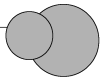
第3章

程序控制结构



将应用问题中的数据和运算引入计算机后，还需要将求解问题的算法使用C＃程序实现出来。语句是C＃程序的最小单位，程序由一条一条的语句组成，程序运行的过程就是语句逐条执行的过程，而语句执行的次序称为流程。因此在多数情况下，算法实现的结果为一定数量的语句和执行流程。

C＃语句分为简单语句、复合语句和控制语句，具有顺序、选择和循环3种基本结构。

3.1 语句

3.1.1 简单语句

简单语句包括表达式语句、空语句和声明语句，它以分号表示语句结束。

1. 表达式语句

C＃语言中，并不是所有的表达式都可以形成表达式语句。具体而言，不允许像x＋y和x＝＝1之类的只计算一个值(此值将被放弃)的表达式作为语句使用。表达式语句的形式为

```
表达式;
```

表达式语句用于对表达式求值。可用作语句的表达式包括方法调用、使用new运算符的对象分配、使用各种赋值运算符的赋值，以及使用自增运算符(＋＋)和自减运算符(－－)的增量和减量运算。如下所示都是正确的表达式语句：

```
x=a+b;                                  //赋值表达式语句
sum+=i;                                 //复合赋值表达式语句
i++;                                    //自增运算符表达式语句
y--;                                    //自减运算符表达式语句
o=new object();                         //对象分配语句
Console.WriteLine("hello world!");      //方法调用语句
```

表达式语句是程序中使用最多的语句，因为程序多数情况下表现为计算和功能执行。

2. 空语句

空语句不进行任何操作，其一般格式如下：

```
;
```

在需要语句但又不进行任何操作的时候使用空语句。例如，循环语句中的循环体。下面语句的功能是从键盘连续输入多个字符直到输入回车符。

```
while(Console.Read()!='\n');                    //使用空语句
```

这里的循环体就是空语句，因为 while 语句要求必须有循环体，而 Console.Read() !='\n' 已经完成了语句功能，此时的循环体不需要做什么。

空语句可以用在任何允许使用语句的地方。但是，由于编程疏忽而意外出现的空语句，虽然不会造成编译器的编译错误，却有可能产生程序的逻辑错误。例如，if 语句中的空语句可能会使条件判断变相失效。下面的程序段中不管 a 值是正数还是负数，程序都会输出提示信息“This is a positive number!”。

```
if(a>0);
    Console.WriteLine("This is a positive number!");
```

3. 声明语句

C＃中用于定义常量、变量等的语句称为声明语句，例如：

```
double area;                                    // 变量声明语句
double radius=2;                                // 变量声明语句
const double pi=3.14159;                        // 常量声明语句
```

C＃中声明语句可以出现在程序中任何允许语句出现的地方，以此实现变量和常量定义的局部性。一般情况下，声明语句最好放在方法或复合语句的开头位置。

3.1.2　复合语句

复合语句又称为块语句或块(blocks)，由一个括在大括号内的语句列表组成，而语句列表(Statement List) 由一个或多个顺序编写的语句组成。语句形式为

```
{语句列表}
```

如果没有语句列表，则称块语句是空的，它与空语句等价。复合语句中可以包含 C＃中的任何语句，如声明语句、表达式语句、选择语句、循环语句、跳转语句，它甚至可以包含另一个复合语句。复合语句的左右大括号明确描述了复合语句的开始和结束位置，因此其后不需要加分号。如下所示都是 C＃中正确的复合语句：

```
//复合语句 1
    {
        int a=5,b=2,t;                          //声明语句
```

```
        t=a;                                                    //表达式语句
        a=b;                                                    //表达式语句
        b=t;                                                    //表达式语句
    }
//复合语句2:嵌套复合语句
    {
        int a=5,b=2,t;                                          //声明语句
        if(a>b)                                                 //选择语句
        {
            t=a;
            a=b;
            b=t;
        }
        Console.WriteLine("a is {0},b is {1}.", a, b);          //表达式语句
    }
```

不管复合语句里包括多少条语句,语法上都把它作为一条语句看待。因此,复合语句经常用在语法上只允许出现一条语句,而一条简单语句又无法满足其功能需求的地方。使用复合语句利于将复杂的语句形式简单化、结构化。

3.1.3 注释

程序中经常需要添加一些辅助程序阅读和理解的说明内容,称为注释。C# 支持两种形式的注释:单行注释和带分隔符的注释。单行注释(Single-line Comment)以字符序列//开头并延续到本行的结尾,格式如下:

```
//注释内容
```

带分隔符的注释(Delimited Comment)以字符序列"/*"开头,以字符序列"*/"结束,可以跨多行。格式如下:

```
/*
注释内容
*/
```

C# 中注释不能嵌套。字符序列"/*"和"*/"在单行注释中没有任何特殊含义,字符序列"//"和"/*"在带分隔符的注释中没有任何特殊含义。同时,C# 不对字符和字符串内的内容做注释处理。

需要特别注意的是,C# 中注释仅是对源程序的文字说明。注释不是程序代码,并且不会对程序运行产生任何影响,在编译期间所有的注释将会被忽略。

3.1.4 语句的写法

C# 中,语句的写法一般有以下规定或惯例。

(1) 虽然 C# 允许一个程序行里写多条语句,或一条语句分多行书写,但是多数情况下,在一个程序行里只写一条语句,这些写法利于阅读、理解和调试,写法清晰。

(2) 注意使用空格或 Tab 键来合理地对程序语句进行间隔、缩进和对齐，使程序形成逻辑相关的块结构，养成优美的程序编写风格。

(3) 由于计算机屏幕宽度有限，C＃中过长的语句可以分成多个程序行来书写，例如：

```
Console.WriteLine("a is {0},b is {1}.",
a, b);
```

一条语句分成多行来书写时需要注意：C＃规定回车换行也是空白符，所以不能在关键字、标识符和常量等中间拆分，否则会产生编译错误。如下是错误的分行写法，将会产生编译错误。

```
Console.WriteLine("a is {0},
b is {1}.",a, b);                                    //产生编译错误
```

3.2　输入和输出

输入和输出是用户与计算机之间进行交流的方法。输入指计算机用户通过键盘、鼠标等外部输入设备将数据送入计算机，输出指计算机通过显示器、打印机等外部输出设备将数据送出呈献给计算机用户。

C＃中不提供输入和输出语句，其输入和输出操作是借助于一些预定义类来实现的。Console 类提供用于从控制台读取单个字符或整行的方法，并且该类还可将值类型的数据、字符数组以及对象集自动转换为格式化或未格式化的字符串，然后将该字符串写入控制台。由于 Console 类隶属于命名空间 System，所以需要在源程序开头使用 using 语句进行命名空间引入，格式如下：

```
using system;
```

如果没有引入该命名空间，则需要在程序设计时使用 Console 类的完全限定名，即按照层次结构完整指定 Console 类所在的命名空间，此时应将 Console 替换为 System.Console。

3.2.1　输入方法

Console 类支持用户使用标准输入设备(如键盘和鼠标)向计算机输入数据，其实现的输入方法有 Read、ReadLine、ReadKey 等。

1. Read 方法

Read 方法从标准输入流读取下一个字符。其返回值是输入流中下一个字符的 Unicode 编码值，返回值类型是 System.Int32；如果当前没有更多的字符可供读取，则返回－1。其方法定义为

```
public static int Read();
```

在输入字符时，Read 方法会阻止其返回，只有当用户按 Enter 键时该方法才会终

止。按 Enter 键会在输入内容后面追加一个与平台有关的行终止序列(例如,Windows 追加一个回车符和换行符序列)。对 Read 方法的后续调用一次检索输入中的一个字符,检索完最后一个字符后,Read 会再次阻止其返回,并重复上述循环。

注意,Read 方法只有在下列情况下才返回－1:①同时按 Ctrl＋Z 键,此按键组合发出到达文件尾条件;②按发出到达文件尾条件的等效键,例如 Windows 中的 F6 功能键;③将输入流重定向到具有实际的文件尾字符的源,例如文本文件。

Read 方法的用法示例如下:

```
char ch;
int x;
x=Console.Read();                    //读取字符,返回字符的 Unicode 编码值给变量 x
try
{
        ch=Convert.ToChar(x);        //将 Unicode 编码转换成对应的字符给变量 ch
        Console.WriteLine("The char is:{0}",ch);
}
catch(OverflowException e)           //转换不成功时进行异常处理
{
        Console.WriteLine("{0} Value read={1}.转换不成功", e.Message, x);
}
```

2. ReadLine 方法

ReadLine 方法从标准输入流读取下一行字符,其返回值类型为 System.String;如果没有更多的可用行,则返回 null。其方法定义为

```
public static string ReadLine();
```

行被定义为后跟回车符(十六进制 0x000d)、换行符(十六进制 0x000a)或 Environment.NewLine 属性值的字符序列。ReadLine 方法返回的字符串不包含终止字符。

如果在该方法从控制台读取输入时按 Ctrl＋Z 键,该方法将返回 null。用户可以借此在循环中调用 ReadLine 方法时防止进一步的键盘输入。下面是 ReadLine 方法的使用示例。

```
string line;
do
{
    line=Console.ReadLine();                    //从标准输入流读取下一行字符
    if(line !=null)
        Console.WriteLine("      "+line);      //字符串不为 null 时,输出
} while(line !=null);          //循环执行读入一行字符并输出该行字符,直到按 Ctrl+Z 键
```

注:如果有多个信息在同一行输入,信息之间用空格或者其他符号隔开,则可使用

ReadLine 方法读取后，调用 string 类型的 Split 方法进行切分。例如，输入一个日期（格式 yyyy/mm/dd），获取其年月日信息的代码如下：

```
int year, month, day;
string date=Console.ReadLine();          //从标准输入流读取一行日期信息
string[]datetemp=date.Split('/');        //将日期信息切分成年、月、日 3 个信息子串
year=Convert.ToInt32(datetemp[0]);       //将年信息子串转为整数类型
month=Convert.ToInt32(datetemp[1]);      //将月信息子串转为整数类型
day=Convert.ToInt32(datetemp[2]);        //将日信息子串转为整数类型
```

有关 Split 方法的更多使用请自行查阅资料。

3. ReadKey 方法

ReadKey 方法获取用户按下的下一个字符或功能键。返回值类型为 System.ConsoleKeyInfo，描述 ConsoleKey 常数和对应于按下的控制键的 Unicode 字符（如果存在这样的字符）。ConsoleKeyInfo 对象还以 ConsoleModifiers 值的按位组合描述是否在按下该控制键的同时按下了 Shift、Alt 或 Ctrl 键中的一个或多个。该方法有如下两种重载方式。

```
方式 1: public static ConsoleKeyInfo ReadKey();
方式 2: public static ConsoleKeyInfo ReadKey(bool intercept);
```

方式 1 中，ReadKey 方法获取用户按下的键后显示在控制台窗口中。下面是方式 1 的使用示例：

```
ConsoleKeyInfo cki;
cki=Console.ReadKey();                   //从键盘读取用户按下的下一个字符或功能键
Console.Write(" ---You pressed ");
/* 根据 ConsoleModifiers 值的按位组合描述是否在按下该控制键的同时按下了 Shift、Alt
或 Ctrl 键中的一个或多个 */
if((cki.Modifiers & ConsoleModifiers.Alt)!=0)Console.Write("ALT+");
if((cki.Modifiers & ConsoleModifiers.Shift)!=0)Console.Write("SHIFT+");
if((cki.Modifiers & ConsoleModifiers.Control)!=0)Console.Write("CTRL+");
Console.WriteLine(cki.Key.ToString());     //输出按下的 Unicode 字符
```

方式 2 中，ReadKey 方法获取用户按下的键后可以选择显示在控制台窗口中。参数 intercept 值为 true 时，按下的键将被截获，不会显示在控制台窗口中；参数值为 false 时，将在控制台窗口中显示按下的键。方式 2 使用方法与方式 1 相似，不再示例。

3.2.2　输出方法

Console 类支持计算机向标准输出设备（如显示器）输出数据，其实现的输出方法有 Write、WriteLine。

1. Write 方法

Console 类的 Write 方法有 18 种重载定义,分别用以实现将布尔型、数值型、字符型、字符串型、对象型等信息写入标准输出流中。其调用形式如下:

```
方式 1: Write(输出项);                    /* 将指定的输出项信息写入标准输出流 */
方式 2: Write(格式控制,输出项列表);       /* 将各输出项按指定的格式写入标准输出流 */
```

Write 方法的使用示例如下:

```
char c='c';
int d=10;
Console.Write(true);                  //使用重载方式 1 输出 bool 型值
Console.Write(c);                     //使用重载方式 2 输出字符
Console.Write(d);                     //使用重载方式 6 输出 int 型值
Console.Write("Hello world!");        //重载方式 10 输出字符串
Console.Write("{0:x}", d);            //将变量 d 的值按照十六进制形式输出,输出:a
```

2. WriteLine 方法

Console 类的 WriteLine 方法用以将布尔型、数值型、字符型、字符串型、对象型等信息写入标准输出流中,它与 Write 方法的不同就是在输出信息后附加当前行终止符,即输出当前信息后自动换行。WriteLine 方法也有多种重载定义,上述 Write 方法的每种重载都对应有 WriteLine 方法的重载定义。除此之外,WriteLine 方法还可用以只把当前行终止符写入标准输出流,即输出一个空行。因此,WriteLine 方法有 19 种重载定义形式。

WriteLine 方法的调用形式如下:

```
方式 1: WriteLine(输出项);               /* 将指定的输出项信息附加当前行终止符写入标
                                             准输出流 */
方式 2: WriteLine(格式控制,输出项列表);  /* 将各输出项按指定的格式写入标准输出流并附
                                             加当前行终止符 */
方式 3: WriteLine();                      //输出空行
```

下面是 WriteLine 方法的使用示例。

```
char c='c';
int d=10;
Console.WriteLine(true);              //输出 bool 型值后回车换行
Console.WriteLine(c);                 //输出字符后回车换行
Console.WriteLine(d);                 //输出 int 型值后回车换行
Console.WriteLine("Hello world!");    //输出字符串后回车换行
Console.WriteLine("{0:x}", d);        //将变量 d 的值按照十六进制形式输出后回车换行
Console.WriteLine();                  //输出一个空行
```

3. 格式字符串

使用 Console 类的 Write 方法和 WriteLine 方法输出时,可以通过设置输出格式字符

串以更合适的格式输出信息。例如，将某个数值作为货币或者某个小数位数的定点值来显示、设置信息输出的对齐方式等。下面对这两个方法中格式字符串的设置方法进行详细介绍。

当使用 Write 和 WriteLine 方法的输出格式设置时，由对象列表和复合格式字符串一起作为方法的参数。复合格式字符串由固定文本和索引占位符混合组成，其中固定文本是所选择的任何字符串，索引占位符称为格式项，每个格式项对应于列表中的一个对象或装箱的结构，每个格式项都被列表中相应对象的字符串表示形式取代。格式设置操作产生的结果字符串由原始固定文本和列表中对象的字符串表示形式混合组成。

每个格式项都采用下面的形式并包含以下组成部分：

```
{索引[,对齐][:格式字符串]}
```

注意：格式项必须使用成对的大括号（“{”和“}”）。下面对格式项的每个组成部分进行介绍。

（1）索引，即强制“索引”组件（也叫参数说明符）是一个从 0 开始的数字，可标识对象列表中对应的项。也就是说，参数说明符为 0 的格式项对应列表中的第一个对象，参数说明符为 1 的格式项对应列表中的第二个对象，以此类推。

通过指定相同的参数说明符，多个格式项可以引用对象列表中的同一个元素。例如，通过指定类似于“{0:X} {0:E} {0:N}”的复合格式字符串，可以将同一个数值设置为十六进制、科学记数法和数字格式进行输出。

每个格式项都可以引用列表中的任一对象。例如，如果有 3 个对象，则可以通过指定类似于“{1} {0} {2}”的复合格式字符串来设置第 2、第 1 和第 3 个对象的格式。格式项未引用的对象会被忽略。如果参数说明符指定了超出对象列表范围的项，将导致运行时异常。

（2）对齐，即对齐组件。可选的“对齐”组件是一个带符号的整数，指示首选的设置了格式的字段宽度。如果“对齐”值小于设置了格式的字符串的长度，“对齐”会被忽略，并且使用设置了格式的字符串的长度作为字段宽度。如果“对齐”为正数，字段中设置了格式的数据为右对齐；如果“对齐”为负数，字段中的设置了格式的数据为左对齐。如果需要填充，则使用空白。如果指定“对齐”，就需要使用逗号。

（3）格式字符串，即格式字符串组件。可选的“格式字符串”组件是适合正在设置格式的对象类型的格式字符串。如果相应的对象是数值，则指定标准或自定义的数字格式字符串；如果相应的对象是 DateTime 对象，则指定标准或自定义的日期和时间格式字符串；如果相应的对象是枚举值，则指定枚举格式字符串。如果不指定“格式字符串”，则对数字、日期和时间或者枚举类型使用常规（“G”）格式说明符。如果指定“格式说明符”，需要使用冒号。表 3.1 描述了标准数字格式的说明符，表 3.2 描述了自定义数字格式的说明符。

表 3.1 标准数字格式说明符

格式说明符	含义	示例
C 或 c	使用货币符号把值格式化为货币;精度说明符为小数位数	Console.WriteLine("{0:C3}",12.5);输出:¥12.500
D 或 d	十进制数字字符串,需要的情况下有负数符号,只能和整数类型配合使用;精度说明符为输出字符串中的最少位数,如果实际数字的位数更少,则左边以 0 填充	Console.WriteLine("{0:d5}", 12);输出:00012 Console.WriteLine("{0:d3}", 1234);输出:1234
F 或 f	带有小数点的十进制数字字符串,需要的情况下有负数符号;精度说明符为小数的位数	Console.WriteLine("{0:f3}", 1234);输出:1234.000 Console.WriteLine("{0:f2}", 1.234);输出:1.23
G 或 g	在没有指定说明符的情况下,会根据值转换为定点或科学记数法表示的紧凑形式;精度说明符为有效位数。如果精度说明符被省略或为零,则数字的类型决定默认精度,分别是 Byte 或 SByte:3 位;Int16 或 UInt16:5 位; Int32 或 UInt32:10 位; Int64:19 位;UInt64:20 位; BigInteger:29 位;Single:7 位;Double:15 位;Decimal:29 位	Console.WriteLine("{0:G4}", 3.1415926);输出:3.142 Console.WriteLine("{0:G4}", 31415926);输出:3.142E+07
X 或 x	十六进制数字字符串,十六进制数字 A~F 会匹配说明符的大小写形式;精度说明符为输出字符串中的最少位数,如果实际数字的位数更少,则左边以 0 填充	Console.WriteLine("{0:X4}", 3142);输出:0C46 Console.WriteLine("{0:x}", 3142);输出:c46
N 或 n	和定点表示法相似,但是在每 3 个数字的一组中间有分隔符,从小数点开始向左数;精度说明符为小数位数	Console.WriteLine("{0:N2}", 3141.5926);输出:3,141.59
P 或 p	表示百分比的字符串,数字会乘以 100;精度说明符为小数的位数	Console.WriteLine("{0:P2}", 0.12345);输出:12.35%
R 或 r	保证输出字符串后如果使用 Parse 方法将其转换为数字,那么该值和原始值一样。精度说明符忽略	Console.WriteLine("{0:R}", 0.12345);输出:0.12345
E 或 e	具有位数和指数的科学记数法,指数前面加字母 E 的大小写与说明符一致;精度说明符为小数的位数	Console.WriteLine("{0:E3}", 0.12345);输出:1.235E-001

表 3.2 自定义数字格式说明符

格式说明符	含义	示例
0	零占位符,用对应的数字(如果存在)替换零;否则,将在结果字符串中显示零	Console.WriteLine("{0:0.00000000}", 3.1415926); 输出:3.14159260

续表

格式说明符	含　义	示　例
#	数字占位符，用对应的数字（如果存在）替换#标记；否则，不会在结果字符串中显示任何数字	Console.WriteLine("{0:#.#####}", 3.142);　　//输出 3.142 Console.WriteLine("{0:#.###}", 3.1415);　　//输出 3.142
.	小数点，确定小数点分隔符在结果字符串中的位置	Console.WriteLine("{0:00.00}", 1.234); 输出：01.23
,	组分隔符和数字比例换算，用作组分隔符和数字比例换算说明符。作为组分隔符时，它在各个组之间插入本地化的组分隔符字符。作为数字比例换算说明符，对于每个指定的逗号，它将数字除以 1000	Console.WriteLine("{0:##,#}", 2147483647); 输出：2,147,483,647 Console.WriteLine("{0:#,#,,}", 2147483647); 输出：2,147
%	百分比占位符，将数字乘以 100，并在结果字符串中插入本地化的百分比符号	Console.WriteLine("{0:%#0.00}", 0.3697); 输出：%36.97 Console.WriteLine("{0:#0.00%}", 0.3697); 输出：36.97%
‰	千分比占位符，将数字乘以 1000，并在结果字符串中插入本地化的千分比符号	Console.WriteLine("{0:#0.00‰}", 0.3697); 输出：369.70‰
E0，E+0，E−0，e0，e+0，e−0	指数表示法，如果后跟至少一个 0（零），则使用指数表示法设置结果格式。"E"或"e"指示指数符号在结果字符串中是大写还是小写。跟在"E"或"e"字符后面的零的数目确定指数中的最小位数。加号（+）指示符号字符总是置于指数前面。减号（−）指示符号字符仅置于负指数前面	Console.WriteLine("{0:0.0##e+00}", 1503.92311);输出：1.504e+03
'字符串' "字符串"	文本字符串分隔符，指示应复制到未更改的结果字符串的封闭字符	Console.WriteLine("{0:# 'degrees'}", 987654); 输出：987654　degrees

3.3　程序顺序结构

3.3.1　顺序执行

多数情况下，程序中的语句按照其书写顺序执行，上一条语句执行完后自动开始下一条语句的执行，这种执行称为顺序执行。因此，顺序执行的程序中语句的次序很重要，不能随意调整顺序执行的语句顺序，这将会导致程序结果出错。

如果在顺序执行的程序中出现了方法或函数调用,当执行到方法或函数调用语句时,会暂停当前程序的执行流程转而进入被调方法或函数内部开始执行,当从被调方法或函数返回后继续被暂停的流程。

3.3.2 跳转执行

顺序执行流程能满足很多简单问题的求解需求。但是,有很多问题的求解仅依靠顺序执行是不能满足要求的,比如,给定3个线段判断它们能不能组成一个三角形、计算从1累加到10 000的和等。因此,跳转执行也是求解问题必需的执行机制。

C#中有多个控制语句可以实现跳转执行,控制程序的执行流程。

(1) 选择语句:if语句、switch语句。

(2) 循环语句:while语句、do语句、for语句。

(3) 跳转语句:goto语句、break语句、continue语句、return语句、throw语句。

其中,goto语句实现使程序执行流程无条件跳转的功能,但是这种无条件跳转机制在很大程度上破坏了程序的结构化,降低了程序的可读性,而且所有含goto语句的程序都可以改写成不用goto语句的程序。因此,少用甚至不用goto是一种良好的编程习惯,鉴于此,本书对goto语句不再介绍。

3.4 程序选择结构

3.4.1 if 语句

if语句可以根据给定表达式的结果选择执行不同的语句,其语句形式有如下几种。

1. if 形式

```
if(表达式)
    语句
```

它的执行过程是先计算表达式,并且表达式必须是bool类型。如果表达式的值为true,则语句被执行,否则语句被跳过。其执行流程如图3.1(a)所示。

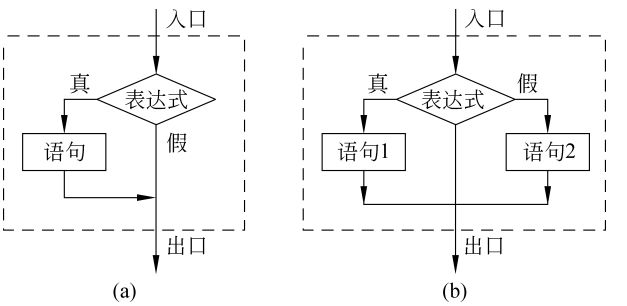


图3.1 if语句的两种形式

2. if-else 形式

```
if(表达式)
    语句 1
else
    语句 2
```

if-else 语句实现双分支,它同样先按照 bool 型计算表达式的结果。如果表达式的值为 true,则语句 1 被执行;如果表达式的值为 false,则语句 2 被执行。其执行流程如图 3.1(b)所示。

下面是 if 语句的使用示例代码:

```
if(a<0)
    value=-1*a;                                    //语句可以是简单语句,不需要花括号

if(a>b)
{
    t=a;                                           //语句也可以是复合语句,需要花括号
    a=b;
    b=t;
}

if(x>0)
    y=1;                                           //简单语句
else
{
    y=0;                                           //复合语句
    x=-1*x;
}
```

if 语句中的子语句既可以是简单语句,也可以是任何种类的其他语句,使用复合语句时必须使用花括号将其括起来。

【例 3.1】 判断输入的年份是闰年还是平年。闰年的年份值满足下面两个条件之一。

(1) 年份值能被 4 整除,但是不能被 100 整除;

(2) 年份值能被 400 整除。

```
1    using System;
2    class Leapyear
3    {
4        static void Main()
5        {
6            int year;
7            Console.WriteLine("请输入要判断的年份:");      /*输出提示信息*/
8            year=Convert.ToInt32(Console.ReadLine());
                                  /*从键盘接收用户输入的年份信息并转换为 int 型值*/
```

```
9           if((year %400==0)||(year %4==0 && year %100 !=0))
                                                    /*按照闰年的判断条件评定是否为闰年*/
10              Console.WriteLine("当前年份是闰年!");      //满足条件是闰年
11          else
12              Console.WriteLine("当前年份是平年!");      //不满足条件是平年
13      }
14  }
```

程序运行情况如下:

```
请输入要判断的年份:
2001↙
当前年份是平年!
```

3.4.2 switch 语句

switch 语句实现多路分支。它计算给定测试表达式的值,根据其结果选择从多个分支中的一个分支入口执行。其语句形式如下:

```
switch(测试表达式)
{
    case 常量表达式 1:语句序列 1
                      break;
    case 常量表达式 2:语句序列 2
                      break;
    …
    case 常量表达式 n:语句序列 n
                      break;
    default:
                      默认语句序列
                      break;
}
```

switch 语句的执行过程是先计算测试表达式,然后将测试表达式的结果与每个 case 分支的常量表达式值进行逐一比较。如果与某个常量表达式的值相等,就执行该分支标签后的语句序列,直到遇上 break 语句或其他跳转语句。因此,switch 语句要求常量表达式与测试表达式的类型一致。switch 语句的执行流程如图 3.2 所示。

switch 语句包含零个、一个或多个分支,但是任意两个分支的常量表达式值都不能相同。每个分支以一个或多个分支标签开始,每个分支必须以 break 语句或其他跳转语句结束,除非这个分支没有相应的语句序列。最常用来结束每个分支的是 break 语句。下面是 switch 语句的使用示例代码。

```
string s;
s=Console.ReadLine();
switch(s)
{
```

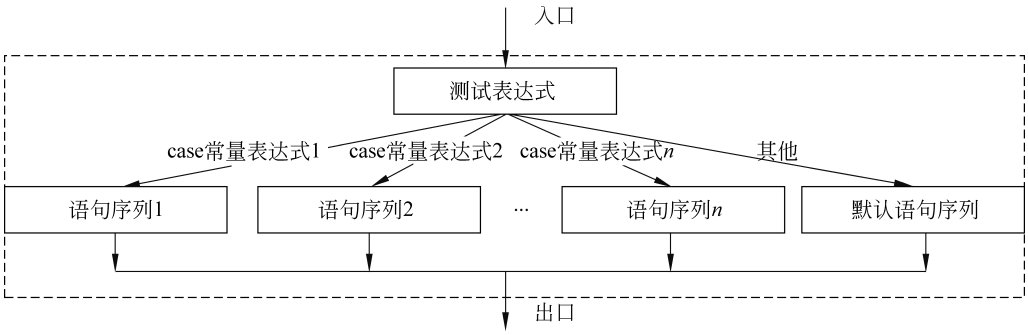


图 3.2　switch 语句执行流程

```
    case "yes":
          Console.WriteLine("YES");
          break;
    case "no":
          Console.WriteLine("NO");
          break;
    default:
          Console.WriteLine("ERROR");
          break;
}

    const int Five=5;
    int x=5;
    switch(x)
    {
           case Five:
                 Console.WriteLine("5");
                 break;
           case 10:
                 Console.WriteLine("10");
                 break;
    }

    int score;
        score=Convert.ToInt32(Console.ReadLine());
        switch(score/10)
        {
            case 10:
            case 9:
            case 8:
            case 7:
```

```
            case 6:
                Console.WriteLine("及格");     //成绩值为 60~100 则输出"及格"
                break;
            case 5:
            case 4:
            case 3:
            case 2:
            case 1:
            case 0:
                Console.WriteLine("不及格");  //成绩值为 0~59 则输出"不及格"
                break;
            default:
                Console.WriteLine("输入错误"); //成绩值不在 0~100 则输出"输入错误"
                    return;
        }
```

default 分支是可选的,表示当表达式的值与所有常量表达式的值都不相等时,将会执行默认语句序列,直到遇上 break 等跳转语句。一般情况下,switch 语句中都应该包含 default 分支,因为它的存在可以捕获程序中潜在的错误。

3.4.3 选择结构的嵌套

if 语句和 switch 语句中,分支的子语句或语句序列中可以是任意的语句。当在 if 语句的子语句或 switch 语句的分支语句序列中出现了又一个 if 语句或 switch 语句时,就形成了嵌套的选择结构。常见的选择结构嵌套形式有以下几种。

(1) if 语句嵌套 if 语句:如 if 语句的 else 分支的子语句出现另一个 if 语句,形成多次多个条件判断,达到一种多分支的效果;在 if 语句的 if 和 else 分支的子语句中都出现另一个 if 语句,形成 if 语句的多重嵌套。

(2) switch 语句与 switch 语句嵌套:如某个 case 分支后的语句序列中包括另一个 switch 语句。

(3) if 语句与 switch 语句嵌套:在 if 语句的子语句中出现 switch 语句或者 switch 语句的分支语句序列中出现 if 语句。

使用选择结构嵌套时,应注意不要使嵌套的层次太深或者嵌套的结构太复杂,这会使得代码的编写难度增大,代码的可读性降低。当问题中的判断机制比较复杂时,应学会在嵌套的选择结构和并列的选择结构之间做好权衡。

3.4.4 选择结构程序举例

【例 3.2】 将输入的五分制成绩转换成百分制成绩。

```
1   using System;
2   class ConvertScore
3   {
```

```
4       static void Main()
5       {
6           char score;
7           score=Convert.ToChar(Console.ReadLine());
8           if(score>='a' && score<='z')     //将用户输入的小写字母转换为大写字母
9               score=Convert.ToChar(score -32); //或 score=Char.ToUpper(score);
10          switch(score)          //根据用户输入的五分制成绩输出相应的百分制成绩段
11          {
12              case 'A':
13                  Console.WriteLine("90~100");
14                  break;
15              case 'B':
16                  Console.WriteLine("80~89");
17                  break;
18              case 'C':
19                  Console.WriteLine("70~79");
20                  break;
21              case 'D':
22                  Console.WriteLine("60~69");
23                  break;
24              case 'E':
25                  Console.WriteLine("0~59");
26                  break;
27              default:
28                  Console.WriteLine("输入错误");
29                  break;
30          }
31      }
32  }
```

程序运行情况如下：

```
A↙
90~100
```

【例 3.3】 输入月份和日期，输出对应的星座。星座和日期的对照关系如下：

白羊座：3 月 21 日－4 月 19 日　金牛座：4 月 20 日－5 月 20 日　双子座：5 月 21 日－6 月 21 日
巨蟹座：6 月 22 日－7 月 22 日　狮子座：7 月 23 日－8 月 22 日　处女座：8 月 23 日－9 月 22 日
天秤座：9 月 23 日－10 月 23 日　天蝎座：10 月 24 日－11 月 22 日　射手座：11 月 23 日－12 月 21 日
摩羯座：12 月 22 日－1 月 19 日　水瓶座：1 月 20 日－2 月 18 日　双鱼座：2 月 19 日－3 月 20 日

```
1   using System;
2   class Constellation
```

```
3   {
4       static void Main()
5       {
6           int m, d;
7           m=Convert.ToInt32(Console.ReadLine());  //从键盘接收月份信息
8           d=Convert.ToInt32(Console.ReadLine());  //从键盘接收日期信息
9           if((m>12 && m<1)||(d>31 && d<1))         //月、日期输入非法结束
10          {
11              Console.WriteLine("输入错误");
12              return;
13          }
14          switch(m)           //借助 case 语句和 if 语句的嵌套实现 12 个星座的查询
15          {
16              case 1:
17                  if(d<=19)
18                      Console.WriteLine("摩羯座");
19                  else
20                      Console.WriteLine("水瓶座");
21                  break;
22              case 2:
23                  if(d<=18)
24                      Console.WriteLine("水瓶座");
25                  else
26                      Console.WriteLine("双鱼座");
27                  break;
28              case 3:
29                  if(d<=20)
30                      Console.WriteLine("双鱼座");
31                  else
32                      Console.WriteLine("白羊座");
33                  break;
34              case 4:
35                  if(d<=19)
36                      Console.WriteLine("白羊座");
37                  else
38                      Console.WriteLine("金牛座");
39                  break;
40              case 5:
41                  if(d<=20)
42                      Console.WriteLine("金牛座");
43                  else
44                      Console.WriteLine("双子座");
45                  break;
46              case 6:
```

```
47                  if(d<=21)
48                      Console.WriteLine("双子座");
49                  else
50                      Console.WriteLine("巨蟹座");
51                  break;
52              case 7:
53                  if(d<=22)
54                      Console.WriteLine("巨蟹座");
55                  else
56                      Console.WriteLine("狮子座");
57                  break;
58              case 8:
59                  if(d<=22)
60                      Console.WriteLine("狮子座");
61                  else
62                      Console.WriteLine("处女座");
63                  break;
64              case 9:
65                  if(d<=22)
66                      Console.WriteLine("处女座");
67                  else
68                      Console.WriteLine("天秤座");
69                  break;
70              case 10:
71                  if(d<=23)
72                      Console.WriteLine("天秤座");
73                  else
74                      Console.WriteLine("天蝎座");
75                  break;
76              case 11:
77                  if(d<=22)
78                      Console.WriteLine("天蝎座");
79                  else
80                      Console.WriteLine("射手座");
81                  break;
82              case 12:
83                  if(d<=21)
84                      Console.WriteLine("射手座");
85                  else
86                      Console.WriteLine("摩羯座");
87                  break;
88          }
```

```
89        }
90    }
```

程序运行情况如下：

```
3↙
15↙
双鱼座
```

3.5 程序循环结构

3.5.1 while 语句

while 循环是一种简单的循环，其语句形式如下：

```
While(表达式)
    语句
```

圆括号内的表达式称为循环条件，语句称为循环体。while 语句的执行流程是先计算作为循环条件的 bool 型表达式的值，如果表达式的值为 false，则程序执行跳转到 while 循环结尾之后继续；如果表达式的值为 true 则执行语句，并且该表达式在语句每次执行结束后都再次被求值。每次表达式的值为 true 时，语句都会被执行一次。循环在表达式结果为 false 时结束。其执行流程可用图 3.3(a)来描述。

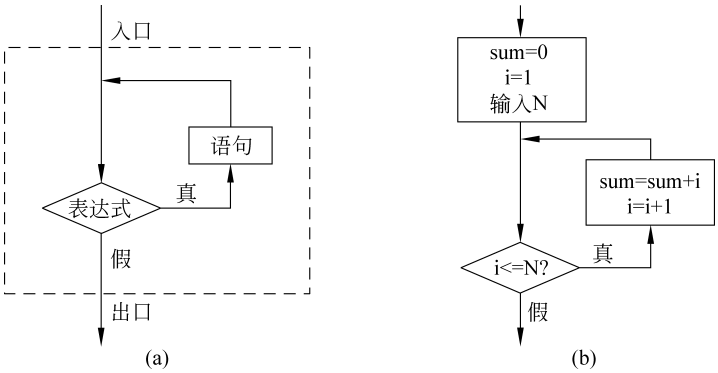


图 3.3　while 语句执行流程

【例 3.4】 计算从 1 累加到 N 的和，其中，N 为大于 1 的自然数。

分析：用流程图表示算法如图 3.3(b)所示。

```
1    using System;
2    class Sum
3    {
4        static void Main()
5        {
6            int N, i=1, sum=0;
```

```
7           N=Convert.ToInt32(Console.ReadLine());    //从键盘接收 N
8           while(i<=N)                                //循环直到 i 大于 N
9           {
10              sum=sum+i;                             //累加
11              i=i+1;
12          }
13          Console.WriteLine("sum is {0}", sum);
14      }
15  }
```

程序运行情况如下：

```
100↙
sum is 5050
```

使用 while 语句时，循环体中应有使 while 表达式值趋向 false 的操作，如例 3.4 中第 11 行的“i=i+1;”，否则表达式值恒为 true，循环将永不结束，称为死循环。有时无意中在 while 条件后书写了额外的分号，将会使得 while 循环体为空语句，从而改变了循环的初始意图，如下所示：

```
while(i<=N);
{
    sum=sum+i;
    i=i+1;
}
```

上面代码将会形成死循环，花括号里的两个赋值语句永远不会被执行。

3.5.2　do 语句

do 语句同样可以实现循环功能，它与 while 语句最大的差别是 do 语句先执行循环体的语句，然后再计算给定的表达式值，根据结果判定是否循环执行。语句形式为

```
do
   语句
While(表达式);
```

do 语句的执行流程是先执行语句，再计算 bool 型表达式的值。如果表达式的值为 true，则语句被再次执行；每次表达式的值为 true 时，语句都会被执行一次。当表达式的值为 false 时，控制流程跳转到循环结构结尾之后的那条语句。其执行流程可用图 3.4 来描述。

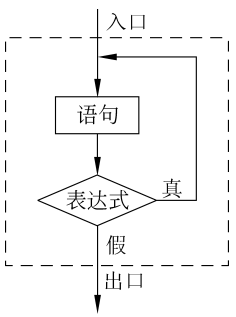


图 3.4　do 语句执行流程

do 语句可以用 while 语句替换。通常情况下，while 语句具有比 do 语句高的使用频率。

【例 3.5】　计算从 1 累加到 N 的和，其中，N 为大于 1 的自然数。用 do 语句实现。