第 3 章

运算符、控制结构与指针

C++语言中集成了大量的运算符，运算符直接作用在数据上，与数据一起构成表达式。C++语言的语句由表达式构成，语句必须以分号结尾。C++语言的最小功能单元为函数，函数由语句组成。程序中语句或函数的组织方式由控制结构决定，程序有 3 种常见的控制结构，即顺序执行、分支执行和循环执行结构。指针是一种重要的数据类型，与第 2 章所介绍的各种数据类型不同的是，指针保存的是变量或函数的地址。本章将详细介绍 C++的运算符、控制结构和指针。

本章的学习目标：

- 了解程序语句的 3 种控制方式
- 熟练掌握两种分支控制和 4 种循环控制方式
- 掌握指针的定义与调用方法
- 学习应用分支控制和循环控制进行程序设计

3.1 运算符

在 C++语言中，运算符与数据结合在一起，构成表达式。在计算表达式的值时，按照运算符的优先级顺序计算，即先计算优先级高的运算符，再计算优先级低的运算符；当运算符的优先级相同时，按照规定的次序（又称结合次序）进行计算，一般按从左向右的顺序，但对于赋值运算符，则按照从右向左的顺序。

当表达式中运算符的优先级和结合性不确定时，可以将需要优先计算的部分用圆括号“()”括起来，此时将优先计算圆括号中的表达式。

C++语言的全部运算符及其优先级如表 3-1 所示。

表 3-1 C++语言的运算符及其优先级

优 先 级	运 算 符	备 注
1	( ),[ ],->,. ,::,!,～,＋＋,－－	后置＋＋和后置－－运算符的优先级高于前置＋＋和前置－－
2	－,＊,&,sizeof	－负号、＊取内容、& 取地址
3	(强制类型转换)	
4	->＊,.＊	
5	＊,/,％	＊乘、/除、％取模
6	＋,－	＋加、－减

续表

优 先 级	运 算 符	备 注
7	<<,>>	
8	<,<=,>,>=	
9	==,!=	
10	&	& 按位与
11	^	
12	|	
13	&&	
14	||	
15	?:	条件运算符,是唯一的三目运算符
16	=,+=,-=,*=,/=,%=,<<=,>>=,&=,^=,|=	赋值或复合赋值运算符
17	,	逗号运算符

下面讨论表 3-1 中常用的运算符的使用方法。

3.1.1 算术运算符

算术运算符包括加、减、乘、除和求余,依次用符号+、-、*、/和%表示。下面在程序段 3-1 中用整数演示了算术运算符的用法。


视频讲解

程序段 3-1 算术运算符的用法

```
1    #include <iostream>
2    #include <cmath>
3    using namespace std;
4
5    int main()
6    {
7      int a1 = 12, a2 = 83, a3 = 4;
8      cout << "a1 + a2 = " << a1 + a2 << endl;
9      cout << "a1 - a2 = " << a1 - a2 << endl;
10     cout << "a1 * a2 = " << a1 * a2 << endl;
11     cout << "a1 / a2 = " << a1 / a2 << endl;
12     cout << "a2 % a1 = " << a2 % a1 << endl;
13     cout << "a1 + a2 / a3 = " << a1 + a2 / a3 << endl;
14     cout << "(a1 + a2) / a3 = " << (a1 + a2) / a3 << endl;
15     cout << "pow(a1,a3) = " << pow(a1, a3) << endl;
16   }
```

在程序段 3-1 中,第 2 行的预编译指令"#include <cmath>"将头文件 cmath 包括到程序中,是因为第 15 行使用其中的函数 pow(),pow()函数用于求乘方运算。

在 main()函数中,第 7 行的语句"int a1=12,a2=83,a3=4;"定义了 3 个整型变量 a1、a2 和 a3,分别赋了初值 12、83 和 4。

第 8 行的语句"cout << "a1+a2=" << a1+a2 << endl;"输出字符串""a1+a2=""和 a1 加 a2 的和。

第 9 行的语句“cout << "a1－a2＝" << a1－a2 << endl;”输出字符串“"a1－a2＝"”和 a1 减去 a2 的差。

第 10 行的语句“cout << "a1 * a2＝" << a1 * a2 << endl;”输出字符串“"a1 * a2＝"”和 a1 乘以 a2 的积。

第 11 行的语句“cout << "a1 / a2＝" << a1 / a2 << endl;”输出字符串“"a1 / a2＝"”和 a1 除以 a2 的商。注意，两个整数的除法运算，其结果仍为整数。

第 12 行的语句“cout << "a2 % a1＝" << a2 % a1 << endl;”输出字符串“"a2 % a1＝"”和 a2 除以 a1 的余数。

第 13 行的语句“cout << "a1＋a2 / a3＝" << a1＋a2 / a3 << endl;”输出字符串“"a1＋a2 / a3＝"”以及 a2 除以 a3 的商加上 a1 的和。这里，先算除法再算加法。

第 14 行的语句“cout << "(a1＋a2) / a3＝" << (a1＋a2) / a3 << endl;”输出字符串“"(a1＋a2) / a3＝"”和(a1＋a2) / a3 的值。这时先算括号内的加法，再计算除法运算。

第 15 行的语句“cout << "pow(a1,a3)＝" << pow(a1,a3) << endl;”调用系统函数(或称库函数)pow 计算 $a1^{a3}$ 的值，并输出在计算机显示器上。函数 pow 为求乘方运算。系统函数 pow 位于头文件 cmath 中，故需要在第 2 行将头文件 cmath 包括到程序中。

算术运算符＋、－、*、/均可以应用于整数和浮点数；但是求余运算%只能应用于整数，而不能应用于浮点数。由于现有的计算机均为 64 位机器，并且 Windows 操作系统也为 64 位系统，因此，建议使用长整型 long long 和双精度浮点型 double 的数据进行算法运算，可避免数据类型的隐式转换，从而提高运算速度。如果使用了其他的类型，那么在计算中含有长整型的表达式将统一转化为 64 位进行运算，含有双精度浮点型的表达式统一转化为 64 位进行运算，运算结果再转换为所需要的数据类型存储。

程序段 3-1 的执行结果如图 3-1 所示。

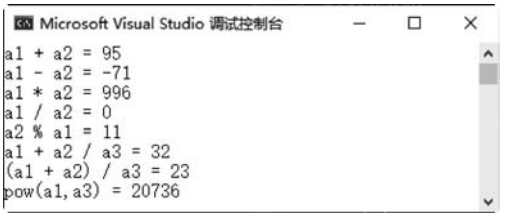

图 3-1　程序段 3-1 的执行结果

3.1.2　关系运算符

关系运算符连接的式子称为关系表达式，关系表达式的返回值为逻辑值(或称布尔值)。关系运算符包括大于、小于、等于、大于或等于、小于或等于和不等于，对应的算符依次为＞、＜、＝＝、＞＝、＜＝和!＝。注意，关系运算符的等于为“＝＝”(两个等号，中间无空格)，而“＝”在 C++语言中为赋值运算符。

关系运算符中，“大于”“小于”“大于或等于”“小于或等于”的优先级相同，其优先级比“等于”和“不等于”的优先级高，而“等于”和“不等于”的优先级相同。

下面程序段 3-2 详细说明各个关系运算符的用法。


视频讲解

程序段 3-2　关系运算符用法实例

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6      int a1 = 3, a2 = 5;
7      cout << "a1 = " << a1 << endl;
8      cout << "a2 = " << a2 << endl;
9      bool b1, b2, b3, b4, b5, b6;
10     b1 = a1 > a2;
11     b2 = a1 < a2;
12     b3 = a1 >= a2;
13     b4 = a1 <= a2;
14     b5 = a1 == a2;
15     b6 = a1 != a2;
16     if (b1)
17         cout << "a1 > a2" << endl;
18     if (b2)
19         cout << "a1 < a2" << endl;
20     if (b3)
21         cout << "a1 >= a2" << endl;
22     if (b4)
23         cout << "a1 <= a2" << endl;
24     if (b5)
25         cout << "a1 == a2" << endl;
26     if (b6)
27         cout << "a1!= a2" << endl;
28   }
```

在程序段 3-2 的 main 函数中，第 6 行的语句“int a1＝3，a2＝5；”定义了两个整型变量 a1 和 a2，分别赋初值为 3 和 5。

第 7 行的语句“cout << "a1＝" << a1 << endl；”输出字符串"a1＝"和 a1 的值。

第 8 行的语句“cout << "a2＝" << a2 << endl；”输出字符串"a2＝"和 a2 的值。

第 9 行的语句“bool b1，b2，b3，b4，b5，b6；”定义 6 个逻辑型变量 b1、b2、b3、b4、b5 和 b6。

第 10 行的语句“b1＝a1 ＞ a2；”将关系表达式“a1 ＞ a2”的逻辑值赋给变量 b1。

第 11 行的语句“b2＝a1 ＜ a2；”将关系表达式“a1 ＜ a2”的逻辑值赋给变量 b2。

第 12 行的语句“b3＝a1 ＞＝a2；”将关系表达式“a1 ＞＝a2”的逻辑值赋给变量 b3。

第 13 行的语句“b4＝a1 ＜＝a2；”将关系表达式“a1 ＜＝a2”的逻辑值赋给变量 b4。

第 14 行的语句“b5＝a1 ＝＝a2；”将关系表达式“a1 ＝＝a2”的逻辑值赋给变量 b5。

第 15 行的语句“b6＝a1 ！＝a2；”将关系表达式“a1 ！＝a2”的逻辑值赋给变量 b6。

第 16 行和第 17 行为一个 if 结构，第 16 行判断 b1 的值是否为真，如果为真，则执行第 17 行，输出字符串“a1＞a2”。

第 18 行和第 19 行为一个 if 结构，第 18 行判断 b2 的值是否为真，如果为真，则执行第 19 行，输出字符串“a1＜a2”。

第 20 行和第 21 行为一个 if 结构，第 20 行判断 b3 的值是否为真，如果为真，则执行第 21 行，输出字符串“a1＞＝a2”。

第 22 行和第 23 行为一个 if 结构，第 22 行判断 b4 的值是否为真，如果为真，则执行第 23 行，输出字符串“a1＜＝a2”。

第 24 行和第 25 行为一个 if 结构，第 24 行判断 b5 的值是否为真，如果为真，则执行第 25 行，输出字符串“a1＝＝a2”。

第 26 行和第 27 行为一个 if 结构，第 26 行判断 b6 的值是否为真，如果为真，则执行第 27 行，输出字符串“a1!＝a2”。

程序段 3-2 的执行结果如图 3-2 所示。

图 3-2　程序段 3-2 的执行结果

3.1.3　逻辑运算符

逻辑运算符只能对逻辑值 true 或 false 进行运算，由于关系表达式的值为逻辑值，所以，逻辑运算符可以连接关系表达式。逻辑运算符包括逻辑与 &&、逻辑或||和逻辑非!。其中，逻辑非的优先级最高，逻辑与的优先级次之，逻辑或的优先级最低。

当逻辑与连接多个关系表达式时，例如，(关系表达式 1) && (关系表达式 2) && (关系表达式 3) && (关系表达式 4)，若“关系表达式 1”的值为假，则后续的关系表达式不再计算，整个表达式直接返回逻辑值假。当逻辑或连接多个关系表达式时，例如，(关系表达式 1) || (关系表达式 2) || (关系表达式 3) || (关系表达式 4)，若“关系表达式 1”的值为真，则后续的关系表达式不再计算，整个表达式直接返回逻辑值真。

下面的程序段 3-3 介绍了逻辑运算符的用法。

程序段 3-3　逻辑运算符的用法


视频讲解

```
#include <iostream>
using namespace std;

int main()
{
  bool b1 = false, b2 = false, b3 = true, b4 = true;
  bool b5, b6, b7;
  b5 = !(b1 && b2) || !(b3 && b4);
  b6 = !b1 || !b2;
  b7 = (12 > 7) && (10 > 3);
  if (b5)
      cout << "b5 = true" << endl;
  if (b6)
      cout << "b6 = true" << endl;
  if (b7)
```

```
16          cout << "b7 = true" << endl;
17    }
```

在程序段 3-3 的 main()函数中，第 6 行的语句“bool b1＝false，b2＝false，b3＝true，b4＝true；”定义了 4 个逻辑型变量 b1、b2、b3 和 b4，分别赋初值为 false、false、true、true。

第 7 行的语句“bool b5，b6，b7；”定义了 3 个逻辑型变量 b5、b6 和 b7。

第 8 行的语句“b5＝！（b1 && b2）|| ！（b3 && b4）；”计算 b1 和 b2 的逻辑与，然后，对这个结果取逻辑非，如果为真，则将逻辑真赋给 b5；如果“！（b1 && b2）”的逻辑值为假，则计算“！（b3 && b4）”的逻辑值，即将 b3 与 b4 进行逻辑与操作，再对其值取逻辑非。如果“！（b3 && b4）”的逻辑值为真，则将逻辑真赋给 b5；否则，将逻辑假赋给 b5。这里根据 b1～b4 的逻辑值，可知 b5 为真。

第 9 行的语句“b6＝！ b1 || ！ b2；”先计算 b1 的逻辑非，如果结果为真，则将逻辑真赋给 b6；否则，计算 b2 的逻辑非，如果结果为真，将逻辑真赋给 b6，否则，将逻辑假赋给 b6。这里根据 b1 和 b2 的逻辑值，可知 b6 为真。

第 10 行的语句“b7＝(12＞7) && (10＞3)；”先计算关系表达式“(12＞7)”的值，如果该结果为假，则将逻辑假赋给 b7；这里，关系表达式“(12＞7)”的值为真，因此，需要继续计算关系表达式“(10＞3)”的值，由于“(10＞3)”的值为真，故整个表达式的返回值为真，即将逻辑真赋给 b7。

第 11 行和第 12 行的语句为一个 if 结构，第 11 行判断 b5 是否为真，如果为真，则执行第 12 行，输出“b5＝true”。

第 13 行和第 14 行的语句为一个 if 结构，第 13 行判断 b6 是否为真，如果为真，则执行第 14 行，输出“b6＝true”。

第 15 行和第 16 行的语句为一个 if 结构，第 15 行判断 b7 是否为真，如果为真，则执行第 16 行，输出“b7＝true”。

程序段 3-3 的执行结果如图 3-3 所示。

图 3-3　程序段 3-3 的执行结果

3.1.4　位运算符

支持位运算是 C++语言的特色，这使 C++语言广泛应用于微控制器的程序设计中。位运算符包括按位取反～、按位与 &、按位或 |、按位异或 ^、按位左移 << 和按位右移 >>。例如，设一个 16 比特的无符号整型变量 a 的值为 0x7755，即 0111 0111 0101 0101B，执行下面的位运算：

(1) ～a

将 a 的各位取值，原来的 0 变为 1，原来的 1 变为 0，则得到 0x88aa，即 1000 1000 1010 1010B。

(2) a & 0xFF

这里与0xFF相与，即保留a的低8位不变，其高8位清0，得到0x0055，即0000 0000 0101 0101B。

(3) a | 0xFF

这里与0xFF相或，即保留a的高8位不变，其低8位全部置1，得到0x77FF，即0111 0111 1111 1111B。

(4) a ^0xFF

将a与0xFF相异或，即保留a的高8位不变，其低8位中的0变为1、1变为0，得到0x77AA，即0111 0111 1010 1010B。

(5) a >> 2

将a向右移动2位，相当于a/4，得到0x1DD5，即0001 1101 1101 0101B。

(6) a << 2

将a向左移动2位，相当于a乘以4，得到0xDD54，即1101 1101 0101 0100B。

上述位运算可通过下面的程序段3-4验证。

程序段 3-4　位运算符的用法示例


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    #define Int16U unsigned short
5
6    int main()
7    {
8      Int16U a = 0x7755;
9      Int16U b1, b2, b3, b4, b5, b6;
10     b1 = ~a;
11     b2 = a & 0xFF;
12     b3 = a | 0xFF;
13     b4 = a ^ 0xFF;
14     b5 = a >> 2;
15     b6 = a << 2;
16     cout << hex << "b1 = 0x" << b1 << endl;
17     cout << "b2 = 0x" << b2 << endl;
18     cout << "b3 = 0x" << b3 << endl;
19     cout << "b4 = 0x" << b4 << endl;
20     cout << "b5 = 0x" << b5 << endl;
21     cout << "b6 = 0x" << b6 << endl;
22   }
```

在程序段3-4中，第4行的宏定义指令“#define Int16U unsigned short”将数据类型unsigned short宏定义为Int16U，在程序中出现Int16U的地方，在编译时自动替换为unsigned short。宏定义指令也是一种预编译指令，在程序的编译阶段进行解析，常用宏定义指令定义一些常量，例如“#define　PI　3.14159”，这样程序中可以使用PI，相当于将PI视为常量。

在main()函数中，第8行的语句“Int16U a=0x7755;”定义无符号短整型a，并赋初值

0x7755，这里的 0x7755 表示这是一个十六进制数的形式。

第 9 行的语句“Int16U b1，b2，b3，b4，b5，b6；”定义 6 个无符号短整型变量 b1、b2、b3、b4、b5 和 b6。

第 10 行的语句“b1＝～a；”将 a 按位取反，取反后的值赋给 b1。

第 11 行的语句“b2＝a & 0xFF；”将 a 与 0xFF 取与，即保留 a 的低 8 位不变，将高 8 位清 0，之后的结果赋给 b2。

第 12 行的语句“b3＝a | 0xFF；”将 a 与 0xFF 取或，即保留 a 的高 8 位不变，将其低 8 位全部置为 1，之后的结果赋给 b3。

第 13 行的语句“b4＝a ^0xFF；”将 a 与 0xFF 取异或，即保留 a 的高 8 位不变，将其低 8 位中的 0 变成 1、1 变成 0，之后的结果赋给 b4。

第 14 行的语句“b5＝a >> 2；”将 a 右移 2 位后的结果赋给 b5。

第 15 行的语句“b6＝a << 2；”将 a 左移 2 位后的结果赋给 b6。

第 16 行的语句“cout << hex << "b1＝0x" << b1 << endl；”中，“hex”表示以十六进制形式输出数值，这里输出字符串“b1＝0x”和 b1 的十六进制值。“hex”是一个全局作用符，其后的所有数值输出都将采用十六进制形式，若需要改为十进制形式输出，则需要指示符“dec”。

第 17～21 行以十六进制形式输出 b2～b6 的值。

程序段 3-4 的执行结果如图 3-4 所示。

图 3-4　程序段 3-4 的执行结果

3.1.5　自增自减运算符

自增自减运算符为单目运算符，只有一个操作数。自增运算符表示为“＋＋”（中间无空格），对于整型变量 i 而言，“i＋＋”或“＋＋i”都相当于 i＝i＋1。自减运算符表示为“－－”（中间无空格），对于整型变量 i 而言，“i－－”或“－－i”都相当于 i＝i－1。

如果将包含自增运算符或自减运算符的表达式用于赋值时，操作数位于自增自减运算符的前面和后面，其表达式的计算结果不相同。例如，对于整型变量 i＝5 和 j 而言，执行表达式“j＝i＋＋”时，先取出 i 的值赋给 j，然后，i 自增 1，即执行后，i 的值为 6，而 j 的值为 5。同样地，对于整型变量 i＝5 和 j 而言，执行表达式“j＝＋＋i”时，先将 i 的值自增 1，然后，将 i 的值赋给 j，即执行后，i 和 j 的值都为 6。对于自减运算符是同样的道理。

下面的程序段 3-5 详细说明了自增和自减运算符的用法。


视频讲解

程序段 3-5　自增与自减运算符的用法

```
1    #include <iostream>
2    using namespace std;
3
4    typedef unsigned int Int32U;
```

```
5
6    int main()
7    {
8      Int32U i, j, k, u, v;
9      i = 10;
10     cout << "i = " << i << endl;
11     j = i++;
12     cout << "i = " << i << ", j = " << j << endl;
13     k = ++i;
14     cout << "i = " << i << ", k = " << k << endl;
15     u = --i;
16     cout << "i = " << i << ", u = " << u << endl;
17     v = i--;
18     cout << "i = " << i << ", v = " << v << endl;
19   }
```

在程序段 3-5 中，第 4 行的语句“typedef unsigned int Int32U;”借助于 typedef 关键字自定义变量类型，这里将已有的变量类型 unsigned int 自定义为一个新的变量类型 Int32U，这种自定义变量类型的作用类似于程序段 3-4 中的宏定义方式，可以把较长的变量类型缩短为新的变量类型名称，但是，这里的“typedef unsigned int Int32U;”是一条 C++语句，而不是预编译指令。自定义后的新类型 Int32U 和 unsigned int 完全相同。

在 main()函数中，第 8 行的语句“Int32U i,j,k,u,v;”定义了 5 个无符号整型变量 i、j、k、u、v。

第 9 行的代码“i=10;”将 i 赋为 10。

第 10 行的代码“cout << "i=" << i << endl;”输出字符串“i=”和 i 的值。

第 11 行的语句“j=i++;”先将 i 的值赋给 j，然后，i 再自增 1。

第 12 行的语句“cout << "i=" << i << ",j=" << j << endl;”输出字符串"i="、i 的值和字符串"j="、j 的值。

第 13 行的语句“k=++i;”先将 i 自增 1，然后，将 i 的值赋给 k。

第 14 行的语句“cout << "i=" << i << ",k=" << k << endl;” 输出字符串"i="、i 的值和字符串"k="、k 的值。

第 15 行的语句“u=--i;”先将 i 的值自减 1，然后，将 i 的值赋给 u。

第 16 行的语句“cout << "i=" << i << ",u=" << u << endl;” 输出字符串"i="、i 的值和字符串"u="、u 的值。

第 17 行的语句“v=i--;”先将 i 的值赋给 v，然后，先将 i 的值自减 1。

第 18 行的语句“cout << "i=" << i << ",v=" << v << endl;” 输出字符串"i="、i 的值和字符串"v="、v 的值。

程序段 3-5 的执行结果如图 3-5 所示。

图 3-5　程序段 3-5 的执行结果

3.1.6 赋值运算符与 sizeof 运算符

赋值运算符形式为“=”,其将赋值运算符右边的表达式的值赋给左边的变量。赋值运算符和算术运算符以及位运算符可以合并为复合赋值运算符,即+=、-=、*=、/=、%=、<<=、>>=、&=、|=、^=,复合运算符中的两个运算符间不能添加空格。

复合赋值运算符主要是为了书写方便,例如,对于两个整型变量 a 和 b,表达式 a+=b 等价于 a=a+b,b 也可以为一个表达式。同样地,a-=b 等价于 a=a-b;a*=b 等价于 a=a*b;a/=b 等价于 a=a/b;a%=b 等价于 a=a%b;a<<=b 等价于 a=a<<b;a>>=b 等价于 a=a>>b;a&=b 等价于 a=a&b;a|=b 等价于 a=a|b;a^=b 等价于 a=a^b。

sizeof 运算符用于统计变量或数据类型所占的存储空间大小,以字节为单位。例如,sizeof(int)返回 int 类型的长度,为 4(个字节);若有双精度浮点型变量 d,则 sizeof(d)返回 d 在存储器中的长度,即 8(个字节)。

下面的程序段 3-6 说明了赋值运算符和 sizeof 运算符的用法。


视频讲解

程序段 3-6 赋值运算符和 sizeof 运算符的用法

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6      int a = 0, b = 10, c = 11, d = 12, e = 13;
7      a += (b -= (c *= (d /= (e %= 5))));
8      cout << "a = " << a << endl;
9      unsigned short u = 0xFFFF, v = 0x6033, w = 0x3AA, x = 0x3C, y = 5;
10     u &= (v |= (w ^= (x <<= (y >>= 1))));
11     cout << hex << "u = 0x" << u << dec << endl;
12     double score[10] = {0.1,0.2};
13     int n = sizeof(score);
14     cout << "Size of double:" << sizeof(double) << ", n = " << n << endl;
15     struct Student
16     {
17         char name[20];
18         char gender;
19         double score[10];
20     }st[10];
21     cout << "Size of st[0]:" << sizeof(st[0]) <<
22         ", Size of st:" << sizeof(st) << endl;
23   }
```

在程序段 3-6 的 main()函数中,第 6 行的语句“int a=0,b=10,c=11,d=12,e=13;”定义了 5 个整型变量 a、b、c、d、e,并分别赋初值 0、10、11、12、13。

第 7 行的语句“a+=(b-=(c*=(d/=(e%=5))));”先算最里面括号的表达式,再依次向外层计算,最后得到 a 的值为-34。

第 8 行的语句“cout << "a=" << a << endl;”输出字符串“a=”和 a 的值。

第 9 行的语句“unsigned short u＝0xFFFF，v＝0x6033，w＝0x3AA，x＝0x3C，y＝5；”定义无符号短整型 u、v、w、x 和 y，并分别赋值为 0xFFFF、0x6033、0x3AA、0x3C、5。

第 10 行的语句“u &＝(v |＝(w ^＝(x <<＝(y >>＝1))))；”先计算最里层括号的表达式，再依次向外层计算，最后得到 u 的值为 0x637B。

第 11 行的语句“cout << hex << "u＝0x" << u << dec << endl；”以十六进制形式输出 u 的值，然后恢复十进制输出格式。

第 12 行的语句“double score[10]＝{0.1，0.2}；”定义双精度浮点型数组 score，包含 10 个元素，前两个元素分别初始化为 0.1 和 0.2，其余元素初始化为 0。

第 13 行的语句“int n＝sizeof(score)；”使用 sizeof 运算符得到 score 数组的长度（即在存储器中占有的字节数），将其赋给变量 n。

第 14 行的语句“cout << "Size of double：" << sizeof(double) << "，n＝" << n << endl；”输出 double 类型的长度和数组 score 的长度（即 n 的大小）（均为字节为单位）。

第 15～20 行声明了结构体类型 Student，并定义了 Student 类型的数组 st，长度为 10。

第 21 行和第 22 行为一条语句，输出了结构体数组的一个元素 st[0]的长度以及结构体数组 st 的长度。这里，需要注意，直观上 Student 类型占 101 个字节（因为 char 占 1 个字节、double 占 8 个字节），但是存储 Student 类型的字段 name 和 gender 将使用 21 个字节，这里，存储后续的 score 将从第 22 个字节开始存储，然后，score 是 double 型（占 8 字节），它的存储首地址的低 3 位必须为 0，所以，将空出 3 个字节，从第 25 个字节开始存储。所以，一个 Student 结构体将占据 104 个字节的空间。为不失一般性，设 name[0]的首地址为 0x1000 0000 0000，则 gender 的存储地址将为 0x1000 0000 0014，此时，score 的存储首地址必须为 0x1000 0000 0018。故第 21 行和第 22 行输出的 st[0]的长度为 104 字节，而 st 的长度为 1040 字节。

程序段 3-6 的执行结果如图 3-6 所示。

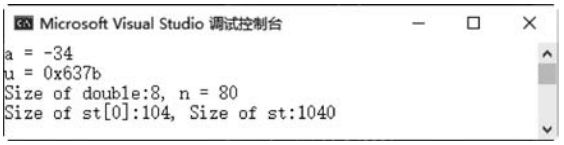

图 3-6　程序段 3-6 的执行结果

3.1.7　条件运算符

大部分运算符具有两个操作数，称为双目运算符；一少部分运算符具有一个操作数，称为单目运算符；C++语言中存在一个唯一个三目运算符，具有 3 个操作数，称为条件运算符，用符号“？：”表示。

设条件运算符连接操作数得到的表达式为“(关系表达式或逻辑表达式)？(表达式 1)：(表达式 2)”，则其执行的操作为：首先判断“关系表达式或逻辑表达式”的值，如果该值为真，则计算“表达式 1”并返回其结果；否则，计算“表达式 2”并返回其结果。

下面程序段 3-7 借助于条件运算符实现下面的分段线性映射：

$$F(x)=\begin{cases}\dfrac{x}{p}, & x\in[0,p]\\ \dfrac{x-p}{0.5-p}, & x\in(p,0.5]\\ F(1-x,p), & x\in(0.5,1]\end{cases}$$


视频讲解

程序段 3-7　条件运算符的用法

```
1    #include <iostream>
2    #include <iomanip>
3    using namespace std;
4
5    int main()
6    {
7      double x, p;
8      cout << "Input x(0 < x < 1,x!= 0.5):";
9      cin >> x;
10     cout << "Input p(0 < p < 0.5):";
11     cin >> p;
12     double y;
13     x = (x > 0.5) ? 1 - x : x;
14     y = (x < p) ? x / p : (x - p) / (0.5 - p);
15     cout << "y = " << setprecision(3)<< y << endl;
16   }
```

在程序段 3-7 中，第 2 行的预编译指令“#include <iomanip>”将头文件 iomanip 包括在程序中，这是因为程序中使用了库函数 setprecision()，其中的“setprecision(3)”表示设定小数位数为 3 位。

在 main()函数中，第 7 行的语句“double x,p;”定义了双精度浮点型变量 x 和 p。

第 8 行的语句“cout << "Input x(0＜x＜1,x! ＝0.5)：";”输出提示信息“Input x(0＜x＜1,x! ＝0.5):”，这里要求输入 x，x 的值为 0～1 间，且不能等于 0.5。

第 9 行的语句“cin >> x;”输入变量 x 的值。

第 10 行的语句“cout << "Input p(0＜p＜0.5)：";”输出提示信息“Input p(0＜p＜0.5):”，这里要求输入 p，且要求 p 的值为 0～0.5。

第 11 行的语句“cin >> p;”输入变量 p 的值。

第 12 行的语句“double y;”定义双精度浮点型变量 y。

第 13 行的语句“x＝(x ＞ 0.5) ? 1－x : x;”使用条件运算符，首先判断 x 的值是否大于 0.5，如果是，则返回 1－x 的值(覆盖原来的 x)；如果 x 的值不大于 0.5，则返回 x 的值。

第 14 行的语句“y＝(x ＜ p) ? x / p : (x－p) / (0.5－p);”使用条件运算符，首先判断 x 的值是否大于 p，如果是，则返回 x/p 的值给 y；如果 x 的值不大于 p，则返回(x－p)/(0.5－p)的值给 y。

第 15 行的语句“cout << "y＝" << setprecision(3)<< y << endl;”输出字符串"y＝"和 y 的值，其中，y 的值保留 3 位小数。

程序段 3-7 的运行结果如图 3-7 所示。

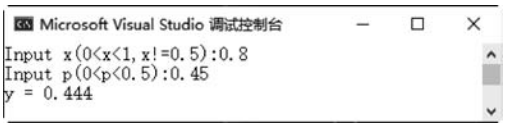

图 3-7　程序段 3-7 的运算结果

3.1.8　逗号运算符

逗号运算符是优先级最低的运算符，比赋值运算符还低一级。在定义多个变量时，例如“int　a,b,c;”中出现的逗号，并非逗号运算符，这里的逗号只起到分隔符的作用。在可执行语句中出现的逗号，则是逗号运算符。逗号运算符可以出现多个，例如，“a=(3*5,5*6,7*8,8*9);”中，出现了 3 个逗号运算符，按照从左向右的次序依次计算各个表达式，最后一个表达式的值为整个括号中的逗号表达式的值，即这里的 a 的值为 72。

程序段 3-8 演示了逗号运算符的用法。

程序段 3-8　逗号运算符的用法


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6      int a, b = 3;
7      a = (b * 4, 6 * 9);
8      cout << "a = " << a << endl;
9      a = b * 4, 6 * 9;
10     cout << "a = " << a << endl;
11     a = b * 4, b = 6 * 9;
12     cout << "a = " << a << ", b = " << b << endl;
13   }
```

在程序段 3-8 的 main()函数中，第 6 行的语句“int a,b=3;”定义了变量 a 和 b，并将 b 赋初值 3。

第 7 行的语句“a=(b * 4,6 * 9);”中使用了逗号运算符，由于使用括号，先算括号里面的，逗号运算符的结合性为自左向右，即先计算“b * 4”，再计算“6 * 9”，并将后者的结果作为逗号运算符连接的表达式的值，故 a 的值为 54。

第 8 行的语句“cout << "a=" << a << endl;”输出字符串"a="和 a 的值。

第 9 行的语句“a=b * 4,6 * 9;”使用了逗号运算符，由于逗号运算符的优先级比赋值运算符低，所以，先计算“a=b * 4”，再计算“6 * 9”，即得到 a 为 12(b 的值为 4)，而“6 * 9”的结果没有保存。

第 10 行的语句“cout << "a=" << a << endl;”输出字符串"a="和 a 的值。

第 11 行的语句“a=b * 4,b=6 * 9;”使用了逗号表达式，由于逗号运算符的优先级比赋值运算符低，所以，先计算“a=b * 4”，再计算“b=6 * 9”，得到 a 的值为 12，b 的值为 54。

第 12 行的语句“cout << "a=" << a << ",b=" << b << endl;”输出字符串"a="和 a 的

值以及字符串"b="和 b 的值。

程序段 3-8 的执行情况如图 3-8 所示。

图 3-8　程序段 3-8 的执行结果

3.2　分支控制

程序的执行方式只有 3 种，即顺序执行、分支执行和循环执行。顺序执行是总的执行方式，是指语句按照先后顺序依次执行。循环执行是指一组语句在特定的条件下反复执行，直到该条件不再满足为止。而分支执行表示在某种情况下将执行这一组语句，而另一种情况下，将执行另一组语句。本节将介绍 C++语言的分支控制方法，主要有两种控制结构，即 if-else 结构和 switch-case 结构。

3.2.1　if-else 结构

if-else 结构具有以下 3 种情况：

(1) 若条件 cond 为真，则执行一条语句或一组语句

```
if(cond)
    一条语句;
```

或者

```
if(cond)
{
    一组语句;
}
```

在上述情况下，若 cond 为真，则执行“一条语句”或“一组语句”，“一组语句”可以为空、空语句、一条语句或多条语句，每条语句均以分号“;”结尾。这里的“空”表示没有语句；“空语句”表示只有分号“;”的语句。在这种情况下，当只有“一条语句”时，可以省略花括号。

这种情况如程序段 3-9 和程序段 3-10 所示。


视频讲解

程序段 3-9　if-else 结构的第一种情况(if 结构中只有一条语句)

```
1    int x = 3;
2    if (x > 0)
3      cout << "x is positive." << endl;
```

在程序段 3-9 中，第 1 行语句定义了变量 x，并赋初值 3。第 2 行语句使用关键字 if 判断 x>0 是否为真，如果为真，则执行第 3 行，输出“x is positive”；如果为假，无操作。

程序段 3-10　if-else 结构的第一种情况(if 结构中有一组语句)

```
1    int x = 3;
```

```
2    if (x > 0)
3    {
4      cout << x;
5      cout << " is positive." << endl;
6    }
```

在程序段 3-10 中，第 1 行语句定义了变量 x，并赋初值 3。第 2 行语句使用关键字 if 判断 x>0 是否为真，如果为真，则执行第 3～6 行的语句组；如果 x>0 为假，则无操作。

(2) 若条件 cond 为真，则无操作；如果 cond 为假，则执行一条语句或一组语句

```
if(cond)
    ;
else
    一条语句;
```

或者

```
if(cond)
{
}
else
{
    一组语句;
}
```

上述情况的示例如程序段 3-11 和程序段 3-12 所示。

程序段 3-11　if-else 结构的第二种情况(单条语句)


视频讲解

```
1    int x =  -3;
2    if (x > 0)
3      ;
4    else
5      cout << "x is negative." << endl;
```

在程序段 3-11 中，第 1 行定义了变量 x，并赋初值－3。第 2 行判断 x>0 是否为真，如果为真，则执行第 3 行的空语句；如果 x>0 为假，则执行第 5 行，输出“x is negative.”。

程序段 3-12　if-else 结构的第二种情况(多条语句)

```
1    int x =  -3;
2    if (x > 0)
3    {
4    }
5    else
6    {
7      cout << x;
8      cout << " is negative." << endl;
9    }
```

在程序段 3-12 中，第 2 行判断 x>0 是否为真，如果为真，则执行第 3～4 行的空语句；如果 x>0 为假，则执行第 5 行，输出“x is negative.”。

(3) 若条件 cond 为真,则执行一组语句;否则执行另一组语句

```
if(cond)
{
    一组语句
}
else
{
    另一组语句;
}
```

这里的语句组可以为空、空语句、一条语句或多条语句,这是 if-else 结构的标准形式。下面的程序段 3-13 说明了这种情况。

程序段 3-13　if-else 结构的第三种情况

```
1    int x = -3;
2    if (x > 0)
3    {
4      cout << x;
5      cout << " is positive." << endl;
6    }
7    else
8    {
9      cout << x;
10     cout << " is negative." << endl;
11   }
```

在程序段 3-13 中,第 2 行判断 x>0 是否为真,若为真,则执行第 3～6 行的语句组;若 x>0 为假,则执行第 8～11 行的语句组。

上述 3 种 if-else 的情况均只有两种分支情况,if-else 结构可以实现多分支情况,称为 if-else 结构的扩展形式。下面是一种四分支的 if-else 扩展结构:

```
if(条件 1)
{
    语句组 1;
}
else if(条件 2)
{
    语句组 2;
}
else if(条件 3)
{
    语句组 3;
}
else
{
    语句组 4;
}
```

在上述结构中,如果“条件 k”(k=1,2,3)成立,则执行相应的“语句组 k”,否则,当所有

条件都不成立时，执行“语句组 4”。上述结构中可以添加更多的 else if 语句，如

```
else if(条件 n)
{
    语句组 n;
}
```

实现多分支的 if-else 结构。在多分支结构中，若某一个分支判断为真(而得到执行)，则不再判断(和执行)其余的分支了。

下面的程序段 3-14 说明了这种多分支 if-else 结构的用法。

程序段 3-14　多分支 if-else 结构的用法


视频讲解

```
#include <iostream>
using namespace std;

int main()
{
  cout << "Input the income x(x > 0):";
  int x = 0;
  cin >> x;
  double tax;
  if (x > 10000)
  {
      tax = x * 0.3;
      cout << "tax = " << tax << endl;
  }
  else if (x > 8000)
  {
      tax = x * 0.25;
      cout << "tax = " << tax << endl;
  }
  else if (x > 6000)
  {
      tax = x * 0.2;
      cout << "tax = " << tax << endl;
  }
  else if (x > 4000)
  {
      tax = x * 0.15;
      cout << "tax = " << tax << endl;
  }
  else if (x > 2000)
  {
      tax = x * 0.1;
      cout << "tax = " << tax << endl;
  }
  else
  {
      tax = x * 0.05;
      cout << "tax = " << tax << endl;
```

```
39      }
40  }
```

在程序段 3-14 的 main()函数中，第 6 行的语句“cout << "Input the income x(x>0):";”输出提示信息“Input the income x(x>0):”。

第 7 行的语句“int x=0;”定义整型变量 x，并赋初值 0。

第 8 行的语句“cin >> x;”输入变量 x 的值。

第 9 行的语句“double tax;”定义双精度浮点型的变量 tax。

第 10～39 行为一个六分支的 if-else 扩展结构。第 10 行判断 x>1000 是否为真，如果为真，则执行第 11～14 行的语句组，即“tax=x * 0.3; cout << "tax=" << tax << endl;”(x 乘以 0.3 的积赋给 tax，并输出 tax 的值)。当第 10 行判断结果为假时，第 15 行判断 x>8000 是否为真，若为真，则执行第 16～19 行的语句组。若第 15 行判断为假时，第 20 行判断 x>6000 是否为真，若为真，则执行第 21～24 行的语句组。若第 20 行判断为假时，第 25 行判断 x>4000 是否为真，若为真，则执行第 26～29 行的语句组。若第 25 行判断为假时，第 30 行判断 x>2000 是否为真，若为真，则执行第 31～34 行的语句组。若第 30 判断为假，则执行第 36～39 行的语句组。注意，当 6 个分支中某一个分支判断为真时，就不再去判断(和执行)剩余的分支了。

程序段 3-14 的执行结果如图 3-9 所示。

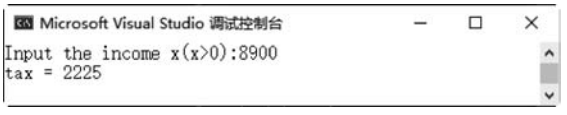

图 3-9　程序段 3-14 的执行结果

此外，if-else 结构可以嵌套使用。if 部分可以嵌套新的 if-else 结构，else 部分也可以嵌套新的 if-else 结构。下面给出了标准形式的二级嵌套的情况，图 3-10 为这一结构的流程图。

```
if(条件 1)
{
    语句组 1;
    if(条件 2)
    {
        语句组 2;
    }
    else
    {
        语句组 3;
    }
    语句组 4;
}
else
{
    语句组 5;
    if(条件 3)
    {
```

```
        语句组 6;
    }
    else
    {
        语句组 7;
    }
    语句组 8;
}
```

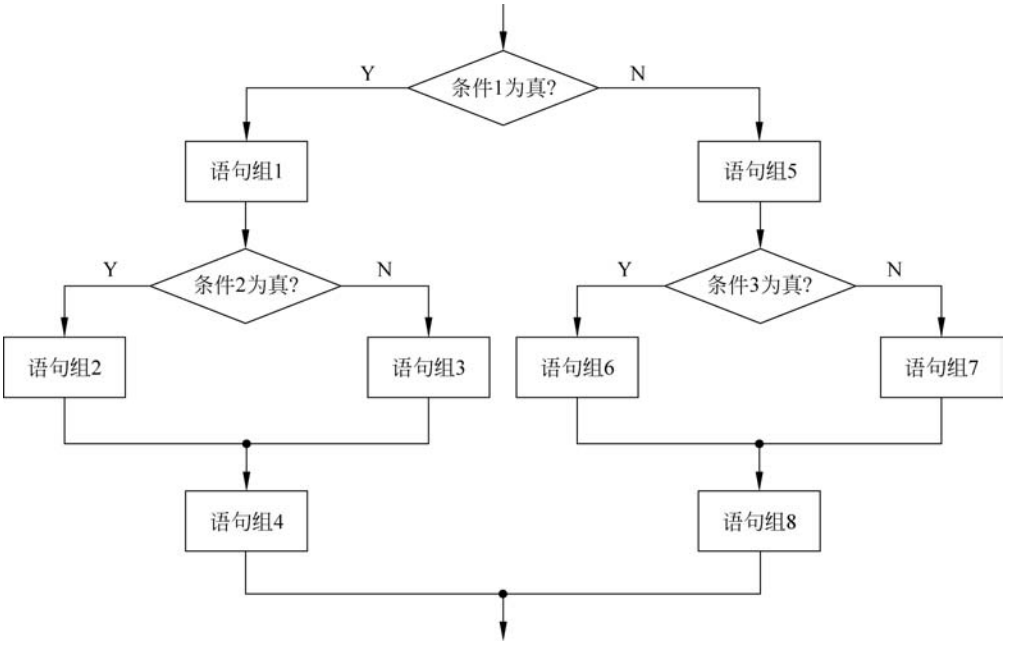

图 3-10　标准二级嵌套的流程图

程序段 3-15 给出了一种 if-else 结构的嵌套用法实例。

程序段 3-15　if-else 结构的一种嵌套用法实例


视频讲解

```
1    #include <iostream>
2    #include <iomanip>
3    using namespace std;
4
5    int main()
6    {
7      double x, p;
8      cout << "Input x(0 < x < 1,x!= 0.5):";
9      cin >> x;
10     cout << "Input p(0 < p < 0.5):";
11     cin >> p;
12     double y;
13     if (x < 0.5)
14     {
15         if (x < p)
16             y = x / p;
```

```
17          else
18              y = (x - p) / (0.5 - p);
19      }
20      else
21      {
22          x = 1 - x;
23          if (x < p)
24              y = x / p;
25          else
26              y = (x - p) / (0.5 - p);
27      }
28      cout << "y = " << setprecision(3) << y << endl;
29  }
```

程序段 3-15 实现的功能与程序段 3-7 完全相同。第 2 行的预编译指令“#include <iomanip>”将头文件 iomanip 包括在程序中。

在 main()函数中,第 7 行的语句“double x,p;”定义了双精度浮点型变量 x 和 p。

第 8 行的语句“cout << "Input x(0<x<1,x!=0.5): ";”输出提示信息“Input x(0<x<1,x!=0.5):”,这里要求输入 x,x 的值为 0～1,且不能等于 0.5。

第 9 行的语句“cin >> x;”输入变量 x 的值。

第 10 行的语句“cout << "Input p(0<p<0.5): ";”输出提示信息“Input p(0<p<0.5):”,这里要求输入 p,且要求 p 的值为 0～0.5。

第 11 行的语句“cin >> p;”输入变量 p 的值。

第 12 行的语句“double y;”定义双精度浮点型变量 y。

第 13～27 行为二级嵌套的 if-else 结构。第 13 行的语句“if (x < 0.5)”判断 x<0.5 是否为真,如果为真,则执行第 14～19 行的语句组;否则,执行第 21～27 行的语句组。

第 15～18 行为内层的 if-else 结构,第 15 行的语句“if (x < p)”判断 x<p 是否为真,如果为真,则执行第 16 行的语句“y=x / p;”;否则,执行第 18 行的语句“y=(x-p) / (0.5-p);”。

第 23～26 行的 if-else 结构与第 15～18 行的 if-else 结构完全相同。

第 28 行的语句“cout << "y=" << setprecision(3) << y << endl;”输出字符串“y=”和 y 的值,其中,y 的值保留 3 位小数。

程序段 3-15 的执行结果如图 3-11 所示。

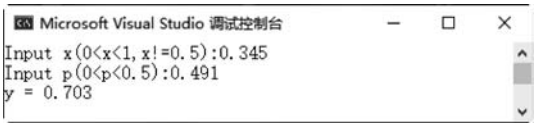

图 3-11 程序段 3-15 的执行结果

3.2.2 switch-case 结构

switch-case 结构是典型的多分支结构,其形式如下:

```
switch(表达式)
```

```
{
    case 值 1:
            语句组 1;
            break;
    case 值 2:
            语句组 2;
            break;
    case 值 3:
            语句组 3;
            break;
    …
    case 值 n:
          语句组 n;
          break;
    default:
          语句组 n + 1;
}
```

这里，“表达式”的值必须为整型或枚举类型，当“表达式”的值与某一个“值 k”(k 为 1～n 的值)时，将执行相应的“语句组 k”，然后执行“break;”语句跳出 switch-case 结构；如果“表达式”的值与所有的“值 k”(k 为 1～n)均不同，则执行“default:”部分的“语句组 n+1”。注意，如果某个 case 部分的“break;”被省略，那么当执行完该 case 部分的“语句组”后将不会跳转，而是顺序执行下面的 case 部分。每个“语句组”均可以为空、空语句、一条语句或多条语句。default 部分可以省略。

下面的程序段 3-16 说明了 switch-case 结构的用法。

程序段 3-16　switch-case 结构用法实例


视频讲解

```
1    #include <iostream>
2    #include <iomanip>
3    using namespace std;
4
5    int main()
6    {
7      cout << "Input the income x(x >= 0):";
8      double x;
9      cin >> x;
10     int y;
11     y = int(x / 1000);
12     double tax;
13     switch (y)
14     {
15     case 1:
16     case 2:
17     case 3:
18         tax = x * 0.05;
19         break;
20     case 4:
21     case 5:
22         tax = x * 0.1;
```

```
23          break;
24      case 6:
25      case 7:
26          tax = x * 0.15;
27          break;
28      case 8:
29      case 9:
30      case 10:
31          tax = x * 0.2;
32          break;
33      default:
34          tax = x * 0.3;
35      }
36      cout << "tax = " << fixed << setprecision(2) << tax << endl;
37  }
```

在程序段 3-16 的 main()函数中，第 7 行的语句“cout << "Input the income x(x>=0):";”输出提示信息“Input the income x(x>=0):”，即要求输入一个非负实数。

第 8 行的语句“double x;”定义双精度浮点型变量 x。

第 9 行的语句“cin >> x;”输入变量 x 的值。

第 10 行的语句“int y;”定义整型变量 y。

第 11 行的语句“y=int(x / 1000);”使用强制类型转换将 x/1000 的值转换为整数赋给变量 y。

第 12 行的语句“double tax;”定义双精度浮点型变量 tax。

第 13～35 行为 switch-case 结构，第 13 行判断 y 的值，当 y 为 1、2 或 3 时(第 15～17 行)，执行第 18 行“tax=x * 0.05;”得到 tax 的值，然后执行第 19 行“break;”跳出 switch-case 结构；当 y 为 4 或 5 时(第 20～21 行)，执行第 22～23 行，然后跳出 switch-case 结构；当 y 为 6 或 7 时(第 24～25 行)，执行第 26～27 行，然后跳出 switch-case 结构；当 y 为 8、9 或 10 时(第 28～30 行)，执行第 31～32 行，然后跳出 switch-case 结构；如果上述 case 情况都不成立，则执行第 33～34 行，然后，跳出 switch-case 结构。

第 36 行的语句“cout << "tax=" << fixed << setprecision(2) << tax << endl;”输出字符串"tax="和 tax 的值。这里的 fixed 为数据输出格式控制符，表示不使用科学计数法输出；setprecision(2)表示保留 2 位小数。

程序段 3-16 的输出结果如图 3-12 所示。

图 3-12　程序段 3-16 的执行结果

3.3　循环控制

循环控制是高效率编写程序的主要手段。例如，实现 1+2+…+100 的简单运算，借助于循环控制将变成一件轻松的工作。循环控制主要有 4 种结构，即 for 结构、while 结构、

do-while 结构和 foreach 结构。其中，foreach 结构是一种专门用来遍历数组（或序列）元素的循环结构，其余 3 种结构都是通用循环结构。

3.3.1　for 结构

for 结构的基本形式为：

```
for(设定循环变量初始值; 循环条件; 调整循环变量)
{
    循环执行的语句组;
}
```

在上述 for 结构中，“设定循环变量初始值”部分用于给循环变量赋初始值，这部分在整个 for 循环中只被执行一次，所以，也可以把一些初始化工作放在此处；“循环条件”是每次执行“循环执行的语句组”前，都要判断的条件，如果“循环条件”为真，则执行这一次循环操作，如果“循环条件”为假，则跳出循环体；“调整循环变量”部分从第二次循环开始，每次都要执行一次“调整循环变量”。

由此可见，for 循环的执行过程为：

（1）执行“设定循环变量初始值”，即给循环控制变量赋初值；

（2）判断“循环条件”是否为真，若为真，则执行第（3）步；否则，跳出循环体，执行第（5）步；

（3）执行“循环执行的语句组”；

（4）执行“调整循环变量”，回到第（2）步；

（5）执行循环体后续的语句。

for 结构有如下注意事项：

（1）有可能 for 循环中“循环执行的语句组”一次也得不到执行。当循环变量的初始值不满足“循环条件”时，将直接跳过循环体而执行后续的语句。

（2）for 循环头部“设定循环变量初始值”“循环条件”“调整循环变量”均可以为空，或部分为空，但是分号不能少。“for(;;)”这种形式表示无限循环（俗称死循环），对于无限循环，需要在其中设定跳出循环的语句。

（3）语句“break;”可以跳出它所在的循环。当有循环嵌套时，语句“break;”仅能跳出它所在的那一层循环，执行循环体后续的语句。

（4）语句“continue;”用于结束本次循次，回到循环头部继续下一次循环，即“continue;”语句后续的循环体部分不再被执行，而是回到循环开始处执行下一次循环，对于 for 结构而言，回到“调整循环变量”处执行。

（5）若 for 结构的循环体只有一条语句，可以省略花括号。

下面的程序段 3-17 给出了计算 1＋2＋…＋100 的几种 for 结构。

程序段 3-17　for 结构的用法实例


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
```

```
5    {
6      int i, sum;
7      sum = 0;
8      for (i = 1; i <= 100; i++)
9      {
10        sum += i;
11     }
12     cout << "1 + 2 + ... + 100 = " << sum << endl;
13
14     for (sum = 0, i = 1; i <= 100; sum += i, i++)
15     {
16     }
17     cout << "1 + 2 + ... + 100 = " << sum << endl;
18
19     i = 1;
20     sum = 0;
21     for (; i <= 100;)
22     {
23         sum += i;
24         i++;
25     }
26     cout << "1 + 2 + ... + 100 = " << sum << endl;
27
28     i = 1;
29     sum = 0;
30     for (;;)
31     {
32         sum += i;
33         i++;
34         if (i > 100)
35             break;
36     }
37     cout << "1 + 2 + ... + 100 = " << sum << endl;
38   }
```

程序段 3-17 列举了 for 结构的 4 种工作方式。在 main 函数中，第 6 行的语句“int i, sum;”定义了整型变量 i 和 sum。

第 7 行的语句“sum＝0;”将 0 赋给变量 sum。

第 8～11 行为一个 for 结构，表示 i 从 1 按步长 1 累加到 100，对于每个 i，循环执行一次第 10 行的语句“sum ＋＝i;”，即将 sum 与 i 的和赋给 sum。

第 12 行的语句“cout << "1＋2＋...＋100＝" << sum << endl;”输出字符串“1＋2＋...＋100＝”和 sum 的值。

第 14～16 行为一个 for 结构，其中，循环体为空，在 for 结构的初始化部分执行“sum＝0, i＝1”，在 for 结构的调整循环变量部分执行“sum ＋＝i, i＋＋”，这样实现了 i 从 1 按步长 1 累加到 100，对每个 i，循环执行“sum ＋＝i”的目的。

第 17 行输出字符串“1＋2＋...＋100＝”和 sum 的值。

第 19 行的语句“i＝1;”将 1 赋给 i。

第 20 行的语句“sum＝0;”将 0 赋给 sum。

第 21～25 行为一个 for 结构，其中，只有“终止循环条件”部分“i ＜＝100”，表示当 i 小于等于 100 时，继续执行循环体。第 19～20 行为该 for 结构作了初始化工作，第 23 行的语句“sum ＋＝i;”为每次循环执行的操作，第 24 行的语句“i＋＋;”相当于该 for 结构的“调整循环变量部分”。

第 26 行输出字符串“1＋2＋...＋100＝”和 sum 的值。

第 28 行的语句“i＝1;”将 1 赋给 i。

第 29 行的语句“sum＝0;”将 0 赋给 sum。

第 30～36 行为一个 for 结构，该 for 结构为一个无限循环结构。每次循环执行第 31～36 行。其中第 32 行的语句“sum ＋＝i;”将 sum 与 i 的和赋给 sum；第 33 行的语句“i++;”将 i 加 1 的值赋给 i；第 34～35 行为循环终止语句，当 i 大于 100 时，执行语句“break;”跳出该无限循环体。

第 37 行输出字符串"1＋2＋...＋100＝"和 sum 的值。

程序段 3-17 的执行结果如图 3-13 所示。

图 3-13　程序段 3-17 的执行结果

for 结构可以多级嵌套使用，并且 for 结构中可以嵌套使用其他的循环结构。

3.3.2　while 结构

类似于程序段 3-17 的第 21～25 行的这种 for 结构，当条件满足时，执行循环体；当条件不满足时，跳出循环体，while 结构的语法如下：

```
while(条件)
{
    语句组;
}
```

在上述 while 结构中，当“条件”为真时，循环执行“语句组”；当“条件”为假时，跳出 while 结构，继续执行后续的语句。

while 结构有时称为“当型”循环结构，即当“条件”为真时，循环执行“语句组”，且每次循环前都需要判断“条件”的真假。一旦发现“条件”为假，则跳出循环。

程序段 3-18 说明了 while 结构的用法。

程序段 3-18　while 结构的用法实例


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
```

```
6       int i = 1, sum = 0;
7       while (i <= 100)
8       {
9           sum += i;
10          i++;
11      }
12      cout << "1 + 2 + ... + 100 = " << sum << endl;
13  }
```

在程序段 3-18 的 main()函数中，第 6 行的语句“int i＝1，sum＝0；”定义整型变量 i 和 sum，并分别初始化为 1 和 0。

第 7～11 行为 while 结构，第 7 行判定 i<＝100 是否为真，若为真，则执行第 8～11 行的语句组；若为假，跳转到第 12 行执行。第 9 行的语句“sum ＋＝i；”将 sum 与 i 的和赋给 sum，第 10 行的语句“i＋＋；”将 i 自增 1。这个 while 循环体执行了 1＋2＋…＋100 的操作。

第 12 行的语句“cout << "1＋2＋...＋100＝" << sum << endl；”输出字符串"1＋2＋...＋100＝"和 sum 的值。

程序段 3-18 的执行结果如图 3-14 所示。

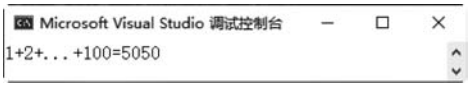

图 3-14　程序段 3-18 的执行结果

3.3.3　do-while 结构

在 while 结构中，有可能循环体一次也得不到执行。在 do-while 结构中，循环体先执行一次，再判断循环条件是否满足，如果满足，则继续执行循环体；如果不满意，则跳出循环体。do-while 结构如下所示：

```
do
{
    语句组;
}while(条件);
```

在 do-while 结构中，先执行“语句组”一次，然后再判断“条件”是否为真，如果为真，则循环执行“语句组”，直到“条件”为假后，跳出循环体。do-while 循环也称“直到型”循环，即循环执行“语句组”，直接“条件”为假。与 while 循环不同的是，do-while 循环至少会执行一次“语句组”。

程序段 3-19 说明了 do-while 结构的用法。


视频讲解

程序段 3-19　do-while 结构的用法实例

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
```

```
6       int i = 1, sum = 0;
7       do
8       {
9           sum += i;
10          i++;
11      } while (i <= 100);
12      cout << "1 + 2 + ... + 100 = " << sum << endl;
13  }
```

在程序段 3-19 的 main()函数中，第 6 行的语句“int i＝1，sum＝0；”定义整型变量 i 和 sum，并分别初始化为 1 和 0。

第 7～11 行为一个 do-while 结构，循环执行第 9～10 行直到“i＜＝100”为假。

第 12 行输出字符串"1＋2＋...＋100＝"和 sum 的值。

程序段 3-19 的执行情况与程序段 3-18 相同，执行结果也如图 3-14 所示。

do-while 结构、while 结构和 for 结构均可互相嵌套使用。

3.3.4　foreach 结构

foreach 结构用于遍历数组这类数据序列的元素，典型结构如下所示：

```
for(数组元素类型 变量名: 数组名)
{
    语句组;
}
```

在上述 foreach 结构中，“变量名”对应的变量将依次取遍“数组名”表示的数组的各个元素（按索引号递增顺序），可以在“语句组”中使用“变量名”对应的变量。

程序段 3-20 介绍了 foreach 结构的用法。

程序段 3-20　foreach 结构的用法实例


视频讲解

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6     int a[10] = { 1,2,3,4,5,6,7,8,9,10 };
7     int sum = 0;
8     for (int e : a)
9     {
10        sum += e * e;
11    }
12    cout << "Square sum of 1 to 10 is: " << sum << endl;
13  }
```

在程序段 3-20 的 main()函数中，第 6 行的语句“int a[10]＝{ 1,2,3,4,5,6,7,8,9,10 };”定义了一维整型数组 a，具有 10 个元素，并用列表{1,2,3,4,5,6,7,8,9,10}中的元素依次初始化每个元素。

第 7 行的语句“int sum＝0；”定义整型变量 sum，并赋初值 0。

第 8～11 行为一个 foreach 结构，对于数组 a 的每个元素(用 e 表示)，循环执行第 10 行语句“sum ＋＝e ＊ e;”将每个元素的平方值加到 sum 上。这个 foreach 结构实现了 1～10 的平方和计算。

第 12 行的语句“cout << "Square sum of 1 to 10 is: " << sum << endl;”输出字符串 "Square sum of 1 to 10 is: "和 sum 的值。

程序段 3-20 的执行结果如图 3-15 所示。

图 3-15　程序段 3-20 的执行结果

程序段 3-20 的第 8 行，常写作“for (auto e : a)”，这里的 auto 类型根据变量的值自动变换为相应的类型，例如，“auto x＝3;”，auto 相当于 int。在“for (auto e : a)”中，auto 将根据 a 的类型设定 e 的类型。

在使用 foreach 结构时，必须使用 foreach 结构专用的局部变量，例如，下面这种方式是错误的：

```
int e;
for(e : a)
{
    语句组;
}
```

3.4　指针

指针是一种特殊的数据类型，可以借助于指针直接访问物理存储器，指针的值为存储器的地址。指针是 C++语言优于其他高级语言的主要体现。计算机操作系统或硬件设备的驱动程序，都需要直接访问硬件接口，因此这类软件(或程序)只能借助 C++语言实现。

3.4.1　常量、变量与指针

在 C++语言中，数值、字符和字符串，例如，3、4.5e2、8.07、'd'、"Hello"等均为常量。而下述的语句定义的均为变量，例如：

```
int x = 0;
double y;
char c;
```

这里的 x、y 和 c 为变量，是因为 x、y 和 c 的值可以改变。x、y 和 c 称为变量名，本质上是这些变量在存储器中的地址。通过取地址符“&”可以得到这些变量的地址值，例如，“&x”将得到变量 x 的地址值，“&y”将得到变量 y 的地址值，“&c”将得到变量 c 的地址值。

可以像定义变量一样定义常量，例如：

```
const int k = 100;
const double pi = 3.14159;
```

```
const char ch = 'Z';
```

上述3条语句通过添加const限定符将k、pi和ch定义为常量，定义常量时必须给这些量赋初值，因为后续不能再改变这些量的值，即后续无法再对这些量进行赋值操作。

指针用于保存变量的地址，定义指针的方式为：

```
变量类型声明符 * 指针名 = & 变量;
```

例如：

```
int a = 30;
int * p = &a;
int b;
b = *p;
```

上述代码定义了整型变量a，并赋初始值30；然后，定义了指向整型变量的指针p，指向变量a；接着，定义了整型变量b；最后，将*p赋给变量b。这里的“*”表示取指针指向的地址中的值，“*p”表示取p指向的地址中的值，这里，*p为30。于是，语句“*p=10;”将10赋给指针p指向的地址，这里赋值后，a的值也为10。

一般地，在定义指针时给指针赋初始地址。一个指针在被赋值前，称为悬浮指针，这类指针不能使用。例如：

```
int * q;
*q = 20;
```

上述语句中“int * q;”定义了指向整数类型的指针q，有时简称整型指针；“*q=20;”向q指向的地址赋值20。由于q没有指向任何地址，故q为悬浮指针，所以“*q=20;”是错误的。为了避免出现悬浮指针，一般在定义指针时给指针初始化。空指针用NULL表示。

下面为正确的指针用法：

```
int * q = NULL;
int d;
q = &d;
*q = 20;
```

在上述语句中，“int * q=NULL;”定义整型指针q为空指针；“int d;”定义整型变量d；“q=&d;”使指针q指向整型变量d，即指针q的值为变量d的地址；“*q=20;”将20赋给指针q指向的地址，赋值后变量d的值也为20。

程序段3-21说明了指针变量与变量地址的关系。

程序段3-21　指针的定义与用法实例


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6      int a = 30;
7      int * p = &a;
```

```
8       cout << "a = " << a << endl;
9       cout << "Address of a: " << &a << endl;
10      cout << "Value of p:" << p << endl;
11      cout << " * p = " << * p << endl;
12       * p = 100;
13      cout << " * p = " << * p << endl;
14      cout << "a = " << a << endl;
15   }
```

在程序段 3-21 的 main()函数中，第 6 行的语句“int a＝30;”定义整型变量 a，并赋初值 30。

第 7 行的语句“int * p＝&a;”定义指向整型变量的指针 p，p 指向变量 a 的地址。

第 8 行的语句“cout << "a＝" << a << endl;”输出字符串"a＝"和 a 的值。

第 9 行的语句“cout << "Address of a：" << &a << endl;”输出字符串"Address of a："和变量 a 的地址。

第 10 行的语句“cout << "Value of p：" << p << endl;”输出字符串"Value of p："和 p 的值。

第 11 行的语句“cout << " * p＝" << * p << endl;”输出字符串" * p＝"和 * p 的值。

第 12 行的语句“ * p＝100;”将 100 赋给指针 p 指向的地址。

第 13 行的语句“cout << " * p＝" << * p << endl;”输出字符串" * p＝"和 * p 的值。

第 14 行的语句“cout << "a＝" << a << endl;”输出字符串"a＝"和 a 的值。

程序段 3-21 的执行结果如图 3-16 所示。

图 3-16　程序段 3-21 的执行结果

由图 3-16 可知，地址的长度为 64 位，即指针的存储长度为 8 个字节，可以在程序段 3-21 中添加一条语句“cout << sizeof(p) << endl;”显示指针 p 的存储长度。

3.4.2　动态数组

在对指针进行赋值时，一般不使用固定的地址，而是将变量的地址赋给指针。然而，在微控制器编程时，将会出现大量使用地址直接赋给指针的用法，例如，“int * p＝(int *) 0x0E0000000000”，表示指针 p 的值为 0x0E0000000000，即指针 p 指向地址 0x0E0000000000。这种直接将地址赋给指针的方式在面向计算机的 C++语言中一般不用。

有时只想定义一个指针，并为指针开辟一块存储空间，而不想定义变量。这时需要用到动态内存分配技术，相关的两个操作符为 new 和 delete。new 用于开辟存储空间，delete 用于释放 new 开辟的空间，以避免空间浪费而造成内存的碎片化。

程序段 3-22 说明了 new 和 delete 的用法。

程序段 3-22　动态内存配置的用法实例


视频讲解

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int * p = new int;
  *p = 10;
  cout << "*p = " << *p << endl;
  delete p;

  int * q = new int(100);
  cout << "*q = " << *q << endl;
  delete q;

  int * w = new int[10];
  for (int i = 0; i < 10; i++)
       *(w + i) = i + 1;
  for (int i = 0; i < 10; i++)
      cout << *(w + i) << " ";
  cout << endl;
  for (int i = 0; i < 10; i++)
      cout << w[i] << " ";
  cout << endl;
  delete[ ] w;

  int * v = new int[10]{ 11,12,13,14,15,16,17,18,19,20 };
  for (int i = 0; i < 10; i++)
      cout << *(v + i) << " ";
  cout << endl;
  delete[ ] v;

  int( *u)[3] = new int[2][3]{ {9,7,5},{10,12,14} };
  for (int i = 0; i < 2; i++)
  {
      for (int j = 0; j < 3; j++)
      {
          cout << left << setw(8) << u[i][j];
      }
      cout << endl;
  }
  int * s = u[0];
  for (int i = 0; i < 2; i++)
  {
      for (int j = 0; j < 3; j++)
      {
          cout << left << setw(8) << *(s + 3 * i + j);
      }
      cout << endl;
```

```
50      }
51      delete[ ] u;
52   }
```

在程序段 3-22 的 main()函数中,第 7 行的语句“int * p=new int;”使用 new 开辟一个整型存储空间(即 4 个字节的存储空间),并将该空间的地址赋给新定义的指针 p。这里的“new int”为动态开辟一个整型存储空间,与第 10 行的语句“delete p;”相对应,“delete p;”用于释放 new 开辟的空间。

第 8 行的语句“ * p=10;”将 10 赋给指针 p 指向的空间。

第 9 行的语句“cout << " * p=" << * p << endl;”输出字符串" * p="和 * p 的值。

第 12 行的语句“int * q=new int(100);”使用 new 开辟一个整型存储空间(即 4 个字节的存储空间),并赋初值 100,然后,将新定义的指针 q 指向该存储空间。这一行与第 14 行对应,第 14 行的语句“delete q;”用于释放该存储空间。

第 13 行的语句“cout << " * q=" << * q << endl;”输出字符串" * q="和 * q 的值。

上述为使用 new 开辟一个存储空间的方法。下面为使用 new 开辟一块连续存储空间的方法。

第 16 的语句“int * w=new int[10];”使用 new 开辟一个长度为 10 个整数的连续的存储空间,使新定义的指针 w 指向该空间的首地址。这里的 w 相当于指向一个长度为 10 的整型数组,* w 或 *(w+0)表示第 1 个元素,*(w+1)表示第 2 个元,以此类推,*(w+9)表示第 10 个元素。事实上,指针与一维数组近似等价,也可以用 w[0]表示第 1 个元素,w[1]表示第 2 个元素,以此类推,w[9]表示第 10 个元素。

第 16 行与第 25 行对应,第 25 行的语句“delete[] w;”释放 new 开辟的空间。

第 17 行和第 18 行为一个 for 结构,向 w 指向的各个元素赋值,这里使用 *(w+i)表示第 i+1 个元素。

第 19 行和第 20 行为一个 for 结构,将 w 指向的各个元素输出,这里使用 *(w+i)表示第 i+1 个元素。

第 21 行的语句“cout << endl;”输出一个回车换行符。

第 22 行和第 23 行为一个 for 结构,将 w 指向的各个元素输出,这里使用 w[i]表示第 i+1 个元素。

第 24 行的语句“cout << endl;”输出一个回车换行符。

第 27 行的语句“int * v=new int[10]{ 11,12,13,14,15,16,17,18,19,20 };”使用 new 开辟一个可存储 10 个整数的连续的存储空间,并使用列表“{11,12,13,14,15,16,17,18,19,20}”对其赋初值,将新定义的指针 v 指向该存储空间的首地址。

第 27 行与第 31 行对应,第 31 行的语句“delete[] v;”释放 v 指向的连续空间。

第 28 行和第 29 行为一个 for 结构,输出指针 v 指向的空间中的所有元素,这里使用 *(v+i)表示第 i+1 个元素。

第 30 行的语句“cout << endl;”输出一个回车换行。

第 33 行的语句“int(* u)[3]=new int[2][3]{ {9,7,5},{10,12,14} };”使用 new 开辟一个二维的整型存储空间,即 2 行 3 列的存储空间,以整型数据为存储单位,并用列表“{ {9,7,5},{10,12,14} }”对该存储空间初始化。二维空间与二维数组类似,将其每行视

为一个一维数组，对于 2 行 3 列的存储空间可视为两个一维数组，每个一维数组有 3 个元素，因此，需要使用“int(*u)[3]”这种形式定义指向这个二维存储空间的指针。不建议使用 new 开辟二维及二维以上的空间，参考第 42～50 行的解释。这里访问二维空间的元素的方法为：使用 u[i][j]或*(*(u+i)+j)表示第 i+1 行和第 j+1 列的元素，即指向二维存储空间的指针与二维数组是近似等价的。

注意：定义“int(*u)[3]”和定义“int *u[3]”完全不同；后者“int *u[3]”表示数组 u 的每个元素为“int *”类型，因此，可以称之为指针的数组。前者“int(*u)[3]”是一种指向二维数组的指针，且数组的第二维必须有 3 个元素。

第 34～41 行一个二级嵌套的 for 结构，输出指针 u 指向的二维空间的各个元素，先输出第一行元素，再输出一个回车换行符，接着，输出第二行元素。

第 42 行的语句“int* s=u[0];”定义一个指针 s，指向 u 指针指向的二维空间的首地址。这里的“u[0]”是 u 指针指向的二维空间的第一行(即一维行向量)的首地址。由于二维空间中各个元素的地址是顺序排列的，可以使用指针按一维向量的形式访问，因此，不建议定义指向二维或二维以上的指针。

第 43～50 为二级嵌套的 for 结构，借助于指针 s 依次输出 u 指针指向的二维空间的各个元素，这里第 47 行的“*(s+3*i+j)”表示第 i+1 行和第 j+1 列的元素。

第 51 行的语句“delete[] u;”释放 u 指针指向的空间。

程序段 3-22 的执行结果如图 3-17 所示。

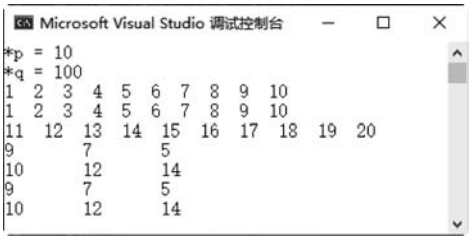

图 3-17　程序段 3-22 的执行结果

由程序段 3-22 可知，指针支持加上(或减去)一个整数的运算，例如，设：

```
int* v = new int[10]{ 11,12,13,14,15,16,17,18,19,20 };
```

则 v 指向 new 开辟的空间的首地址，即指针“11”所在的地址，不妨设为 0x2000 0000 0000；则 v+1 指向“12”所在的地址，即 0x2000 0000 0004，v+1 是 v 所指的地址加上 4 个字节；v+2 指向“13”所在的地址，即 0x2000 0000 0008；v+3 指向“14”所在的地址，即 0x2000 0000 000C；以此类推，v+9 指向“20”所在的地址，即 0x2000 0000 0000+9*4。指针的步进 1 相当于地址步进 4 个字节，这是因为指针指向的变量类型(int)占 4 个字节。

3.4.3　数组与指针

一维数组和指针近似等价，例如，定义如下一维数组

```
inta[10] = { 3,5,7,9,11,13,15,17,19,21 };
```

可以使用类似指针的方法访问数组元素，例如，*(a+3)将访问数组的第 4 个元素，即元素

“9”。这说明，对于一维数组而言，数组名就是数组的首地址，也可视为指向数组首地址的指针。因此对于上述的一维数组“int a[10]={ 3,5,7,9,11,13,15,17,19,21 };”，定义指向该数组的指针时，使用如下语句：

```
int * p = a;
```

或

```
int * p = &a[0];
```

上述指针 p 和 a 的唯一区别在于，p 是变量，可以指向新的地址，或者说可以赋给它新的地址，而 a 是数组名，a 是固定的地址，不能改变 a 的值，可以视 a 为一个地址常量。

对于二维数组或二维以上的高维数组而言，和一维数组的情况稍有不同。这里以三维数组为例进行讨论。设有如下三维数组：

```
int b[2][3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24 };
```

这里数组名 b 是数组 b 的首地址，但是数组名 b 本身是一种指向三维数组的指针，而不是一般意义上的指针，下面的 w 指针指向了数组 b：

```
int( * q)[3][4] = b;
```

这里三维数组 b 可以视为 2 个二维数组，而每个二维数组又可以按行划分视为 3 个一维数组，这样，“b、b[0]、b[0][0]、&b、&b[0]、&b[0][0]、&b[0][0][0]”都相同，即都指向同一个地址。但是，只有“&b[0][0][0]”才可以赋给普通意义上的指针；其他的地址均为指向数组的指针形式。

由于数组(包括高维数组)的元素是按顺序存储的，因此，习惯上使用普通指针访问高维数组元素，例如定义：

```
int * w = &b[0][0][0];
```

可以使用 w 指针访问三维数组 b 的各个元素，其中，*(w+i * 12+j * 4+k)即为 b[i][j][k]。

下面的程序段 3-23 说明了数组与指针的关系。


视频讲解

程序段 3-23　借助指针访问数组元素的实例

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
  int a[10] = { 3,5,7,9,11,13,15,17,19,21 };
  for (int i = 0; i < 10; i++)
      cout << left << setw(5) << *(a + i);
   cout << endl;
  int * p = a;
  for (int i = 0; i < 10; i++)
      cout << left << setw(5) << *(p + i);
```

```
14      cout << endl;
15
16      int b[2][3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,
17          14,15,16,17,18,19,20,21,22,23,24 };
18      int( * q)[3][4] = b;
19      cout << b << " " << b[0] << " " << b[0][0] << endl;
20      cout << &b << " " << &b[0] << " " << &b[0][0] << endl;
21      cout << &b[0][0][0] << endl;
22      for (int i = 0; i < 2; i++)
23      {
24          for (int j = 0; j < 3; j++)
25          {
26              for (int k = 0; k < 4; k++)
27              {
28                  cout << left << setw(6) << * ( * ( * (q + i) + j) + k);
29              }
30              cout << endl;
31          }
32          cout << endl;
33      }
34      int * w = &b[0][0][0];
35      for (int i = 0; i < 2; i++)
36      {
37          for (int j = 0; j < 3; j++)
38          {
39              for (int k = 0; k < 4; k++)
40              {
41                  cout << left << setw(6) << * (w + i * 12 + j * 4 + k);
42              }
43              cout << endl;
44          }
45          cout << endl;
46      }
47  }
```

在程序段 3-23 的 main()函数中，第 7 行的语句“int a[10]={ 3,5,7,9,11,13,15,17,19,21 };”定义一维整型数组 a，具有 10 个元素，并使用列表“{ 3,5,7,9,11,13,15,17,19,21 }”初始化数组 a。

第 8 行和第 9 行为一个 for 结构，用“ * (a+i)”这种方式输出数组 a 的各个元素。

第 10 行的语句“cout << endl;”输出一个回车换行。

第 11 行的语句“int * p=a;”定义整型指针 p，指向数组 a 的首地址。

第 12 行和第 13 行为一个 for 结构，用“p[i]”这种方式输出数组 a 的各个元素。

第 14 行的语句“cout << endl;”输出一个回车换行。

第 16 行和第 17 行的语句“int b[2][3][4]={ 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24 };”定义一个三维数组 b 并作了初始化，这种初始化方式中，根据三维数组 b 的元素存储顺序依次实始化各个元素。

第 18 行的语句“int(* q)[3][4]=b;”定义一个指向三维数组的指针 q，并初始化为 b，

即 q 指向三维数组 b 的首地址。

第 19 行的语句“cout << b << " " << b[0] << " " << b[0][0] << endl;”输出 b、b[0]和 b[0][0]的内容，这 3 个值相同，都是三维数组 b 的首地址。

第 20 行的语句“cout << &b << " " << &b[0] << " " << &b[0][0] << endl;”输出&b、&b[0]、&b[0][0]的内容，这 3 个值相同，都是三维数组 b 的首地址。

第 21 行的语句“cout << &b[0][0][0] << endl;”输出 b[0][0][0]的地址，该地址为三维数组 b 的首地址。

第 22～33 行为一个三级嵌套的 for 结构，使用指向三维数组的指针“*(*(*(q+i)+j)+k)”的形式输出三维数组 b 的各个元素，这里的“*(*(*(q+i)+j)+k)”就是 b[i][j][k]。

第 34 行的语句“int * w=&b[0][0][0];”定义整型指针 w，该指针指向三维数组 b 的首元素的地址。

第 35～46 行为一个三级嵌套的 for 结构，这里使用了普通指针“*(w+i * 12+j * 4+k)”的形式输出三维数组 b 的各个元素，这里的“*(w+i * 12+j * 4+k)”就是 b[i][j][k]。

推荐使用第 34～46 行这种方式访问高维数组。

程序段 3-23 的执行结果如图 3-18 所示。

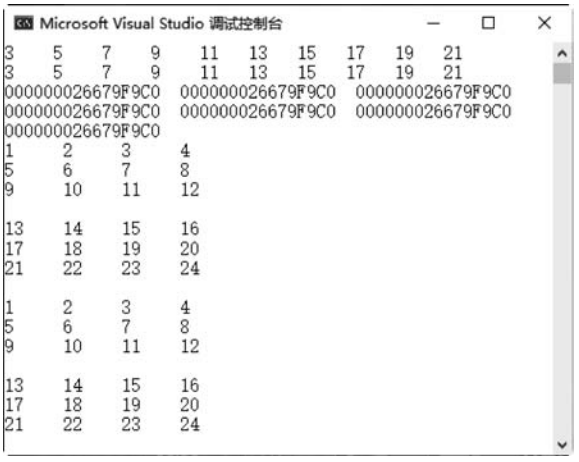

图 3-18　程序段 3-23 的执行结果

3.5　引用

一个变量的引用定义为该变量的别名，引用不占用新的存储空间。引用仅针对单个变量。定义引用时必须初始化，因为引用是作为变量的别名存在的，必须有它的载体变量。引用的两个重要作用在于：

(1) 作为函数的返回值；

(2) 作为函数的形参变量。

由于函数在定义时不会为形参分配空间(在函数调用时为形参指定实参)，故引用作为函数形参时无须初始化。这部分内容将在第 4 章详细介绍。

程序段 3-24 说明了引用的用法。

程序段 3-24　引用的用法实例


视频讲解

```
1    #include <iostream>
2    using namespace std;
3
4    int main()
5    {
6      int a = 5;
7      int& r1 = a;
8      cout << "a = " << a << endl;
9      cout << "r1 = " << r1 << endl;
10     cout << "Addr of a: " << &a << endl;
11     cout << "Addr of r1: " << &r1 << endl;
12
13     int& r2 = r1;
14     cout << "r2 = " << r2 << endl;
15     cout << "Addr of r2: " << &r2 << endl;
16
17     int b[5] = { 3,5,7,9,11 };
18     int& r3 = b[4];
19     cout << "r3 = " << r3 << endl;
20
21     int* p = b;
22     int& r4 = p[3];
23     cout << "r4 = " << r4 << endl;
24   }
```

在程序段 3-24 的 main()函数中，第 6 行的语句“int a=5;”定义整型变量 a，并赋初值 5。

第 7 行的语句“int& r1=a;”定义引用 r1，r1 作为变量 a 的引用。定义引用用符号“&”表示。

第 8 行的语句“cout << "a=" << a << endl;”输出字符串"a="和 a 的值。

第 9 行的语句“cout << "r1=" << r1 << endl;”输出字符串"r1="和 r1 的值。r1 为 a 的引用，r1 的值等于 a 的值。

第 10 行的语句“cout << "Addr of a: " << &a << endl;”输出字符串"Addr of a: "和 a 的地址。

第 11 行的语句“cout << "Addr of r1: " << &r1 << endl;”输出字符串"Addr of r1: "和 r1 的地址，r1 的地址和 a 的地址相同。

第 13 行的语句“int& r2=r1;”定义引用 r2 作为 r1 的引用，即可以定义引用的引用。

第 14 行的语句“cout << "r2=" << r2 << endl;” 输出字符串"r2="和 r2 的值。r2 为 r1 的引用，r1 为 a 的引用，所以，r2 的值等于 a 的值。

第 15 行的语句“cout << "Addr of r2: " << &r2 << endl;”输出字符串"Addr of r2: "和 r2 的地址，r2 的地址和 r1 与 a 的地址相同。

第 17 行的语句“int b[5]={ 3,5,7,9,11 };”定义一维整型数组 b，并对 b 进行初始化。

第 18 行的语句“int& r3=b[4];”定义引用 r3 作为 b[4]的引用，即可以定义对数组元素的引用。

第 19 行的语句“cout << "r3=" << r3 << endl;”输出字符串"r3="和 r3 的值。

第 21 行的语句“int * p=b;”定义指针 p,指向数组 b 的首地址。

第 22 行的语句“int& r4=p[3];”定义引用 r4 作为 p[3]的引用。

第 23 行的语句“cout << "r4=" << r4 << endl;”输出字符串"r4="和 r4 的值。

程序段 3-24 的执行结果如图 3-19 所示。

图 3-19 程序段 3-24 的执行结果

3.6 排序实例

数组中的数据排序是常见的一种操作,常用的排序方法有冒泡法和选择法。

不妨设有整型数组 a[n],数组名为 a,具有 n 个元素{a[0], a[1], …, a[n-1]}。下面依次介绍冒泡法排序和选择法排序。

1. 冒泡法排序

冒泡法排序(按升序排列)的工作原理为:

(1) 令循环变量 i 从 0 按步长 1 递增到 n-2,依次比较 a[i]与 a[i+1],如果 a[i]大于 a[i+1],则将 a[i]与 a[i+1]对换。这一步将数组 a 中最大的元素保存在 a[n-1]中。

(2) 令循环变量 i 从 0 按步长 1 递增到 n-3,依次比较 a[i]与 a[i+1],如果 a[i]大于 a[i+1],则将 a[i]与 a[i+1]对换。这一步将数组 a 中次大的元素保存在 a[n-2]中。

(3) 令 j 从 n-3 按步长 1 递减到 1,令循环变量 i 从 0 按步长 1 递增到 j-1,按照上述方法,依次将 a[0]至 a[j]中最大的元素保存在 a[j]中。

可见,冒泡法排序的每一步均找出数组中没有排序的元素中的值最大的元素,并将所有比该元素小的元素“冒泡”到该元素的左边。

2. 选择法排序

选择法排序(按升序排列)的工作原理为:

(1) 令变量 i 为 0,令变量 k 为 0,令循环变量 j 从 1 按步长 1 递增到 n-1,依次比较 a[k]与 a[j]的值,如果 a[k]大于 a[j],则将 j 赋给 k。循环结束后,如果 k 不为 0,则将 a[k]与 a[0]对换。这一步将数组 a 中最小的值保存在 a[0]中。

(2) 令变量 i 为 1,令变量 k 为 1,令循环变量 j 从 2 按步长 1 递增到 n-1,依次比较 a[k]与 a[j]的值,如果 a[k]大于 a[j],则将 j 赋给 k。循环结束后,如果 k 不为 1,则将 a[k]与 a[1]对换。这一步将数组 a 中次小的值保存在 a[1]中。

(3) 令变量 i 从 2 按步长 1 递增到 n-2,令 k=i,令循环变量 j 从 i+1 按步长 1 递增到 n-1,依次比较 a[k]和 a[j]的值,如果 a[k]大于 a[j],则将 j 赋给 k。每次循环结束后,如果 k 不等于 i,则将 a[k]与 a[i]对换。

可见，选择法排序的每一步都将选出数组中没有排序的元素中的最小元素的下标，将所有比它大的元素保存在该元素的右侧。

程序段 3-25 给出了冒泡法和选择法排序的实现代码。

程序段 3-25　冒泡法和选择法排序实例


视频讲解

```
#include <iostream>
using namespace std;

int main()
{
  int a[10] = { 45,23,90,18,56,71,15,39,61,80 };
  cout << "Original a:" << endl;
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
  cout << endl;

  int n = 10;
  for (int j = n - 1; j > 0; j-- )
  {
    for (int i = 0; i <= j - 1; i++)
    {
      if (a[i] > a[i + 1])
      {
        int t = a[i];
        a[i] = a[i + 1];
        a[i + 1] = t;
      }
    }
  }
  cout << "Sorted a:" << endl;
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
  cout << endl << endl;

  int b[10] = { 45,23,90,18,56,71,15,39,61,80 };
  cout << "Original b:" << endl;
  for (int i = 0; i < 10; i++)
    cout << b[i] << " ";
  cout << endl;

  n = 10;
  int k;
  for (int i = 0; i <= n - 2; i++)
  {
    k = i;
    for (int j = i + 1; j <= n - 1; j++)
    {
      if (b[k] > b[j])
      {
        k = j;
      }
    }
    if (k != i)
```

```
49      {
50        int t = b[i];
51        b[i] = b[k];
52        b[k] = t;
53      }
54    }
55    cout << "Sorted b:" << endl;
56    for (int i = 0; i < 10; i++)
57      cout << b[i] << " ";
58    cout << endl << endl;
59  }
```

在程序段 3-25 的 main()函数中，第 6 行的语句“int a[10] = { 45,23,90,18,56,71,15,39,61,80 };”定义整型一维数组 a，并初始化为列表“{45,23,90,18,56,71,15,39,61,80}”。

第 7 行的语句“cout << "Original a: " << endl;”输出提示信息“Original a:”。

第 8 行和第 9 行为一个 for 结构，输出数组 a 的全部元素。

第 12 行的语句“int n = 10;”定义整型变量 n，并赋初值 10。这里的 n 为数组 a 的长度。

第 13～24 行为一个嵌套的 for 结构，实现了冒泡法排序，将数组 a 的元素按升序排列。

第 25 行的语句“cout << "Sorted a:" << endl;”输出提示信息“Sorted a:”。

第 26 行和第 27 行为一个 for 结构，输出升序排列好的数组 a 的全部元素。

第 30 行的语句定义数组 b，这里数组 b 和原始的数组 a 相同。

第 32 行和第 33 行为一个 for 结构，输出数组 b 的全部元素。

第 37 行的语句“int k;”定义整型变量 k。

第 38～54 行为一个嵌套的 for 结构，实现了选择排序，将数组 b 中的元素按升序排列。

第 55 行的语句“cout << "Sorted b: " << endl;”输出提示信息“Sorted b:”。

第 56 行和第 57 行为一个 for 结构，输出升序排列后的数组 b 的全部元素。

程序 3-25 的执行结果如图 3-20 所示。

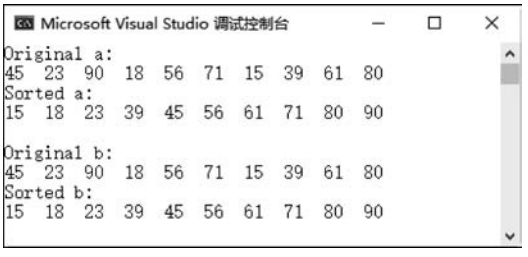

图 3-20 程序段 3-25 的执行结果

3.7 本章小结

本章详细介绍了 C++语言常用的 9 类运算符，即算术运算符、关系运算符、逻辑运算符、位运算符、自增自减运算符、赋值运算符、sizeof 运算符、条件运算符和逗号运算符。C++语言的运算符大多需要连接两个操作数，这类运算符称为双目运算符；少数运算符只需要一

个操作数，称为单目运算符；只有条件运算符是唯一的三目运算符。然后，本章讨论了 C++语言程序的控制结构。只有 3 种程序控制方式，即顺序方式、分支(或称选择)方式和循环方式，这里重点阐述了 if-else 和 switch-case 两种分支结构以及 for、while、do-while、foreach 四种循环结构。接着，讨论了指针的用法和动态创建数组的方法。最后，分析了引用及其用法。

习题

1. 编写程序，输入两个整数及一个双目算术运算符，计算这两个整数经该运算符处理后的结果。

2. 编写程序，输入两个实数，输出这两个实数及其大小关系。

3. 编写程序证明德·摩根定律，即 !(P&&Q)=!P‖!Q；!(P‖Q)=!P&&!Q。

4. 使用位运算符实现 16 位无符号整数 0x389A 的循环右移 5 位和循环左移 12 位。

5. 使用位运算符实现 16 位无符号整数 0xA3D8 的第 3、5、7、9 位置位，第 4、8、12、16 清零，且同时第 1、6、10 位由原来的 0 变为 1 或由原来的 1 变为 0，输出变换后的整数。

6. 使用 if-else 结构编写程序，输入 x 和 p 的值，输出长度为 100 的状态序列，使用的公式为：

$$F(x)=\begin{cases}\dfrac{x}{p}, & x\in[0,p]\\ \dfrac{x-p}{0.5-p}, & x\in(p,0.5]\\ F(1-x,p), & x\in(0.5,1]\end{cases}$$

7. 已知 Fibonacci 序列满足下式：

$$\begin{cases}F_0=0\\ F_1=1\\ F_n=F_{n-1}+F_{n-2}, & n\geqslant 2\end{cases}$$

计算 $n=10、20、30$ 时的 Fibonacci 数(使用 long 整型)。

8. 借助于 for 结构生成高为 5 行的杨辉三角形，即

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

9. 编写程序，分别使用 for 结构、while 结构和 do-while 结构生成九九乘法口诀表。

10. 设有二维数组 a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12}，使用普通指针和指向二维数组的指针实现对数组 a 元素的输出，要求输出为 3 行 4 列的矩阵形式。