第 3 讲　C 语言函数与机器人

巡航控制

学习情境

通过对单片机编程可以使机器人完成各种巡航动作。本讲使用 C51 单片机和 C 语言来重新实现这些功能，同时详细了解 C 语言函数的定义和使用方法。前面已经提到，函数是 C 语言的核心概念和方法。

本讲所要完成的主要任务如下。

（1）对单片机编程使机器人做基本巡航动作：向前、向后、左转、右转和原地旋转。

（2）编写程序使机器人由突然启动或停止变为逐步加速或减速运动。

（3）写一些执行基本巡航动作的函数，每一个函数都能够被多次调用。

（4）将复杂巡航运动记录在数组中，编写程序执行这些巡航运动。

任务 1　基本巡航动作

图 3-1 定义了机器人的前、后、左、右 4 个方向：当机器人向前走时，它将走向本页纸的右边；当向后走时，会走向纸的左边；向左转会使其向纸的顶端移动；向右转它会朝着本页纸的底端移动。

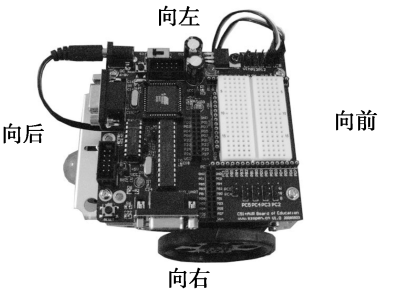

图 3-1　机器人及其前进方向的定义

向前巡航

按照图 3-1 前进方向的定义，机器人向前走时，从机器人的左边看，它向前走时轮子是逆时针旋转的；从右边看另一个轮子则是顺时针旋转的。

回忆一下第 2 讲的内容，发给单片机控制引脚的高电平持续时间决定了伺服电机旋转的速度和方向。for 循环的参数控制了发送给电机的脉冲数量。由于每个脉冲的时间是相同的，因而 for 循环的参数也控制了伺服电机运行的时间。下面是使机器人向前走 3s 的程序实例。

例程：RobotForwardThreeSeconds.c

- 确保控制器和伺服电机都已接通电源；
- 输入、保存、编译、下载并运行程序 RobotForwardThreeSeconds.c。

```
#include<BoeBot.h>
#include<uart.h>
int main(void)
{
    int counter;
    uart_Init();
    printf("Program Running!\n");

    for(counter=0;counter<130;counter++)//运行 3s
    {
        P1_1=1;
        delay_nus(1700);
        P1_1=0;

        P1_0=1;
        delay_nus(1300);
        P1_0=0;

        delay_nms(20);
    }
    while(1);
}
```

RobotForwardThreeSeconds.c 是如何工作的

理解该例程的运行你应该没什么问题：for 循环体中前三行语句使左侧电机逆时针旋转，接着的三行语句使右侧电机顺时针旋转。因此两个轮子转向机器人的前端，使机器人向前运动。整个 for 循环执行 130 次大约需要 3s，从而机器人也向前运动 3s。

关于例程调试的一点说明

例程中使用 printf 函数是为了起提示作用。若你觉得串口线影响了机器人的运动，可以不用此函数。还有一个进行调试的方法：让机器人的前端悬空，让伺服电机空转。这样调试起来就方便了，机器人不会到处乱跑。后面的例程调试也是这样。

该你了——调节距离和速度

- 将 for 循环的循环次数调到 65，可以使机器人运行时间减少到刚才的一半，运行距离也是一半；
- 以新的文件名保存程序 RobotForwardThreeSeconds.c；
- 运行程序来验证运行的时间和距离是否为刚才的一半；
- 将 for 循环的循环次数调到 260，重复这些步骤。

delay_nus 函数的参数 *n* 为 1700 和 1300，都使电机接近它们的最大速度旋转。把每个 delay_nus 函数的参数 *n* 设定得更接近让电机保持停止的值——1500，可以使机器人减速。

更改程序中相应的代码片段如下：

```
P1_1=1;
delay_nus(1560);
P1_1=0;
P1_0=1;
delay_nus(1440);
P1_0=0;
delay_nms(20);
```

运行程序，验证一下机器人运行速度是否减慢。

向后走、原地转弯和绕轴旋转

将 delay_nus 函数的参数 *n* 以不同的值组合就可以使机器人以其他的方式运行。例如，下面的程序片段可以使机器人向后走。

```
P1_1=1;
delay_nus(1300);
P1_1=0;
P1_0=1;
delay_nus(1700);
P1_0=0;
delay_nms(20);
```

下面的程序片段可以使机器人原地左转。

```
P1_1=1;
delay_nus(1300);
```

```
P1_1=0;
P1_0=1;
delay_nus(1300);
P1_0=0;
delay_nms(20);
```

下面的程序可以使机器人原地右转。

```
P1_1=1;
delay_nus(1700);
P1_1=0;
P1_0=1;
delay_nus(1700);
P1_0=0;
delay_nms(20);
```

你可以把上述命令组合到一个程序中让机器人向前走、左转、右转及向后走。

例程：ForwardLeftRightBackward.c

- 输入、保存并运行程序 ForwardLeftRightBackward.c；

```
#include<BoeBot.h>
#include<uart.h>
int main(void)
{
    int counter;
    uart_Init();
    printf("Program Running!\n");

    for(counter=1;counter<=65;counter++)//向前
    {
        P1_1=1;
        delay_nus(1700);
        P1_1=0;

        P1_0=1;
        delay_nus(1300);
        P1_0=0;

        delay_nms(20);
```

```
}

for(counter=1;counter<=26;counter++)          //向左转
{
    P1_1=1;
    delay_nus(1300);
    P1_1=0;

    P1_0=1;
    delay_nus(1300);
    P1_0=0;

    delay_nms(20);
}

for(counter=1;counter<=26;counter++)          //向右转
{
    P1_1=1;
    delay_nus(1700);
    P1_1=0;

    P1_0=1;
    delay_nus(1700);
    P1_0=0;

    delay_nms(20);
}

for(counter=1;counter<=65;counter++)          //向后
{
    P1_1=1;
    delay_nus(1300);
    P1_1=0;

    P1_0=1;
    delay_nus(1700);
    P1_0=0;

    delay_nms(20);
```

```
    }
    while(1);
}
```

该你了——以一个轮子为支点旋转

你可以使机器人绕一个轮子旋转。诀窍是使一个轮子不动而另一个旋转。例如，保持左轮不动而右轮从前面顺时针旋转，机器人将以左轮为轴旋转。

```
P1_1=1;
delay_nus(1500);
P1_1=0;
P1_0=1;
delay_nus(1300);
P1_0=0;
delay_nms(20);
```

如果想使它从前面向右旋转，很简单，停止右轮，左轮从前面逆时针旋转。

```
P1_1=1;
delay_nus(1700);
P1_1=0;
P1_0=1;
delay_nus(1500);
P1_0=0;
delay_nms(20);
```

这些命令使机器人从后面向右旋转。

```
P1_1=1;
delay_nus(1300);
P1_1=0;
P1_0=1;
delay_nus(1500);
P1_0=0;
delay_nms(20);
```

最后这些命令使机器人从后面向左旋转。

```
P1_1=1;
delay_nus(1500);
```

```
P1_1=0;
P1_0=1;
delay_nus(1700);
P1_0=0;
delay_nms(20);
```

把 ForwardLeftRightBackward.c 另存为 PivotTests.c。

用刚讨论过的代码片段替代前进、左转、右转和后退相应的代码片段，通过更改每个 for 循环的循环次数来调整每个动作的运行时间，更改注释来反映每个新的旋转动作。

运行更改后的程序，验证上述旋转运动是否不同。

任务 2　匀加速/减速运动

在前面机器人运动过程中，你是否发现机器人在每次启动和停止的时候，是不是有些太快，从而导致机器人几乎要倾倒。为什么会这样呢？

回忆一下学过的物理知识，还记得牛顿第二定律和运动学知识吗？前面的程序总是直接给机器人伺服电机输出最大速度控制命令。根据运动学知识，一个物体要从零加速到最大运动速度时，时间越短，所需加速度就越大。而根据牛顿定律，加速度越大，物体所受的惯性力就越大。因此，前面的程序因为没有给机器人足够的加速时间，所以受到的惯性力就比较大，从而导致机器人在启动和停止时有一个较大的前倾力或者后坐力。要消除这种情况，就必须让机器人速度逐渐增加或逐渐减小。采用均匀加速或减速是一种比较好的速度控制策略，这样不仅可以让机器人运动得更加平稳，还可以增加机器人电机的使用寿命。

编写匀加速运动程序

匀加速运动程序片段示例：

```
for(pulseCount=10;pulseCount<=200;pulseCount=pulseCount+1)
{
    P1_1=1;
    delay_nus(1500+pulseCount);
    P1_1=0;

    P1_0=1;
    delay_nus(1500-pulseCount);
    P1_0=0;
    delay_nms(20);
}
```

上述 for 循环语句能使机器人的速度由停止到全速。循环每重复执行一次，变量 pulseCount 就增加 1：第 1 次循环时，变量 pulseCount 的值是 10，此时发给 P1_1、P1_0 的脉冲宽度分别为 1.51ms、1.49ms；第 2 次循环时，变量 pulseCount 的值是 11，此时发给 P1_1、P1_0 的脉冲宽度分别为 1.511ms、1.489ms。随着变量 pulseCount 值的增加，电机的速度也在逐渐增加。到执行第 190 次循环时，变量 pulseCount 的值是 200，此时发给 P1_1、P1_0 的脉冲宽度分别为 1.7ms、1.3ms，电机全速运转。

回顾第 2 讲的任务 3，for 循环也可以由高向低计数。可以通过使用 for(pulseCount=200; pulseCount>=0;pulseCount=pulseCount-1)来实现速度的逐渐减小。下面是一个使用 for 循环实现电机速度逐渐增加到全速，然后逐步减小的例子。

例程：StartAndStopWithRamping.c

```
#include<BoeBot.h>
#include<uart.h>
int main(void)
{
    int pulseCount;
    uart_Init();
    printf("Program Running!\n");

    for(pulseCount=10;pulseCount<=200;pulseCount=pulseCount+1)
    {
        P1_1=1;
        delay_nus(1500+pulseCount);
        P1_1=0;

        P1_0=1;
        delay_nus(1500-pulseCount);
        P1_0=0;
        delay_nms(20);
    }

    for(pulseCount=1;pulseCount<=75;pulseCount++)
    {
        P1_1=1;
        delay_nus(1700);
        P1_1=0;
```

```
            P1_0=1;
            delay_nus(1300);
            P1_0=0;
            delay_nms(20);
        }

        for(pulseCount=200;pulseCount>=0;pulseCount=pulseCount-1)
        {
            P1_1=1;
            delay_nus(1500+pulseCount);
            P1_1=0;

            P1_0=1;
            delay_nus(1500-pulseCount);
            P1_0=0;
            delay_nms(20);
        }
        while(1);
    }
```

- 输入、保存并运行程序 StartAndStopWithRamping.c；
- 验证机器人是否逐渐加速到全速，保持一段时间，然后逐渐减速到停止。

可以创建一个程序，将加速或减速与其他的运动结合起来。下面是一个逐渐增加速度向后走而不是向前走的例子。加速向后走与向前走的唯一不同之处在于发给 P1_1 的脉冲宽度由 1.5ms 逐渐减小，而向前走是逐渐增加的；相应的，发给 P1_0 的脉冲宽度由 1.5ms 逐步增加。

```
    for(pulseCount=10;pulseCount<=200;pulseCount=pulseCount+1)
    {
        P1_1=1;
        delay_nus(1500-pulseCount);
        P1_1=0;
        P1_0=1;
        delay_nus(1500+pulseCount);
        P1_0=0;
```

```
        delay_nms(20);
    }
```

也可以通过增加程序中两个 pulseCount 的值到 1500 来创建一个在旋转中匀变速的程序。通过逐渐减小程序中两个 pulseCount 的值，可以沿另一个方向匀变速旋转。这是一个匀变速旋转 1/4 周的例子。

```
    for(pulseCount=1;pulseCount<=65;pulseCount++)//匀加速向右转
    {
        P1_1=1;
        delay_nus(1500+pulseCount);
        P1_1=0;
        P1_0=1;
        delay_nus(1500+pulseCount);
        P1_0=0;
        delay_nms(20);
    }
    for(pulseCount=65;pulseCount>=0;pulseCount--)//匀减速向右转
    {
        P1_1=1;
        delay_nus(1500+pulseCount);
        P1_1=0;
        P1_0=1;
        delay_nus(1500+pulseCount);
        P1_0=0;
        delay_nms(20);
    }
```

从任务 1 中打开程序 ForwardLeftRightBackward.c，另存为 ForwardLeftRightBackward－Ramping.c。

更改新的程序，使机器人的每一个动作都能够匀加速和匀减速。

提示：你可以使用上面的代码片段和 StartAndStopWithRamping.c 程序中相似的片段。

任务 3　用函数调用简化运动程序

在后面讲节中，机器人将执行各种运动来避开障碍物和完成其他动作。不过，无论机器人要执行何种动作，都离不开前面讨论的各种基本动作。为了各种应用程序方便使用这些基

本动作程序，可以将这些基本动作放在函数中，供其他函数调用来简化程序。

C 语言提供了强大的函数定义功能。一个 C 程序就是由一个主函数和若干个其他函数构成，由主函数调用其他函数，其他函数也可以相互调用。同一个函数可以被一个或多个函数调用任意多次。

实际上，为了实现复杂的程序设计，在所有的计算机高级语言中都有子程序或者子过程的概念。在C 语言程序中，子程序的作用就是由函数来完成的。

从函数定义的角度来看，函数有以下两种

（1）标准函数，即库函数。由开发系统提供，用户不必自己定义而直接使用，只需在程序前包含有该函数原型的头文件即可在程序中直接调用。例如，前面已经用到的串口标准输入（printf）和输出（scanf）函数。应该说明，不同的语言编译系统提供的库函数的数量和功能会有一些不同，但许多基本函数是共同的。

（2）用户定义函数，以解决你的专门需要。不仅要在程序中定义函数本身，而且在主调函数模块中还必须对该被调函数进行类型说明，然后才能使用。

从有无返回值角度来看，函数又分为以下两种

（1）有返回值函数。函数被调用执行完后将向调用者返回一个执行结果，称为函数返回值。由用户定义的返回函数值，必须在函数定义中明确其类型。

（2）无返回值函数。此类函数用于完成某项特定的处理任务，执行完成后不向调用者返回函数值。用户在定义此类函数时可指定它的返回值为“空类型”，即“void”。

从主调函数和被调函数之间数据传送的角度看，函数也可分为两种

（1）无参函数。函数定义、说明及调用中均不带参数，主调函数和被调函数之间不进行参数传送。此类函数通常用来完成一组指定的功能，可以返回或不返回函数值。

（2）有参函数。在函数定义及说明时都有参数，称为形式参数（简称为形参）。在函数调用时也必须给出参数，称为实际参数（简称为实参）。进行函数调用时，主调函数将把实参的值传送给形参，供被调函数使用。

第 1 讲就已经给出了函数定义的一般形式：

```
类型标志符　函数名(形式参数列表)
{
    声明部分
    语句
}
```

其中，类型标志符和函数名称为函数头。类型标志符指明了本函数的类型，函数的类型

实际上是函数返回值的类型。函数名是由用户定义的标志符，函数名后有一个括号（不可少写）。若函数无参数，则括号内可不写内容或写“void”；若有参数，则形式参数列表给出各种类型的变量，各参数之间用逗号间隔。

{ }中的内容称为函数体。函数体中的声明部分，是对函数体内部用到的变量的类型说明。在很多情况下都不要求函数有返回值，此时函数类型符可以写为 void。

main 函数的返回值

前面说过，main 函数是不能被其他函数调用的，那它的返回值类型 int 是怎么回事呢？

其实不难理解，main 函数执行完后，它的返回值是给操作系统的。虽然在 main 函数体内并没有什么语句来指出返回值的大小，但系统默认的处理方式是：当 main 函数成功执行时，它的返回值为 1；否则为 0。

现在看看下面的函数定义。

```
void Forward(void)
{
    int i;
    for(i=1;i<=65;i++)
    {
      P1_1=1;
      delay_nus(1700);
      P1_1=0;
      P1_0=1;
      delay_nus(1300);
      P1_0=0;
      delay_nms(20);
    }
}
```

Forward 函数可以使机器人向前运动约 1.5s，该函数没有形式参数，也没有返回值。在主程序中，可以调用它来让你的机器人向前运动约 1.5s。但是这个函数并没有太大的使用价值，如果想让你的机器人向前运动 2s，该怎么办呢？是重新写一个函数来实现这个运动吗？当然不是！通过修改上面的函数，给它增加两个形式参数，一个是脉冲数量，另一个是速度参数。这样主程序调用时就可以按照你的要求灵活设置这些参数，从而使函数真正成为一个有用的模块。重新定义向前运动函数如下：

```
void Forward(int PulseCount，int Velocity)
/* Velocity should be between 0 and 200    */
```

```
{
    int i;
    for(i=1;i<=PulseCount;i++)
    {
      P1_1=1;
      delay_nus(1500+Velocity);
      P1_1=0;
      P1_0=1;
      delay_nus(1500-Velocity);
      P1_0=0;
      delay_nms(20);
    }
}
```

函数定义下方，增加了一行注释，提醒你在调用该函数时，速度参量的值必须在 0～200 之间。

注释符

除“//”外，C 语言还提供了另一种语句注释符——“/*”和“ */”。

“/*”和“*/”必须成对使用，在它们之间的内容将被注释掉。它的作用范围比“//”大：“//”仅仅对它所在的一行起注释作用；但“/*…*/”可以对多行注释。

注释是你在学习程序设计时要养成的良好习惯。

下面是一个完整的使用向前、左转、右转和向后 4 个函数的例程。

例程：MovementsWithFunctions.c

输入、保存、编译、下载并运行程序 MovementsWithFunctions.c。

```
#include<BoeBot.h>
#include<uart.h>
void Forward(int PulseCount,int Velocity)
/* Velocity should be between 0 and 200    */
{
    int i;
    for(i=1;i<= PulseCount;i++)
    {
        P1_1=1;
        delay_nus(1500+ Velocity);
        P1_1=0;
```

```
            P1_0=1;
            delay_nus(1500- Velocity);
            P1_0=0;
            delay_nms(20);
        }
    }
    void Left(int PulseCount,int Velocity)
    /* Velocity should be between 0 and 200    */
    {
        int i;
        for(i=1;i<= PulseCount;i++)
        {
            P1_1=1;
            delay_nus(1500-Velocity);
            P1_1=0;
            P1_0=1;
            delay_nus(1500-Velocity);
            P1_0=0;
            delay_nms(20);
        }
    }
    void Right(int PulseCount,int Velocity)
    /* Velocity should be between 0 and 200    */
    {
        int i;
        for(i=1;i<= PulseCount;i++)
        {
            P1_1=1;
            delay_nus(1500+Velocity);
            P1_1=0;
            P1_0=1;
            delay_nus(1500+Velocity);
            P1_0=0;
            delay_nms(20);
        }
    }
    void Backward(int PulseCount,int Velocity)
    /* Velocity should be between 0 and 200    */
    {
```

```
    int i;
    for(i=1;i<= PulseCount;i++)
    {
        P1_1=1;
        delay_nus(1500-Velocity);
        P1_1=0;
        P1_0=1;
        delay_nus(1500+ Velocity);
        P1_0=0;
        delay_nms(20);
    }
}
int main(void)
{
    uart_Init();
    printf("Program Running!\n");

    Forward(65,200);
    Left(26,200);
    Right(26,200);
    Backward(65,200);
    while(1);
}
```

这个程序的运行结果与程序 ForwardLeftRightBackward.c 产生的效果是相同的。很明显，还有许多方法可以构造一个程序而得到同样的结果。实际上，4 个函数的具体实现部分几乎完全一样，有没有可能将这些函数进行归纳，用一个函数来实现所有这些功能呢？当然有，前面的 4 个函数都用了两个形式参数，一个是控制时间的脉冲个数，另一个是控制运动速度的参数，而 4 个函数实际上代表了 4 个不同的运动方向。如果能够通过参数控制运动方向，显然这 4 个函数就完全可以简化成为一个更为通用的函数，它不仅可以涵盖以上 4 个基本运动，同时还可以使机器人朝你希望的方向运动。

由于机器人由两个轮子驱动，实际上两个轮子的不同速度组合控制着机器人的运动速度和方向，因此可以直接用两个车轮的速度作为形式参数，就可以将所有的机器人运动用一个函数来实现。

例程：MovementsWithOneFuntion.c

这个例子使你的机器人做同样动作，但是它只用了一个子函数来实现。

```
#include <BoeBot.h>
#include <uart.h>
void Move(int counter,int PC1_pulseWide,int PC0_pulseWide)
{
    int i;
    for(i=1;i<=counter;i++)
    {
        P1_1=1;
        delay_nus(PC1_pulseWide);
        P1_1=0;
        P1_0=1;
        delay_nus(PC0_pulseWide);
        P1_0=0;
        delay_nms(20);
    }
}
int main(void)
{
    uart_Init();
    printf("Program Running!\n");

    Move(65,1700,1300);
    Move(26,1300,1300);
    Move(26,1700,1700);
    Move(65,1300,1700);
    while(1);
}
```

- 输入、保存并运行程序 MovementsWithOneFuntion.c；
- 你的机器人是否执行了向前、向左、向右、向后运动呢？
- 修改 MovementsWithOneFuntion.c，使机器人走一个正方形。第一边和第二边向前走，另外两个边向后走。

任务 4　高级主题——用数组建立复杂运动

到目前为止，你已经试过 3 种不同的编程方法来使机器人向前走、左转、右转和向后走。每种方法都有它的优点，但是如果要让机器人执行一个更长、更复杂的动作时用这些方法都很麻烦。下面要介绍的两个例子将用子函数来实现每个简单的动作，将复杂的运动存储在数

组中，然后在程序执行过程中读出并解码。避免了重复调用一长串子函数。这里，要用到 C 语言的一种新的数据类型——数组。

前面，只用到了 C 语言的基本数据类型之一的整型数据，以 int 作为类型说明符。另外一种基本数据类型是字符型，以 char 作为类型说明符。

字符型数据

字符常量

字符常量是指用一对单引号括起来的一个字符，如'a'、'9'、'! '。字符常量中的单引号只起到定界作用并不表示字符本身。单引号中的字符不能是单引号（'）和反斜杠（\），它们特有的表示法将在转义字符中介绍。

在 C 语言中，字符是按其所对应的 ASCII 码值来存储的，一个字符占一个字节，见表 3-1。

表 3-1　字符与其所对应的 ASCII 码值

字符	ASCII 码值
!	33
0	48
1	49
9	57
A	65
B	66
a	97
b	98

ASCII

ASCII 是美国标准信息交换码（American Standard Code for Information Interchange）的缩写，用来制订计算机中每个符号对应的代码，也称为计算机的内码（code）。

每个 ASCII 以 1 个字节（Byte）储存，从 0 到数字 127 代表不同的常用符号。例如，大写 A 的 ASCII 是 65，小写 a 的 ASCII 则是 97。这套内码加上了许多外文和表格等特殊符号，成为目前常用的内码。

注意字符'9'和数字 9 的区别，前者是字符常量，后者是整型常量，它们的含义和在计算机中的存储方式都截然不同。

由于 C 语言中字符常量是按整数存储的，所以字符常量可以像整数一样在程序中参与相关的运算，如：

```
'a'-32;          //执行结果 97-32=65
'A'+32;          //执行结果 65+32=97
'9'-9;           //执行结果 57-9=48
```

转义字符

转义字符是一种特殊的字符常量，以反斜杠“\”开头，后跟一个或几个字符。转义字符具有特定的含义，不同于字符原有的意义，故称“转义”字符。例如，前面各例题 printf 函数中用到的“\n”就是一个转义字符，其意义是“回车换行”。

通常使用转义字符表示用一般字符不便于表示的控制代码。例如，用于表示字符常量的单引号（'），用于表示字符串常量的双引号（"）和反斜杠（\）等。

表 3-2 给出了 C 语言中常用的转义字符。

表 3-2　C 语言中常用的转义字符

转 义 字 符	含　　义	ASCII 值（十进制）
\b	退格（BS）	008
\n	换行（LF）	010
\t	水平制表（HF）	
\\	反斜杠	092
\'	单引号字符	039
\"	双引号字符	034
\0	空字符（NULL）	
\ddd	任意字符三位八进制	
\xhh	任意字符二位十六进制	

广义地讲，C 语言字符集中的任何一个字符均可用转义字符来表示。表 3-2 中的\ddd 和\xhh 正是为此而提出的。ddd 和 hh 分别为八进制和十六进制的 ASCII 代码。例如，\101 表示字母“A”，\102 表示字母“B”，\134 表示反斜线，\XOA 表示换行等。

字符变量

字符变量用来存放字符常量，注意只能存放一个字符。

字符变量的定义形式如下：

```
char c1,c2;
```

它表示 c1 和 c2 为字符变量，各放入一个字符。因此可以用下面语句对 c1、c2 赋值：

```
c1='a';c2='A';
```

数组

在程序设计中，为了处理方便，可以把具有相同类型的若干变量按有序的形式组织起来。这些按序排列的同类数据元素的集合称为数组。一个数组可以分解为多个数组元素，根据数组元素数据类型的不同，数组可以分为多种不同类型。数组又分为一维数组、二维数组甚至三维数组。本节只用到一维数组。一维数组的定义方式：

类型说明符 数组名[常量表达式];