;
    char sl[4];
}uc;
int main() {
    int i=0;
    uc.a=0x12345678;
    printf("0x*x\n", suc);
    for(i=0);<4;;++)
        printf("0x*x \n", suc.sl[i], uc.sl[i]);
    return 0;
}</pre>
```

声明十六进制整数a,它与字符串数组共有地址,a的最低字节是0x0D,按小端存储,则应存入"&uc.s1[0]"中,也就是0x4227A8中,最高位地址0x4227AB则应存入0x0A,也就是数据的高位。下面的运行结果证明了这一点。其实,可以在调试环境中直接看到这些结果。

```
0x4237A8 0xD
0x4237A8 0xD
0x4237A9 0xC
0x4237AA 0xB
0x4237AB 0xA
```

其实, 【例2.6】的程序也说明了这个问题。

### 2.指针相等操作

两个指针变量相等,是指它们指向同一个地址。例如

p1=p2;

不仅使得p1和p2都指向原来p1指向的地址,而且保证\*p2=\*p1。注意它们的值是原来\*p1的值。也就是说,p2放弃自己原来的指向地址及指向地址里存储的值。而语句

\*p1=\*p2;

的作用是使p1放弃自己原来的指向地址里存储的值,但并没有放弃自己的指向地址。p1和p2仍然保留各自原来的指向。至于\*p1里面的值,则要视具体情况而定,不能由此得出p1指向地址里的值与\*p2相等。关于这一点,只要用两个不同结果的例子就可以说明这一点。

### 【例2.17】演示整数指针相等操作的程序。

```
#include <stdio.h>
int main()

{
    int *pl, *p2;
    int sl=0xl2345678,s2=0x78;
    pl=6s1;p2=6s2;
    printf("0x8x\t0x8x\n",p1,p2);
    *p2=*p1;
    printf("0x8x\t0x8x\n",*p2,*p1);
    printf("0x8x\t0x8x\n",p1,p2);
    // 值相等
    printf("0x8x\t0x8x\n",p1,p2);
    // 地址不变
```

```
p2=p1;
printf("0x%x\t0x%x\n",*p1,*p2); // 值相等
printf("0x%x\t0x%x\n",p1,p2); // 地址改3
return 0;
```

这个例子的语句"\*p2=\*p1; "使用\*p1取代\*p2,但p2不变。运行结果证明了这一点。

```
0x12ff74 0x12ff70
0x12345678 0x12345678
0x122ff74 0x12ff70
0x12345678 0x12ff70
0x12345678 0x12345678
0x12ff74
```

#### 【例2.18】演示字符指针相等操作的程序。

```
#include <stdio.h>
int main()
{
    char *pl, *p2;
    char s![16]="987654321",s2[16]="G";
    pl=s![p2=s2;
    printf("0x&x\tox*x\n",pl,p2);
    *p2=p1;
    printf("%x\tox*x\tox*x\n",pl,p2);
    printf("%x\tox*x\tox*x\n",pl,p2);
    // 地址不变
    p2=p1;
    printf("%x\tox*x\n",pl,p2);
    // 地址不变
    p2=p1;
    printf("%x\tox*x\n",pl,p2);
    // 地址改变
    return 0;
}
```

这个例子的语句"\*p2=\*p1;"并不使用\*p1取代\*p2,只是使用字符"1"取代原来的字符"G",运行结果如下:

```
0x12ff68 0x12ff58
1 987654321
0x12ff68 0x12ff58
987654321 987654321
0x12ff68 0x12ff68
```

这是字符操作特征引起的,所以不能只看表面现象。以后还会做进一步的分析。

### 2.6 对指针使用const限定符

可以用const限定符强制改变访问权限。用const正确地设计软件可以大大减少调试时间和不良的副作用,使程序易于修改和调试。

1.指向常量的指针

如果想让指针指向常量,就要声明一个指向常量的指针,声明的方式是在非常量指针声明前面使用const,例如:

```
const int *p; // 声明指向常量的指针
```

因为目的是用它指向一个常量,而常量是不能修改的,即\*p是常量,不能将\*p作为左值进行操作,这其实是限定了"\*p="的操作,所以称为指向常量的指针。当然,这并不影响p既可作为左值,也可作为右值,因此可以改变常量指针指向的常量。下面是在定义时即初始化的例子。

指向常量的指针p指向常量y,\*p和y都不能作为左值,但可以作为右值。

如果使用一个整型指针p1指向常量y,则编译系统就要给出警告信息。这时可以使用强制类型转换。例如:

```
      const int y=66;
      // 常量y不能作为左值

      int *p1;
      // *p1既可以作为左值,也可以作为右值

      p1=(int *)&y;
      // 因为 y是常量,p1不是常量指針,所以要将&y进行强制转换
```

如果在声明p1时用常量初始化指针,也要进行转换。例如:

```
int *pl=(int *)&v; // 因为 v是常量、pl不是常量指针、所以要将&y进行强制转换
```

在使用时,对于常量,要注意使用指向常量的指针。

### 2.指向常量的指针指向非常量

因为指向常量的指针可以先声明,后初始化,所以也会出现在使用时将它指向了非常量的情况。所以这里用一个例子演示一下如果指向常量的指针p指向普通变量,则会出现什么结果。

【例2.19】使用指向常量的指针和非常量指针的例子。

```
#include <stdio.h>
int main()

{
    int x=55;
    const int y=88;
    const int *p;
    int *p1;
    j=6;
    p=6;
    printf("%d",*p);
    p=6x;
    printf("%d",*p);
    x=128;
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d",*p);
    printf("%d\n",*p1);
    return 0;

}

// 変型x能作为左值。但可以作为右值
    // 声明指针
    // 声明指针
    // 声明指针
    // 声明指针
    // P作为左值,使常量指针及为指向变量x,*p不能作为左值
    // 用文件为左值。有效变*p的值,使*p=x=128
    // 非常量指针指向常量需要强制特换
    return 0;
```

#### 运行结果如下。

```
88 55 128 88
```

使用指向常量的指针指向变量时,虽然\*p不能作为左值,但可以使用"x="改变x的值,x改变则也改变了\*p的值,也就相当于将\*p间接作为左值。所以说,"const"仅是限制直接使用\*p作为左值,但可以间接使用\*p作为左值,而\*p仍然可以作为右值使用。

与使用非常量指针一样,也可以使用运算符"&"改变常量指针的指向,这当然也同时改变了\*p的值。

必须使用指向常量的指针指向常量,否则就要进行强制转换。当然也要避免使用指向常量的指针指向非常量,以免产生操作限制,除非是有意为之。

以上结论可以从程序的运行结果中得到验证。

#### 3. 常量指针

将const限定符放在\*号的右边,就使指针本身成为一个const指针。因为这个指针本身是常量,所以编译器要求给它一个初始化值,即要求在声明的同时必须初始化指针,这个值在指针的整个生存期中都不会改变。编译器将"p"看作常量地址,所以不能作为左值(即"p="不成立)。也就是说,不能改变p的指向,但\*p可以作为左值。

#### 【例2.20】使用常量指针的例子。

#### 运行结果如下。

45 100 55 100 300 100

语句 "x=y; " 和 "\*p=sum; " 都可以改变x的值, 但p指向的地址不能改变。

显然,常量指针是指这个指针p是常量,既然p是常量,当然p不能作为左值,所以定义时必须同时用变量对它进行初始化。对常量而言,需使用指向常量的指针指向它,含义是这个指针指向的是不能作为左值的常量。不要使用常量指针指向常量,否则就需要进行强制转换。

### 4.指向常量的常量指针

也可以声明指针和指向的对象都不能改动的"指向常量的常量指针",这时也必须初始化指针。例如:

```
int x = 2;
const int* const p= &x;
```

告诉编译器,\*p和p都是常量,都不能作为左值。这种指针限制了"&"和"\*"运算符,所以在实际应用中很少用到这种特殊指针。

### 5.void指针

在一般情况下,指针的值只能赋给相同类型的指针。void类型不能声明变量,但可以声明void类型的指针,而且void指针可以指向任何类型的变量。

### 【例2.21】演示void指针的例子。

```
#include <stdio.h>
int main()

{
    int x=256, y=386, *p=6x;
    void *vp = 6x;
    void *vp = 7x;
    void *vp = 1x;
    voi
```

虽然void指针指向整型变量对象x,但不能使用\*vp引用整型对象的值。要引用这个值,必须强制将void指针赋值给与值相对应的整型指针类型。程序输出如下。

```
0x0012FF7C,0x0012FF7C,256
0x0012FF78,0x0012FF78,386
```

## 2.7 使用动态内存

可以自己申请一块内存,且这块内存也由自己释放,这样分配的内存称为动态内存。

如何使用这种动态内存,也就是如何定位这些动态内存的地址呢?我们知道,与地址有关的变量是指针变量,所以可以使用指针对动态内存进行操作。

## 第3章 一维数组

在C语言中,数组是一个非常重要的概念,一定要熟练掌握数组的用法,尤其是深刻理解数组的边界不对称并避免数组越界错误。

### 3.1 一维数值数组

利用基本数据类型能构造出相应数据类型的数组,所以说数组是一种构造型数据类型。本节重点以一维整数数组为例,讨论它们的特征,然后再推广到其他一维数组。

#### 1.一维数值数组的特征

假如要表示5个连续整数变量1~5,则需要5个整数变量名称,如用X1~X5表示。这批变量的特点是它们的基本数据类型一样。现在构造一个新的数据类型,假设它的名称为A,在符号"[]"内用序号表示为A[0]~A[4],它们的对应关系如图3-1所示。

| X1   | X2   | X3   | X4   | X5   |  |
|------|------|------|------|------|--|
| 1    | 2    | 3    | 4    | 5    |  |
| A[0] | A[1] | A[2] | A[3] | A[4] |  |

图3-1 数组构成示意图

后一种表示有很大的进步。X1~X5之间没有内在关系,而A[0]~A[4]之间是通过"[]"内的序号0~4(也就是下标)构成了唯一的连续对应关系。如果将A[0]~A[4]看作连续的房间,则可以改变房间里所存放整数的值。暂且将A称作整数房间。也就是房间里必须都存放整型变量,这样构造出来的数据类型称为整型数组。

### 假定使用语法定义如下:

```
int A[5]={1, 2, 3, 4, 5};
```

这条语句的含义是整型数组A的下标从0开始,A[4]的值为5,A[5]本身不是数组的元素。这个数字5代表数组A共有5个元素A[0]~A[4]。将若干个同类型变量线性地组合起来,就构成一维数组。在使用一维数组之前首先必须声明它,这包括数组的类型、数组名和数组元素的个数。

### 声明一维数组的一般方式如下:

datatype array\_name[n];

其中,datatype是数据类型,可以是基本数据类型,也可以是构造类型。array\_name为数组名(标识符),n为数组中所包含的数组元素的个数。数组与各种基本数据类型的不同之处是:数组须由datatype、数组标志"[]"及长度n三者综合描述,它是在基本数据类型基础上导出的一种数据类型。例如:

### 数组中各元素总是从0开始连续编号,直到n-1为止。所以上述定义的数组a、b、c的数组元素分别为

```
a[0], a[1], a[2], ..., a[9]
b[0], b[1], b[2], b[3], b[4], b[5]
c[0], c[1], c[2], ..., c[24]
```

### 对数组中的任何元素都可单独表示并对它进行访问。假设a是整型数组,则语句

printf("%d",a[5]);

输出a[5]的值。C语言中规定只能对数组的元素操作,而不能对整个数组操作,即语句

printf("\$d",a);

### 是不允许的,必须使用for语句依次遍历整个数组元素。例如遍历数组a:

数组下标可以是变量,这个变量称为下标变量。下标变量的值原则是从0开始到n-1的整数,但是在C语言中对数组的下标变量的值是不进行合法性检查的,所以允许数组的下标越界,对此程序员必须引起注意并避免产生错误。具有丰富程序设计经验的程序员有时会巧妙地利用数组的下标越界来进行程序设计。

【例3.1】给出求斐波那契数列中前20个元素值的问题。所谓斐波那契数列,就是可以将它表示成 $F_0$ 、 $F_1$ 、 $F_2$ 、…,其中除 $F_0$ 和 $F_1$ 以外,其他元素的值都是它们前两个元素值之和,即 $F_n$ = $F_{n-2}$ + $F_{n-1}$ ,而 $F_0$ 、 $F_1$ 分别为 $F_0$ 和 $F_1$ ,为 $F_0$ 0、声明整数数组fibonacci(20)来依次存於等於那契数列的前20个元素值。

#### 程序输出结果如下:

```
0 1 1 2 3
5 8 13 21 34
55 89 144 233 377
610 987 1597 2584 4181
```

实际上,斐波那契数列在数学和计算机算法研究领域有许多用途。斐波那契数列起源于"兔子问题": 开始有一对兔子,假设每对兔子每个月生下一对新的兔子,而每一对新生下来的兔子在出生后的 第2个月月底开始繁殖后代,而且这些兔子永远不死,那么在1年之后一共会有多少对兔子?这一问题的答案建立在这样一个事实上,即在第n个月结束时,有总数为Fn+2的兔子。所以,根据程序输出结 果。在第12个月结束时将一共有377对兔子。

数组的元素可像一般变量那样进行赋值、比较和运算。如同简单变量一样,在说明数组之后就能对它的元素赋值,方法是从数组的第1个元素开始依次给出初始值表,表中各值之间用逗号分开,并用一对花括号将它们括起来。例如:

```
int A[]= {1, 2, 3, 4, 5};
```

A数组元素的个数由编译程序根据初始值的个数来确定。如果数组很大,又不需要对整个数组进行初始化,数组初始值表可以只对前几个元素置初值,但这时必须指出数组的长度。例如,在【例3.1】中存放斐波那契数列的数组中前两个元素是已知的,而后面18个元素是要通过计算产生的,所以对此可初始化为:

```
int fibonacci[20] = {0, 1};
```

这样数组中的前两个元素分别为0和1, 而后18个元素皆为0。

如果没有明确进行初始化,则编译程序将外部型和静态型的数组初始化为0,而自动型数组的值不定。如不给自动数组置初值,程序中又没有使用它,编译系统会给出警告信息。

C语言允许自动型数组有初始值,它与外部型和静态型数组初始值都包含在定义语句的花括号里,每个初始值用逗号分隔。例如:

```
int d[5]={0,1,2,3,4};
```

如果定义时只对数组元素的前几个初始化,则其他元素均被初始化为0;若未进行初始化,则C编译程序使用如下规则对其进行初始化:

- 1) 外部型和静态型数组元素的初始值为0;
- 2) 自动型和寄存器型数组元素的初始值为随机数。

### 2.数组元素的地址

每个数组元素都有自己的地址,并且是按下标序号顺序排列的。

数组名就是数组的首地址,也就是第1个元素的地址。假设有整型数组a[3],则第1个元素的地址有两种表示方法,即a和&a[0]。VC为整数分配4字节,下一个元素&a[1]的首地址与&a[0]相差4字节。由指针的知识可知,a+1代表将a的地址移动一个元素的长度,即移动4字节。a+1就是&a[1]的首地址。由此类推,可以写出如下程序输出它们各个元素存储的首地址。

### 【例3.2】演示数组元素的地址。

### 程序运行结果如下:

```
0x0012FF70 0x0012FF70 0x0012FF70
0x0012FF7C 0x0012FF74 0x0012FF78
0x0012FF70 0x0012FF74 0x0012FF78
0x0012FF70 0x0012FF76 0x0012FF78
```

在编程中,这三种表示方法都会用到。但在使用时要注意,虽然&a也是数组的首地址,但&a+i的含义与a+i的并不一样,所以不能使用&a+i方式输出其他元素的地址。上面的程序中使用语句

```
for(i=0; i<3; i++)
printf("%#p ", &a + i);
```

输出地址,因为&a+0就是&a,所以第1个地址是对的。但&a+1则不是&a[1]的地址,编译系统会在数组a的最后一个地址+1,即在数组a的第3个元素的首地址0x0012FF78处移动4字节,即存储变量i 的首地址0x0012FF7C。同理,&a+2应从i的尾部0x0012FF80开始移动8字节,输出0x0012FF88。即最后这段程序的输出为

0x0012FF70 0x0012FF7C 0x0012FF88

有人说可以使用& (a+i) , 那也是错误的, 且在编译时就会出现错误。

#### 3.存取数组元素的值

a是数组第1个元素存储的首地址,则\*a就是这个地址存储的值。同理,数组第i个元素的值可以用下标表示为a[i],也可以表示为\*(a+i)。下面的程序演示了两者的对应关系。

【例3.3】演示数组元素的存取。

#### 程序运行结果如下:

```
50 50
50 51 52
50 51 52
```

运行结果也证实了上述结论。这两种方法都要熟练掌握。

## 3.2 一维字符串数组

字符数组是数组的一个特例。它除了具有一般数组的性质之外,还具有自己的一些特点。它存储的是一串字符序列,其中还包括转义字符序列。在字符串的最后位置上存放一个标记串结束的字符(ASCII字符的0),即空字符。用转义字符\0'表示。字符串数组以\0'结束,它的长度应是存储字符长度加1。现在分析

```
int a[3];
char s[3];
```

这两个定义的区别。字符数组与整型数组的主要不同之处有如下几点。

- 1)整型数组a的各个单元的数值长度是可以不相同的,如a[0]=123,a[2]=12345,a[0]是3位数字,a[2]是5位数字。而字符数组s的各个单元都只能放一个字符,如s[0]='w',s[1]='e',w和e都是一个字符。
- 2)它们的有效长度不同,a[3]代表有3个数组元素,没有a[3]。而字符数组s中的s[2]='\0',存放的是字符串结束标志,实际可用的有效上界是s[2],即只能放2个字符。它的第s[2]个单元是定义的有效上界,当然也没有s[3]。
  - 3) 字符数组的初始化格式比较特殊,可以用字符串来代替,如:

```
char str[3] = { 'w', 'e', '\0'}; // 必须手工以'\0'结束
```

上述两种方式置初值是等价的。在对字符数组初始化时,为使程序能觉察到此数组的末尾,在内部表示中,编译程序要用'\0'来结束这个数组,从而存储的长度比双引号之间的字符的个数多一个。在上面的具体例子中。"we"字符串中的字符数是2。而str的长度却是3。

4)与数值数组一样,字符串数组的名字就是字符串的首地址。不同的是,字符串是顺序存放并以'\0'作为结束标志。所以字符串既可以按顺序输出,也可以用名字整体输出。在读入字符串时,单个字符串中不能有空格。

【例3.4】演示字符串数组的例子。

字符串顺序存放,字符串名是其首地址。s输出的不是地址,而是地址里的内容。程序输出结果如下。

需要注意的是,输出字符串首地址格式可使用"%p"以十六进制输出,也可以"%u"或者"%d"以十进制格式输出,但不能使用

```
printf("%s\n",&s);
```

语句,这个语句仍然是输出字符串的内容。

【例3.5】演示使用不同格式输出字符串数组首地址的例子。

```
#include <stdio.h>
int main()
{
    char s[]="fish";
    printf("\frac{4}{2}p \frac{4}{2}p\frac{1}{2}n\frac{1}{2}, \frac{1}{2}s, \frac{1}{2}s;
    printf("\frac{1}{2}l) \frac{4}{2}n\frac{1}{2}s, \frac{1}{2}s;
    printf("\frac{1}{2}l) \frac{4}{2}n\frac{1}{2}s, \frac{1}{2}s;
    printf("\frac{1}{2}l) \frac{8}{2}n\frac{1}{2}s, \frac{1}{2}s;
    printf("\frac{1}{2}l) \frac{8}{2}n\frac{1}{2}s, \frac{1}{2}s;
    return 0;
}
```

#### 输出结果如下:

```
0x0012FF78 0x0012FF78
1245048 1245048
1245048 1245048
fish fish
```

## 3.3 使用一维数组容易出现的错误

使用数组最容易犯的错误是数组越界和初始化错误。本节的分析仅局限于一维数组。

## 3.4 综合实例

本节列举几个典型的例子, 说明数组的使用方法。

【例3.14】计算输入字符串中数码和字符出现次数的程序。

### 程序运行示范如下:

```
247909347674e ewfqgq\=.,
0 1 2 3 4 5 6 7 8 9
1 0 1 1 3 0 1 3 0 2
nw=12
```

本例依赖于数字的字符表示,switch语句将数字分离出来,而且此数字的值是s[j]-'0',这个值刚好落在数值0~9之间,恰好是nd[]的下标值。非数字则由nw计数。

【例3.15】统计输入串中每个数字、字母和其他字符的个数。

```
#include <stdio.h>
void main()
{
   int i, nother;
   char c;
   int ndigit[10], nchar[26];
   nother=0;
   for ( i=0; i<10; ++i )
       ndigit[i]=0;
   for ( i=0; i<26; ++i )
       nchar[i]=0;
   while ( c=getchar() ) != '\n' )
   if ( c >= '0' && <<= '9' )
       ++ndigit[c-'0'];
   else if ( c >= 'a' && <<= 'z' )
       ++nchar[c-'a'];</pre>
```

#### 程序运行示范如下:

```
1234567898734abcdeghujxyz=-@&^%321
digits=0 2 2 3 2 1 1 2 2 1
characters=1 1 1 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 1 1
other= 6
```

#### 测试语句:

```
if ( c>= 'a' && c<= 'z' );
```

用来确定c中字符是否是小写英文字母,如果是小写英文字母,则数值c-'a'这个值刚好落在数值0~25之间,恰好是nchar[]的下标值。大写字母和数字处理方法类似。

通过上述例子可以看出,数组的元素可像一般变量那样进行赋值、比较和运算。

【例3.16】求解6个元素的整型数组中正元素之和的程序。

#### 程序运行输出:

sum=64

【例3.17】A、B、C、D、E合伙夜间捕鱼,凌晨时都疲惫不堪,各自在河边的树丛中找地方睡着了。日上三竿,A第一个醒来,他将鱼平分作5份,将多余的一条扔回湖中,拿自己的一份回家去了;B 第二个醒来,也将鱼平分作5份,扔掉多余的一条,只拿走自己的一份;接着C、D、E依次醒来,也都按同样的办法分鱼。5人至少合伙捕到多少条鱼?每个人醒来后看到的鱼数是多少条?

【解题思路】假定A、B、C、D、E5人的编号分别为1、2、3、4、5,为了容易理解,让整数数组的标号直接与这5个人的序号对应,定义整数数组fish[6]。不使用fish[0],从而可以使用数组fish[k]表示第k个人所看到的鱼数。fish[1]表示A所看到的鱼数,fish[2]表示B所看到的鱼数……显然有如下关系:

fish[1]=5人合伙捕鱼的总鱼数

```
fish[2]= (fish[1]-1) ×4/5
fish[3]= (fish[2]-1) ×4/5
fish[4]= (fish[3]-1) ×4/5
fish[5]= (fish[4]-1) ×4/5
由此可以写出如下一般表达式:
fish[i]= (fish[i-1]-1) ×4/5 i=2, 3, 4, 5
```

这个公式可用于从已知A看到的鱼数去推算B看到的,再推算C看到的……现在能否倒过来,先知E看到的再反推D看到的……直到A看到的。为此将上式改写为

 $fish[i-1]=fish[i]\times 5/4+1$  i=5, 4, 3, 2

分析上式如下:

- 1) 当i=5时,fish[5]表示E醒来后看到的鱼数,该数应满足被5整除后余1,即fish[5]%5==1。
- 2) 当i=5时,fish[i-1]表示D醒来后看到的鱼数,该数既要满足fish[4]=fish[5]×5/4+1,又要满足fish[4]%5==1。显然,fish[4]只能是整数,这个结论同样可以用于fish[3]、fish[2]、和fish[1]。
- 3)按题意要求5人合伙捕到的最少鱼数,可以从小往大枚举,先设E所看到的鱼数最少为6条,即fish[5]初始化为6,之后每次增加5再试,直至递推到fish[1]且所得整数除以5之后的鱼数为1。根据上述思路,可以将程序分为3个部分:程序准备(包括声明和初始化)部分、递推部分和输出结果部分。

程序准备部分包含定义数组fish[6]并初始化为1,定义循环控制变量i并初始化为0。输出结果部分就是输出计算结果。以上两个部分都很简单,下面着重介绍递推部分的实现方法。

递推部分使用do...while直到型循环结构,其循环体又包含两部分:

- 1)枚举过程中的fish[5]的初值设置,一开始fish[5]=1+5;以后每次增5。也就是说,第1个边界条件是fish[5]=6,以后的边界条件是每次递增5。
- 2)使用一个for循环,i的初值为4,终值为1,步长为-1,该循环的循环体是一个分支语句,如果fish[i+1]不能被4整除,则跳出for循环(使用break语句);否则,从fish[i+1]计算出fish[i]。当由 break语句让程序退出循环时,意味着某人看到的鱼数不是整数,当然不是所求,必须令fish[5]加5后再试,即重新进入直到型循环do...while的循环体。当正常退出for循环时,一定是循环控制变量i从初值4,一步一步执行到终值1,每一步的鱼数均为整数;最后i=0,表示计算完毕,且也达到了退出直到型循环的条件。

#### 程序运行的输出结果如下:

第1个人看到的鱼是3121条。第2个人看到的鱼是2496条。第3个人看到的鱼是1996条。第4个人看到的鱼是1596条。第5个人看到的鱼是1276条。

## 第4章 指针与数组

在C语言中,数组和指针是两个非常重要的概念,一定要熟练掌握数组和指针的用法。虽然数组和指针是两个不相关的概念,但在使用时,却又存在难以割舍的关系,所以一定要掌握两者配合使用的方法。只有掌握这些内容,才能用好它们。

## 4.1 数组与指针的关系

C语言的数组名字就是这个数组的起始地址,指针变量存储地址,数组名就是指针常量,所以指针和数组有密切关系。任何能由数组下标完成的操作,也能用指针来实现。使用指向数组的指针,有助于产生占用存储空间小、运行速度快的高质量的目标代码。

指向数组的指针实际上指的是能够指向数组中任一个元素的指针。这种指针应当说明为数组元素类型。例如,程序中声明如下整型数组a

int a[5];

这时只要声明如下一个整型指针变量:

int \*pa;

就可以使用pa指向整型数组a中的任何一个元素。即 "pa=&a[i]" 代表下标为i的元素的地址。所以指向第一个元素的等效赋值语句为:

pa=&a[0];

但使pa指向a的第1个元素的最简单的方法是:

pa=a;

虽然这两种方法是等效的,但一般都使用"pa=a"的简单方式。

如果需要表示指针所指向的数组存储单元的内容,可使用"\*"运算符,例如:

\*pa=\*a[i];

就指向下标为i的数组元素。假设pa正指向数组a中的最后一个元素(pa=&a[4]),那么如下的赋值语句

a[4]=386;

也可以表示成等效的语句

\*pa=386;

在数组名和指针之间有一个区别,必须记住指针是变量,故pa=a或pa++都是有意义的操作。但数组名是指针常量,而不是变量,因此表达式a=pa、a++、pa=&a都是非法操作。假设指针现在指向a[0],则数组的第i个(下标为i)元素可表示为a[i]或\*(pa+i),还可使用带下标的指针pa,即pa[i]和\*(pa+i)的含义一样。若要将a[4]设置为123,下面语句是等效的:

```
a[4]=386; *(a+4)=386; *(pa+4)=386; pa[4]=386;
```

所以,在程序设计中,凡是用数组表示的均可使用指针来实现,一个用数组和下标实现的表达式可以等价地用指针和偏移量来实现。表4-1给出了一个数组元素的4种关系。

【例4.1】演示没有另外定义指针,直接使用数组名作为指针的例子。

```
#include <stdio.h>
void main()
{
    int i, a[]={1, 3, 5, 7, 8};
    *(a+4)=86;
    // 使用数组名a作为数组第1个元素的指针
```

```
for(i=0; i<5; ++i)
printf("%d %d ",a[i],*(a+i)); // 交替使用两种方式輸出
printf("\n");
```

#### 程序运行结果为

1 1 3 3 5 5 7 7 86 86

表4-1 指针与数组元素的关系

| 下标   | 数组名 | 指针   | 指针下标  | 四者逻辑关系                              |
|------|-----|------|-------|-------------------------------------|
| a[0] | a   | pa   | pa[0] | a[0] = *a = *pa = = pa[0]           |
| a[1] | a+1 | pa+1 | pa[1] | a[1] = *(a+1) = = *(pa+1) = = pa[1] |
| a[2] | a+2 | pa+2 | pa[2] | a[2] = *(a+2) = = *(pa+2) = = pa[2] |

(续)

| 下标   | 数组名 | 指针   | 指针下标  | 四者逻辑关系                              |
|------|-----|------|-------|-------------------------------------|
| a[3] | a+3 | pa+3 | pa[3] | a[3] = *(a+3) = *(pa+3) = = pa[3]   |
| a[4] | a+4 | pa+4 | pa[4] | a[4] = *(a+4) = = *(pa+4) = = pa[4] |

#### 【例4.2】演示数组与指针关系的例子。

### 开始执行最后一个for语句时,p=&a[5]已经越界,所以使用减1操作。程序输出如下:

```
1 1 1 3 3 3 5 5 5 7 7 7 8 8 8 8 0x0012FF6C,0x0012FF6C 1 3 5 7 8 1 3 5 7 8 8 7 5 3 1
```

## 4.2 一维字符串数组与指针

字符串名是其首地址,使用指针可以方便地处理字符串。因为字符串用途广泛,所以单独用一节讲述。本节主要通过几个典型的例子说明其使用方法。

### 【例4.3】演示字符串数组和指针的例子。

### 程序运行输出结果如下:

ABCD, abcd a b c d abcd cd ABCD, ABCD

字符串顺序存放,字符串名是其首地址,p输出的不是地址,而是地址里的内容。

【例4.4】将字符串常量t的内容复制到数组s中,程序中t++是指针加1,即移动一个字符位置,从而实现一个字符一个字符的复制。

```
#include <stdio.h>
void main ()
{
    char s[32], *t=" You are welcome!";
    int i=0;
    while (s[i] = *t++)
        i++;
    printf ( "%s\n", s );
}
```

#### 程序输出结果如下:

```
You are welcome!
```

程序中的while语句首先将t指向的字符复制到数组的单元中,然后将指针t加1,最后根据复制的内容来决定条件的真假,若当复制的内容不为'\0',即值不为零,故为真,继续循环。只有当复制到结尾时,内容才为'\0',即值为零(假),结束循环。

#### 【例4.5】使用两个指针进行相减操作求串长度。

```
#include <stdio.h>
#include <malloc.h>
void main ()
{
    char *p,*s;
    s=( char *) malloc(128);
    scanf ( "%s", s );
    p=s;
    while (*p != '\0')
        p++;
    printf ( "%d\n", (p-s) );
}
```

#### 程序运行示范如下:

```
123456789ABCDEFG
16
```

程序没有对p的操作进行计数,而是在操作p之前先使用一个临时指针s保存p,最后计算p-s的值,即得到字符串的长度。

## 4.3 字符串常量

字符串常量存放在文字常量区。这一点要特别注意,不管是全局字符串常量,还是局部字符串常量,都由系统分配在文字常量区,这个区位于全局区。当然,局部字符串常量只是存储在文字常量区,并不像全局字符串那样可以共享。

由此可见,文字常量区仅用于保存字符串常量。字符串常量是不能改变的,由定义字符指针时同时定义常量字符串。如【例4.4】中的字符指针t。

字符串数组与字符串常量是不一样的,字符串常量不仅节约内存,而且使用方便。

#### 【例4.6】使用字符串常量的例子。

### 程序运行结果如下:

```
p輸出we are here! s輸出we are here!
st輸出we are here!
p輸出we are here! s輸出
p輸出we are here! s輸出are here!
```

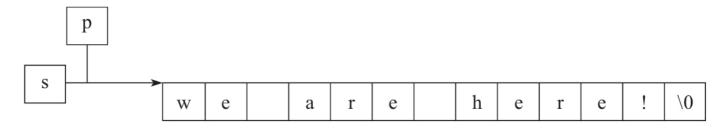
### 不要混淆指针与指针所指向的数据。尽管赋值语句

```
s= "we are here!";
```

使得打印语句输出s的值就是字符串"we are here! ",但却不能认为s的值就是这个字符串。实际上,s的值是一个指向由'w'、'e'、''、'a'、...、'! '和'\0'等13个字符组成的字符数组的起始元素(即第1个元素w)的指针。因此,执行语句

```
p=s;
```

则使得p和s的指向一样,即两个指针都指向内存中同一地址。这条语句并没有同时复制内存中的字符。图4-1给出它们的示意图,第1行的输出验证它们指向的数据相同。一定要记住:复制指针并不同时赋值指针所指向的数据。



还需要注意的是,常量字符串是不能修改的。ANSI C标准中也禁止对它进行修改。不过,某些C编译器允许修改,但VC不允许修改。

对字符串数组,只能一个一个地赋值。注意while语句中的s指针的指向是移动的,但p并没有动,所以两个指针的指向已经不同。也就是说,一个字符串常量可以供几个指针使用,并且互不影响。

赋值完成后,s指向字符串的结束标志,所以输出内容为空,这就是第3行的输出结果。将它往回移动,使它指向第2个单词的首字符a,使它输出"are here!"。当然,p的输出不变,仍然是完整的字符串内容,这是最后1行的输出结果。

由此可见,使用字符串常量能减少内存开支并提高程序效率。

## 4.4 指针数组

指针本身也是变量,所以将指向相同变量的指针变量集合在一起就构成一个指针数组。指针数组的每一个元素均为指针变量。语句

```
int *p[5];
```

说明p是一个数组,它由5个元素组成,每个元素均是指向整型变量的指针。

【例4.7】演示字符串指针数组的例子。

```
#include <stdio.h>
int main()
{
    int i = 0;
    char *p[]=("DOG","CAT","COMPUTER","MICROPROCESSOR","FISH" };
    for ( i=0; i<5; ++i )
        printf("%s ", *(p+i));
    printf("\n");
    return 0;
}</pre>
```

p为具有5个字符指针的数组,而地址里所含有的字符数量不一,这就大大减少了内存开销。如果使用字符串数组,则必须选取最长的字符串作为基准,会浪费内存。

程序运行输出结果如下:

DOG CAT COMPUTER MICROPROCESSOR FISH

### 4.5 配合使用一维数组与指针

从变量的角度看,C语言数组和指针与基本数据类型不同,它们都属于从基本数据类型构造出来的数据类型,但又分属于不同的数据类型,从这一点看来,它们两者之间并不存在某种关系。如果换一个角度看,C语言的数组名就是这个数组的起始地址(数组名就是指针常量),而指针变量用来存储地址,因为从使用的角度看,它们都涉及地址,所以在使用时,它们又有着密切的关系。其实,任何能由数组下标完成的操作,也能用指针实现。所以说,指针和数组是离不开的一对"难兄难弟",是拴在一根绳上的"蚂蚱"。

在使用数组上, 正确灵活地使用指针, 能起到事半功倍的效果。

## 4.6 动态内存分配与非数组的指针

数组、指针和动态内存也是密切相关的,容易出现的错误仍然是边界和初始化问题,所以要特别留神。

【例4.19】下面程序将数组t和s中的内容赋给指针变量p,但输出结果并没有包括s的全部内容。找出错误之处并改正之。

```
#include <stdio.h>
#include <string.h>
int main ()
{
        int i=0,j=0;
        char t(]="abcdefghij",s[]="klmnopqrstuvwxyz",*p;
        p=t;
        i=strlen(t);
        while ((p[i+j] = s[j]) !='\0')
        j+t;
        printf("%s\n",p);
        return 0;
}
```

使用数组t初始化指针的想法是希望利用超出t的存储空间来存储s,这是危险的做法。越界之后,并不能保证有连续的有效存储空间用以存储字符串s。

可以另外定义一个大于s和t总长度的字符数组。例如

```
char st[30];
p=st;
```

然后使用如下两个循环完成赋值:

```
p=(char*)malloc ( strlen(t)+strlen(s)+1)
```

strlen函数计算的是实际字符串长度,所以要增加一个结束位。实际使用时,需要判别申请是否成功。这块内存虽然是非数组的指针,但却可以像数组那样使用下标。程序中演示了两种反序输出的方法,特别是演示下标为负值的使用方法,以便更好地理解动态内存的特点及指针的灵活使用方法。

```
// 先数相称
#include <stdio.h>
#include <stdiin.h>
#include <stdiin.h

#include <stdiin.h
```

#### 程序输出结果如下:

abcdefghijklmnopqrstuvwxyz zyxwvutsrqponmlkjihgfedcba zyxwvutsrqponmlkjihgfedcba

释放内存,必须保证指针指向申请的动态内存的开始位置,否则会出错,所以程序中执行"p=p-25;"。申请内存时多申请了6个,是为了保证free函数可靠执行。

数值数组的使用方法与此类似,不再赘述。

申请的动态内存虽然是非数组的指针,但可以像使用数组指针那样使用这个指针。

【例4.20】下面程序将数组t中的内容存入到动态分配的内存中。找出错误之处并改正。

```
#include <stdio.h>
#include <string.h>
#include <stdiib.h>
void main ()

{
    int i=0;
    char t[]="abcde";
    char *p;
    if ( (p=malloc ( strlen(t) ) ) == NULL ) {
        printf ( "内存分配错误!\n" );
        exit(1);
    }
    while (( p[i] = t[i]) !='\0' )
        i++;
    printf("%s\n",p);
}
```

这个程序可以编译并正确运行,但如果从语法上讲,可以找出几个问题。首先指针初始化不对,需要强迫转换为char指针类型。另外申请的内存不够装入字符串。因为库函数strlen计算出来的是实际字符串的长度,但存入它们时,还需要增加一个标志位。即正确的形式应该为:

```
if ( (p=(char *)malloc ( strlen(t)+1 ) ) == NULL )
```

但是,为什么能正确运行呢? 这就是指针的特点了。虽然申请的内存不够,但却能正确运行。如果使用

```
p=(char *)malloc (1);
```

语句,也能正确运行。因为毕竟使指针p指向一个有效的地址,也就是对指针正确地执行了初始化。至于分配的地址不够,并不限制指针的移动,这时指针可以去占用"他人"的地址。这一点务必引起注意,如果它"跑"到别人要用到的区域,就会起到破坏作用,甚至造成系统崩溃。

申请内存时,要注意判别是否申请成功。在使用完动态内存之后,应该使用语句

free (p);

释放内存,这条语句放在程序结束之前即可。因为是在复制了结束位之后满足结束循环条件,所以就不能再写入结束标志了。

```
// 正确程序
#include <stdio.h>
#include <stdib.h>
#include <stdib.h>
void main ()
{
    int i=0;
        char t[]="abcde";
        char 'p;
    if ( (p=(char *)malloc ( strlen(t)+1 ) ) == NULL ) {
            printf ( "内存分配错误!\n" );
            exit(1);
        }
    while ((p[i] = t[i]) !='\0')
        i++;
        printf("%s\n",p);
        free(p);
}
```

## 4.7 二维数组与指针

虽然本节只限于讨论二维数组,但这些方法也可以推广到三维数组。

## 4.8 综合设计实例

本节给出一个综合设计实例。

假设给定班级各科考试平均成绩的原始资料如下:

数学:75物理:80外语:83政治:85体育:86人数:30

要求统计出全班学期总平均成绩以及得分最低的科目和该科目的成绩。要求的输出结果如下:

原始信息如下: 数学:75物理:80外语:83政治:85体育:86人数:30平均成绩:0最低分数科目的成绩:0最低分数的科目:全班各科平均成绩加下: 数学:75物理:80外语:83政治:85体育:86统计结果如下: 人数:30平均成绩:81最低:

## 第5章 函数基础知识

C语言的模块设计离不开函数,设计函数离不开变量。掌握函数设计和调用的正确方法是程序设计的基本功,而正确设计和使用变量,既能提高函数的效率,又能保证程序的正确性并提高程序设计的效率。本章主要介绍函数的基本概念及其基础知识、C程序的典型结构和变量的作用域。

### 5.1 函数

C语言是结构化程序设计语言,它的程序设计特点就是函数设计。一般来讲,C的源程序是由若干个函数组成的。运行时,程序从主函数main开始执行,到main的终止行结束。其他函数由main或别的 函数或自身调用后组成可执行程序。

所谓函数,就是模块的基本单位,是对所处理问题的一种抽象。例如,将求绝对值的功能抽象为abs(参数),就有abs(25)=25和abs(-25)=25。称abs为求一个数的绝对值函数,而称25和-25为函数abs的参数。

将一切逻辑功能完全独立或相对独立的程序部分都设计成函数,并让每一个函数只完成一个功能。这样,一个函数就是一个程序模块,程序的各个部分除了必要的信息交流之外,互不影响。相互隔离的程序设计方法就是模块化程序设计方法。可以说,结构化思想改进了函数的设计原则,而一个函数设计的好坏又体现在结构化思想的运用上。由此可见,C程序设计的质量决定于如何设计结构化好的函数。

【例5.1】编写一个具有两个参数的函数max,比较这两个参数的大小,并将大者作为函数的返回值。

下面的主程序调用max函数,x的值就等于a、b中的最大值。为了方便讲解,将用注释的形式给出每行的行号。

## 5.2 C程序的典型结构

可以按编制C程序使用程序文件(包括头文件和源程序文件)的数量将其分为单文件结构和多文件结构,而且单文件结构没有自己定义的头文件。多文件结构又可以按编制C程序源文件和头文件的多少分为两类,一类是只有一个源程序文件和头文件,另一类有多个源程序文件和一个或多个头文件。当然,前者只是后者的特例而已。

### 5.3 变量的作用域

内存地址由系统分配,不同机器为变量分配的地址大小虽然可以不一样,但都必须给它分配地址。变量的存储类型在变量声明中指定。变量声明的一般形式为

C定义的常用存储类型有4种,即auto、extern、static、register,分别称为自动型、外部型、静态型和寄存器型。可以在声明时为变量赋初值。例如,下面的例子

```
auto int a;
static float b, c;
extern double x;
register int i=0;
static int size=60;
```

volatile影响编译器编译的结果,这里暂不讨论volatile的使用方法。

应该养成在声明时就为变量赋初值的习惯,但在某些特殊场合,则只能声明,如头文件中对外部变量的声明。

C程序的源文件可以分散在几个文件之中,事先编好的程序可以从库中装入,这就涉及变量的作用域范围。如何说明变量,才能在编译期间被适当地编译?才能使程序装入时所有的程序段都能被适当 地连接起来?下面将就这些问题分别加以说明。

变量作用域根据其起作用的范围分为一个块结构、一个函数、一个源程序文件及包括多个文件(多个C文件及头文件)的完整程序等层次。

## 5.4 变量的存储地址分配

可以将存储区分为代码区、文字常量区、全局区(静态区)、堆和栈。代码区用来存放程序的二进制代码,由系统负责。文字常量区存放字符串常量。这一点要特别注意,不管是全局字符串常量,还是局部字符串常量,都由系统分配在文字常量区,这个区位于全局区,当然,局部字符串常量只是存储在文字常量区,并不像全局字符串那样可以共享。

## 5.5 main函数原型及命令行参数

main函数是一个特殊的函数。首先,每个C程序都必须有一个名为main的函数,程序从这里开始执行。C语言可以对没有main函数的程序部分进行编译,但若其他模块也没有包含main函数,则连接 失败。

main函数另一个独特的属性是,它有两种正式的原型且经常会使用一些其他形式。两种标准原型如下。

```
int main(void);
int main(int argc, char* argv[]);
```

整型返回值的作用是向系统返回一个状态码,这在复杂应用程序的过程间通信时会用到,不过它对单一的程序是没有什么意义的。在这种状态下,可以采用一些非标准的形式,如 "void main (void)"。无返回值的程序能够正常执行,是因为多数系统都不依赖于返回的状态码,简单程序的状态码没有意义。

在程序开始执行时,可将命令行传送给程序。当调用主程序main时,让它带有两个参数。第1个参数习惯上叫作argc,是表示被调用程序所带命令行参数的数目;第2个参数argv是指针数组,其中每个元素是指向包含命令行参数的字符串的指针,即每个指针对应一个字符串,而第1个指针指向的通常是命令名字符串。在实际应用中,命令行参数是很有用的。

【例5.24】实现echo命令的例子。

系统有一个命令用来实现参数回响,将它的命令行参数回响在一个单行上,并用空格将它们分隔。这个程序不能在Windows下运行,要转到DOS窗口下执行。例如:

```
C:\> echo hello world <CR>
```

### 程序在DOS窗口输出如下信息:

hello world

在执行上述命令时,argc参数的值为3,表示命令行参数的数量。argv[0]含有可执行程序的名字 "echo" ,argv[1]是 "hello" ,argv[2]是 "world" 。

下面的程序将这些参数分行输出。假如可执行文件名为test,输入如下命令行。

C:\> test echo hello world

这里的文件名为test, echo只是参数, 应得到的输出如下:

```
argc=4
argv[0]=test
argv[1]=echo
argv[2]=hello
argv[3]=world
```

也就是说,main函数可以获得命令行参数的个数及参数的内容。test.c的源程序如下。



图5-9 选择命令提示符示意图



图5-10 命令提示符窗口示意图

假设这个文件夹的路径为f: \test\Debug,使用命令行命令进入Debug文件夹, "■"表示光标的位置,则有: F:\test\Debug> 在光标处输入如下命令并按回车键: F:\test\Debug> test echo hello world 程序在窗口内输出如下结果: argc=4 argv[0]=test argv[1]=echo argv[2]=hello argv[3]=world

## 第6章 函数设计

一般而言,设计一个函数并不难,难的是设计合理的函数类型及参数传递方式,并能正确地使用它们。函数类型决定函数返回值的类型,而参数传递既要考虑哪些可以作为函数参数,也要考虑是否允许资章议业表彰。要综合考虑函数返回值和表数,在决到目的的前提下,尽量简化函数设计。推莲使用const限定符修饰那些不允许改变的函数表数。

## 6.1 函数设计的一般原则

C语言是结构化程序语言,它的程序设计特点是函数设计、函数返回值和参数传递都涉及系统如何为变量分配地址和作用域,所以要注意结合第5章5.3节和5.4节的知识。

### 6.2 函数的返回值

函数类型决定返回值的类型,两者必须保持一致。如前所述,函数的类型只是要求返回值的类型与其一致,但提供的返回值是否有效及程序如何正确地得到返回值,才是编程要实现的目标。本节将主 要围绕这个问题深入讨论函数的返回值。

数组不能定义函数,所以函数也不能返回数组。函数可以返回指针和结构。实际上,函数返回值与函数参数也是相关的,所以讨论时也不可能绝对分开。

一定要注意函数返回值和参数的区别。可以通过函数返回值改变调用函数中参数的值,也可以通过参数传递的方式改变参数的值。虽然推荐将不需要返回值的函数定义为void类型以简化设计,但也不 是说不允许符它设计为非void类型,即允许被调函数不使用函数的返回值。总之,设计时应尽量遵循简单合理的原则。

### 6.3 函数参数的传递方式

C语言函数参数的传递方式只有传值方式一种。传值又分为传数值和传地址值,参数传递时可以使用const限定函数参数,以防止被调函数修改。

## 6.4 函数指针

在C语言中,指针函数与函数指针是不同的,一个特别容易引起混淆而又相当有用的特征是函数指针。前面讲过,可以用指针变量指向整型变量、字符变量及字符串、浮点变量和数组。其实,指针变量也可以指向一个函数。因为尽管函数本身不是一个变量,但它在内存中仍然有其物理地址。在编译过程中,原代码被转换成目标代码,函数的入口地址也同时确立,所以就能够将函数的入口地址赋给指针变量。程序调用函数,也就是机器语言的"CALL"指向了这个入口点。因为指向函数的指针实际上包含了函数的入口点的内存地址,所以赋给指针变量的地址就是函数的入口地址,从而该指针就可以用来代替函数名。这一特性也使得函数可以作为实参传递给其他函数。由此可见,可以定义一个指向函数的指针变量,它可以被处理,如传递给函数,或放置在数组中等等。

通过函数指针变量可以完成对函数的调用。其原理是通过把一个函数赋给一个函数指针变量,然后又通过该函数指针变量来完成对函数的引用。因为函数的指针是一个比较高深的概念,所以要通过多 用来加深理解。

【例6.32】 演示使用函数指针输出多项式x3-4x+6和x2-3x在区间 [-1, +1] 增长步长为0.1时的所有结果。

给函数指针变量赋值时,只需要给出函数名而不必给出参数。因为语句

p=f1;

是将函数入口地址赋给指针变量p,而不涉及实参与形参结合的问题,如果写成

p=f1(x);

### 的形式则是错误的。

正如数组名代表的是数组起始地址一样,这里的函数名代表函数的入口地址。p是指向函数f1的指针变量,它和f1都指向函数的开头,调用p就是调用f1。与过去所讲的变量的重要区别是:它只能指向函数的入口处,而不能指向函数中间的具体指令。因此,\* (p+1) 、p+n、p--及p++等运算对它都是没有意义的。

"double (\*p) ();"语句仅仅说明定义的p是一个指向函数的指针变量,此函数带回实型的返回值。但它并不是固定指向哪一个函数的,而只是表示定义了这样一个类型的变量,专门用来存放函数的入口地址。在程序中把哪一个函数的地址赋给它,它就指向哪一个函数。这个程序使用循环语句分别计算两个函数。

下面给出部分运行结果。

```
第1个方程:
-1.00000 9.00000
-0.90000 8.871000·····
1.00000 3.00000第2个方程:
-1.00000 4.00000····· ····
-0.20000 0.640000
-0.100000 0.310000
```

函数指针可以作为函数参数,下面使用函数指针的方法,求函数10x2-9x+2在区间 [0, 1] 内x以0.01的增量变化的最小值。

【例6.33】 使用表达式调用函数指针的例子。

#### 运行结果如下:

最小值是: - 0.025

#### 【例6.34】 使用函数参数调用函数指针的例子。

### 6.5 理解函数声明

只有避免语法"陷阱",才能读懂函数。为了更好地编程,还要避免词法"陷阱"。为此,先介绍一下词法分析中的"贪心法"。

### 6.6 函数设计举例

本节将列举一些简单的例子以加深读者对函数设计的理解。

## 第7章 函数设计实例

进行函数设计时, 首先要清楚函数的类型和返回值, 然后根据类型和返回值确定合适的参数。换个角度思考, 函数是实现一种特定算法, 所以本章也简要介绍一下算法的概念, 以便使读者自己设计的 函数能满足算法要求。 一般来讲,除了很简单的程序之外,计算机程序都是通过主程序调用相应的函数实现的。为了设计好程序,就要熟悉计算机"解题"的方法,也就是计算机的"思维方式";这样才能根据计算机的特点设计出合适的函数以实现相应的算法,然后编制主程序正确地调用这些函数完成预定功能。所以本章先讨论函数设计的一般原则,然后结合典型算法,用实例说明设计的具体方法,以便进一步开阔读者的眼界。

## 7.1 函数的类型和返回值

调用一个函数的目的有两个,一个是直接在被调函数里输出结果,另一个是从它那里得到供程序继续使用的中间值。

### 7.2 正确选择函数参数

选定了函数的类型和返回值后,接着就要为函数设计正确的参数。设计参数时,尤其要注意需要通过键盘进行人机交互的参数的设计。

7.1.2节已经结合实例说明了这个问题,本节主要讨论结构作为函数参数的问题。

### 7.3 算法基本概念

本节简要介绍算法的基本概念。

从直观上讲,使用计算机来解决一个实际问题,就是对应给定的一组输入,求得一组相应的输出。要解决这个问题,就要设计能解决问题的具体步骤和方法,人们把对问题的解题方案的精确而完整的 描述称为算法。程序可以作为算法的一种描述,因为在编写程序时要考虑计算机系统运行环境的限制,所以程序通常要考虑很多与方法和分析无关的细节。程序是利用计算机程序设计语言实现的算法,但 算法并不等于程序。

#### 1.算法的特征

因为算法实际上是一种抽象的解题方法,所以说现代计算机是面向算法的自动机。算法具有动态性,因此算法的行为非常重要,它具有如下特征。

- 1) 确定性: 算法中的每一个步骤都必须是确切定义的,不允许有模棱两可的解释,也不允许二义性。
- 2) 有限性: 一个算法必须在执行有限个步骤之后终止。
- 3)可行性:指能用现有性能的计算机在有实际意义的时间内解决问题。可行性是对确定性和有限性进一步的精确化。首先,算法中的每一步骤必须是能实现的。例如不允许出现分母为0,实数范围内不能求一个负数的平方根。另外,算法执行的结果要能达到预期的目的。对有限性来说,不仅要求执行步骤有限,而且要求能在有意义的时间内用现有的水平的计算机实现。以国际象棋为例,它可能出现的棋局为10<sup>120</sup>种,可以使用计算机模拟全部棋局,但用亿次机,以每秒处理3×10<sup>6</sup>个棋局的速度计算,也需要10<sup>106</sup>年,虽然算法是可以实现的,但却没有意义。
  - 4) 输入: 具有0个或多个输入的外界量, 它们是算法开始前赋给算法的初始量。
  - 5)输出:至少产生一个输出,它们是同输入有某种特定关系的量。

### 2.算法效率和算法分析

设计算法时应注意实现如下两个目标。

- 1) 设计一个正确的、容易理解的、容易编码和调试的算法。
- 2) 设计一个能有效利用计算机资源和求解效率高的算法,即空间复杂度与时间复杂度低的算法。

以上这两个目标有时互相冲突,需要综合考虑。通过对算法进行分析,可以清楚地看到在解决同一个问题时不同算法在效率上或存储量需求上的差异,所以可以通过算法分析来度量算法的优劣。

在分析算法利用计算机资源的效率时,主要应从执行算法所耗费的时间和使用存储器的多少两方面来分析。

### 3.算法的重要性

无论是制造芯片的电子线路设计软件、多媒体图像压缩技术,还是"深蓝"计算机在与连续12年保有国际象棋世界冠军头衔的卡斯帕罗夫的对弈中的获胜,都是靠巧妙的算法。

算法设计是人类智慧的结晶。计算机科学中的知识创新,算法的创新占有重要的地位。从某种意义上说,一种算法的创新意义不亚于一种新机型的发明。

由此可见,算法不是编程技巧。不能把"编程快手和能手"作为衡量和掌握计算机专业知识技能的唯一标准。这就是说,在以后的学习中,要重视算法的学习。在编程中要注意选择更有效的算法。

### 7.4 使用库函数

设计函数时,可以直接利用函数库的资源,即调用库函数。另外,所有实用的C程序都必须使用库函数,所以必须懂得如何正确引用库函数。

引用库函数要特别注意如下几个问题。

### 1.引用的库函数与头文件不匹配

引用库函数时的首要条件是使用系统头文件。因为所有库函数都提供了一个头文件,在该头文件中,已经精确地描述了对自变量类型与返回类型的说明,为了保证能够得到正确的结果,不仅需要使用系统头文件,还必须保证库函数与头文件的引用是相互匹配的。例如,如果要引用求绝对值的库函数abs,它的头部文件在math.h里,正确的包含应为

#include <math.h>

#### 2.与库函数的参数类型不匹配

如果程序包含的头文件是对的,下一步就要正确调用库函数。在使用库函数时,一定要弄清楚怎样才是正确的使用,怎样才能达到预期的目的。例如要使用库函数sin,可以查库函数手册中的函数原型 声明。函数sin的原型在math.h中,函数声明为

double sin ( double arg )

由此可见,该程序调用的参数arg是弧度,返回值是double。

大部分库函数都很简单,人们都能正确地使用它们。常常发生问题的原因是对库函数里定义的数据类型没有掌握,或者在引用时,自己另外对它们进行了定义,而这些定义又与原定义不符。

#### 3.对库函数的作用理解不正确

只有正确理解库函数的原型,才能正确使用它。例如, strlen的函数原型为

size\_t strlen( const char \*string );

size\_t是unsigned integer,即strlen函数返回字串的长度(字符串的个数),这个长度不包含字符串的结束标志\0′,也就不是存储字符串的长度。假设对于字符串string,如果想使用下面的语句

printf("字符串%s的长度为%d。\n",string,strlen(string)+1);

来输出它的长度,就是错误的。虽然strlen函数返回字串的长度(字符串的个数),但这个长度不包含字符串的结束标志\0′,也就不是存储字符串的长度。所以正确的方式是将输出语句中的"+1"去掉。即改为

printf("字符串%s的长度为%d。\n",string,strlen(string));

## 7.5 设计实例

由于篇幅限制。这些实例主要局限于函数的实现。而不涉及讨多的细节。

## 第8章 多文件中的函数设计

多文件编程的目的是保证程序的结构化设计质量。编制多文件涉及头文件和预处理问题。本章将结合具体实例详细介绍头文件的编制、多个C语言文件及工程文件的编制等方法,以提高读者多文件编程 的能力。

### 8.1 C语言预处理器

C语言预处理器是C编译程序的一部分,它负责分析处理几种特殊的语句,这些语句被称为预处理语句。顾名思义,预处理器对这些语句的分析处理是在编译程序的其他部分之前进行的。预处理语句有 3种,分别是宏定义、文件包含和条件编译。为了与一般C程序语句相区别,所有预处理语句都以位于行首的符号"#"开始。

C语言预处理器和有关语句能够帮助程序员编写易读、易改、易移植及易调试的程序,对于模块化程序设计也提供了很大的帮助。

## 8.2 模块化程序设计基础

本节仅简要介绍最基础的知识。

### 8.3 使用两个文件的设计实例

本节的程序含有一个头文件和一个C源程序文件。

## 8.4 使用3个文件的设计实例

一元多项式的运算包括加、减和乘法运算,而多项式的减法和乘法都可以用加法来实现,因此,这里只需要实现一个加法运算。

## 8.5 使用条件编译的多文件设计实例

本节设计一个简单的通讯录,记录信息仅仅为成员代号和电话。要求建立链表,使用链表存储结构信息,增加链表结点及删除链表结点等基本操作,并使用条件编译和多文件编程。

## 第9章 多文件综合设计实例

本章给出两个典型的多文件编程实例,以进一步说明如何设计头文件、划分多文件及设计函数。这两个程序一个使用链表,另一个使用数组,所以具有代表性。

## 9.1 使用链表设计一个小型通讯录程序

本节的程序功能相对简单,重点是练习设计合适的文件。文件之间的变量传递方式以及函数的类型、参数及返回值。

## 9.2 使用数组设计一个实用的小型学生成绩管理程序

本节使用数组设计一个实用的小型学生成绩管理程序,它有查询、检索和排序等功能,并且能够对指定文件操作,也可将多个文件组成一个文件。

## 第10章 设计游戏程序实例

本章讲解设计游戏实例的目的有两个:一是通过趣味程序进一步加强函数设计的练习;二是进一步认识一个应用程序的函数之间及文件之间的各种关系,加深对一个完整工程项目的理解,从而提高读者的多文件编程能力。

## 10.1 剪刀、石头、布

剪刀、石头、布是个古老的两人对弈游戏。这里是实现人机对弈,即电脑出拳与玩家出拳相比较,看谁是赢家。

输赢规则:剪刀能剪布,石头能打剪刀,布又能包石头。

### 10.2 洣宫

编程时一般以一个m×m阶方阵表示迷宫,0和1分别表示迷宫中的通路和障碍。

这里简化设计要求,限定为8×8方阵,要求设计一个程序,随机产生一个8×8方阵的迷宫,使用顺序栈探索这个迷宫,判断是否有一条从第1行入口到第8行出口的通道,如果有则给出有的结论并给出 探路的轨迹,没有则给出没通路的结论。

## 10.3 空战

这个程序很简单,用按键A或D控制我方飞机左右移动,用W键发射子弹。敌机不断从对面飞来,子弹击中敌机得分,我方飞机被敌方飞机碰撞则我方飞机坠毁。

我方飞机只能左右移动,敌方飞机只能上下移动。可以设置飞机的密度和飞行的速度以改变游戏难度。

## 10.4 贪吃蛇

用键盘控制蛇的方向,寻找"吃的东西",每"吃"一口就能得到一定的积分,而且"蛇"的身子会越"吃"越长,规则是不能碰墙,不能"咬"到自己的身体,达到一定的分数就能过关,然后继续玩下一关。

这里给出一个范例,整个程序编写在一个文件里,而且都在主程序里,程序可以运行。提供这个范例的目的是让读者自己优化,为它编写头文件并改造主程序,将其分解为相应的函数,使其结构性更好并具有易读性。

注意,"蛇"每隔单位时间向当前方向前进一步,然后刷新(也就是把画面重新输出一遍),在随机位置产生"食物","蛇"的"尾巴"经过"食物"(即已经消化)后加长一段。按控制方向的键时改变记录当前方向变量的值。加入判断死亡的机制。

程序中给出了分支语句的层次提醒以方便分析。

## 10.5 停车场

本程序模拟一个管理停车场车位和收费的软件。程序很简单,不再详细讲解。

## 10.6 画矩形

Trubo C提供的头文件"graphics.h"包含了系统的图形函数,为编写游戏界面提供了方便。

Microsoft C提供了绘制Windows图形的功能,可以用于编制Windows程序。使用Windows的API函数可以很方便地编程。

用C可以编制Windows程序,也就是编制面向对象的程序。

## 10.7 俄罗斯方块

俄罗斯方块的发明人是俄罗斯人阿列克谢·帕基特诺夫(АлексейПажитнов英文Alexey Pazhitnov)。俄罗斯方块原名是俄语Тетрис(英语是Tetris),这个名字来源于希腊语tetra,意思是"四",而游戏的作者最喜欢网球(tennis)。于是,他把两个词tetra和tennis合而为一,将此游戏命名为Tetris,这也就是俄罗斯方块名字的由来。如图10-12所示是典型的游戏界面之一。



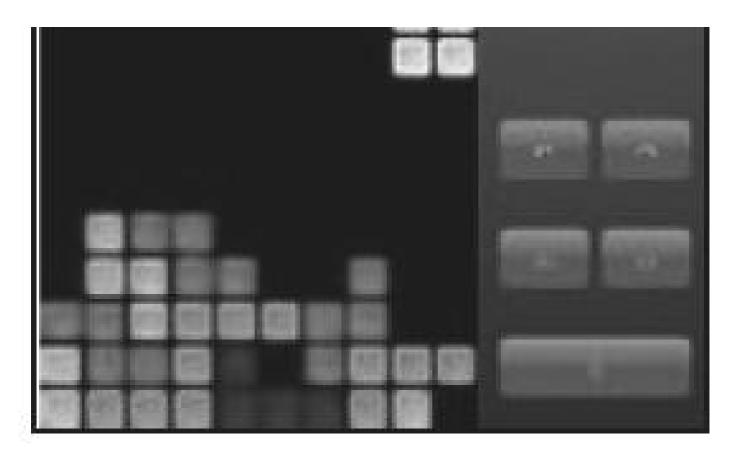


图10-12 典型的俄罗斯方块游戏界面图

由小正方形组成的不同形状的板块陆续从屏幕上方落下来,玩家通过调整板块的位置和方向,使它们在屏幕底部拼出完整的一条或几条横条。这些完整的横条会随即消失,给新落下来的板块腾出空间,与此同时,玩家得到分数奖励。没有被消除掉的方块则会不断堆积起来,一旦堆到屏幕顶端,玩家便告输,游戏结束。

# 10.8 用C语言编写Windows下的贪吃蛇游戏

附录 7位ASCII码表

| $d_3d_2d_1d_0$ | $d_6d_5d_4$ |     |     |     |     |     |     |     |  |
|----------------|-------------|-----|-----|-----|-----|-----|-----|-----|--|
| u3u2u1u0       | 000         | 001 | 010 | 011 | 100 | 101 | 110 | 111 |  |
| 0000           | NUL         | DLE | SP  | 0   | @   | P   | ,   | p   |  |
| 0001           | SOH         | DC1 | !   | 1   | A   | Q   | a   | q   |  |
| 0010           | STX         | DC2 | "   | 2   | В   | R   | b   | r   |  |
| 0011           | ETX         | DC3 | #   | 3   | С   | S   | С   | S   |  |
| 0100           | EOT         | DC4 | \$  | 4   | D   | T   | d   | t   |  |
| 0101           | ENQ         | NAK | %   | 5   | Е   | U   | e   | u   |  |
| 0110           | ACK         | SYN | &   | 6   | F   | V   | f   | v   |  |
| 0111           | BEL         | ETB | 1   | 7   | G   | W   | g   | W   |  |
| 1000           | BS          | CAN | (   | 8   | Н   | X   | h   | x   |  |
| 1001           | HT          | EM  | )   | 9   | I   | Y   | i   | у   |  |
| 1010           | LF          | SUB | *   | ;   | J   | Z   | j   | Z   |  |
| 1011           | VT          | ESC | +   | ;   | K   | [   | k   | {   |  |
| 1100           | FF          | PS  | ,   | >   | L   | \   | 1   |     |  |
| 1101           | CR          | GS  | -   | =   | M   | ]   | m   | }   |  |
| 1110           | SO          | RS  |     | <   | N   | ^   | n   | ~   |  |
| 1111           | SI          | US  | /   | ?   | О   | _   | О   | DEL |  |

## 参考文献

- [1] 刘振安. C语言程序设计 [M] .北京: 机械工业出版社, 2007.
- [2] 刘振安,等. C程序设计课程设计 [M] .3版.北京: 机械工业出版社, 2016.
- [3] 谭浩强. C程序设计 [M] .北京:清华大学出版社,2001.
- [4] 刘燕君,等. C程序设计实践教程 [M] .北京: 机械工业出版社, 2009.
- [5] 刘振安,等. C语言程序设计教程 [M] .北京: 机械工业出版社, 2008.
- [6] 刘振安. C语言程序设计 [M] .2版.北京:清华大学出版社, 2008.
- [7] 裘宗燕. 从问题到程序:程序设计与C语言引论[M].北京:机械工业出版社,2005.
- [8] Eric Brechner.完美代码 [M] .徐旭铭,译.北京: 机械工业出版社, 2009.
- [9] Eric Brechner.代码之道 [M] .陆其明,译.北京: 机械工业出版社,2010.
- [10] Andy Oram, Greg Wilson.代码之美 [M] .BC Group,译.北京:机械工业出版社,2010.
- [11] Mario Hewardt, Daniel Pravat.Windows高级调试 [M] .聂雪军,等译.北京:机械工业出版社,2010.
- [12] Eric S Roberts.C程序设计的抽象思维(英文版) [M] .北京: 机械工业出版社, 2004.
- [13] 柯奈汉,派克·程序设计实践 [M] .裘宗燕,译.北京:机械工业出版社,2000.
- [14] Andrew Koenig.C陷阱与缺陷 [M] .高巍, 译.北京: 人民邮电出版社, 2002.
- [15] 刘振安,苏仕华,周淞梅.C程序设计与错误分析 [M] .合肥:中国科学技术大学出版社,1995.
- [16] 刘燕君, 等. C语言程序设计实践教程 [M] .北京: 机械工业出版社, 2012.
- [17] 刘振安,等. C语言程序设计教程 [M] .北京:邮电大学出版社,2012.
- [18] Jon Bentley.编程珠玑 [M].钱丽艳, 等译.北京: 人民邮电出版社, 2008.
- [19] 刘振安. Windows实用编程指南 [M] .北京: 人民邮电出版社, 1995.
- [20] 刘振安. Windows编程 [M] .北京: 人民邮电出版社, 1995.

- [21] 刘振安. Windows编程简明教程 [M].合肥:安徽科学技术出版社, 1995.
- [22] 刘振安. Windows可视化程序设计 [M] .北京: 机械工业出版社, 2007.
- [23] 刘振安, 刘燕君. C语言解惑 [M] .北京: 机械工业出版社, 2014.