chapter5 循环与关系表达式

1）for 循环：

（1）组成部分：

```
for(initialization; test-expression ; update-expression)
{
    body...
}
```

（2）设置步骤：

> [1]设置初始值
> [2]执行测试，考察循环是否可以继续执行
> [3]执行循环
> [4]更新用于测试的值

（3）要点说明：

> [1]test-expression决定循环体是否被执行，这个表达式是关系表达式，值为0的表达式被转化成bool值false，导致循环结束；反之继续！

```
eg: for ( int i = limit ; i ; i-- )   // 当 i 减为 0 时退出循环
```

> [2]update-expression在每轮循环结束时执行，此时循环体已经执行完毕

```
eg: for( int i = 1 ; i <= 100 ; i += 2 )  // 每次 + 2
eg: for( int i = 0 ; i <= 1000 ; i = i*i + 3 )
```

> [3]自增自减表达式：
> x++: 使用x的当前值计算表达式，然后将x的值+1
> ++x: 先将x的值+1，然后用新的值计算表达式
> x--: 同上
> --x: 同上

> [4]前/后缀格式在for循环中的异同：

```
下面二者相同 [reason: 不管是--n还是n--，它们都将在程序进入下一步之前完成，终效果相同]
        for(int n=lim;n;--n){......}    // 前缀的效率更高
```

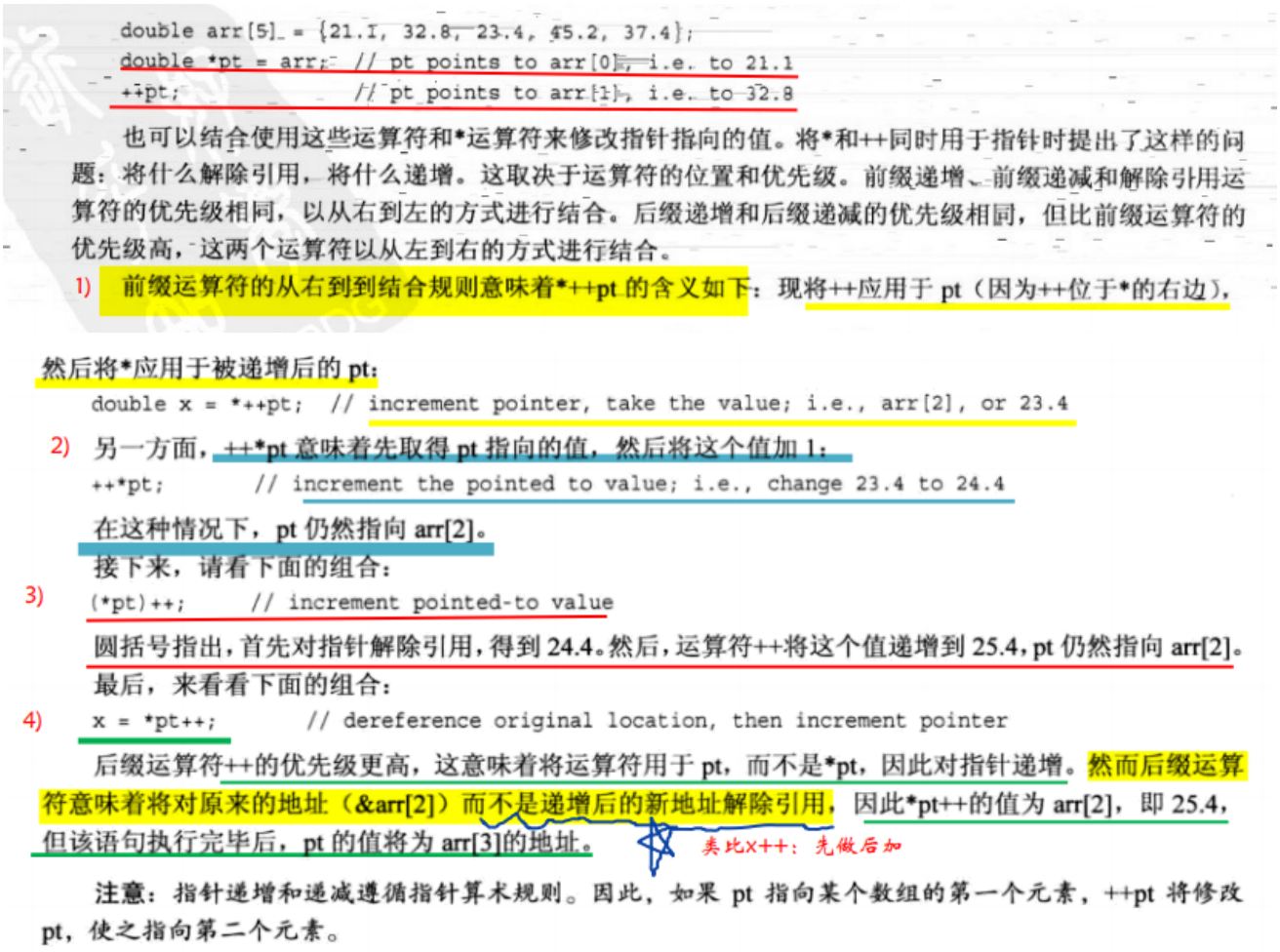
```
for(int n=lim;n;n--){......}     // 后缀的效率相对低
```

（4）工具：

1）递增/递减运算符 与 指针：

[1]前缀 递增/递减 and 解引用 优先级相同，利用自右向左的方式结合

[2]后缀 递增/递减 优先级相同，但比前缀运算符的优先级高，这两个运算符利用自左向右的方式结合



```
double arr[5] = {21.1, 32.8, 23.4, 45.2, 37.4};
double *pt = arr; // pt points to arr[0], i.e. to 21.1
++pt;             // pt points to arr[1], i.e. to 32.8
```

也可以结合使用这些运算符和*运算符来修改指针指向的值。将*和++同时用于指针时提出了这样的问题：将什么解除引用，将什么递增。这取决于运算符的位置和优先级。前缀递增、前缀递减和解除引用运算符的优先级相同，以从右到左的方式进行结合。后缀递增和后缀递减的优先级相同，但比前缀运算符的优先级高，这两个运算符以从左到右的方式进行结合。

1) 前缀运算符的从右到到结合规则意味着*++pt 的含义如下：现将++应用于 pt（因为++位于*的右边），然后将*应用于被递增后的 pt：

```
double x = *++pt;  // increment pointer, take the value; i.e., arr[2], or 23.4
```

2) 另一方面，++*pt 意味着先取得 pt 指向的值，然后将这个值加 1：

```
++*pt;       // increment the pointed to value; i.e., change 23.4 to 24.4
```

在这种情况下，pt 仍然指向 arr[2]。

接下来，请看下面的组合：

3)
```
(*pt)++;     // increment pointed-to value
```

圆括号指出，首先对指针解除引用，得到 24.4。然后，运算符++将这个值递增到 25.4，pt 仍然指向 arr[2]。

最后，来看看下面的组合：

4)
```
x = *pt++;       // dereference original location, then increment pointer
```

后缀运算符++的优先级更高，这意味着将运算符用于 pt，而不是*pt，因此对指针递增。然而后缀运算符意味着将对原来的地址（&arr[2]）而不是递增后的新地址解除引用，因此*pt++的值为 arr[2]，即 25.4，但该语句执行完毕后，pt 的值将为 arr[3]的地址。

类比x++：先做后加

注意：指针递增和递减遵循指针算术规则。因此，如果 pt 指向某个数组的第一个元素，++pt 将修改 pt，使之指向第二个元素。

2）逗号表达式：

[1]常见应用：允许把两个表达式放到“只允许放一个表达式”的地方

```
for(int j = 0,i = word.size() - 1 ; j < i ; --i,++j)
{
        ......
}
```

[2]自身性质：

（1）它先计算第一个表达式，而后计算第二个表达式

```
i = 20 , j = 2 * i;    // i set to 20 , then j ste to 40
```

> （2）逗号表达式的值是第二部分的值
> 比如上述表达式整体的值是40（因为 " , " 后面部分的值是40）
>
> （3）在所有运算符中，逗号运算符的优先级是最低的

```
eg1:    cats = 17 , 240;
被解释为：(cats=17),240;    将cats设置成17, 240不起作用!

eg2:    cats = (17 , 240);
因为括号的优先级最高，执行顺序是：
（1）括号内部：17,240    得出值是240
（2）240返回给括号
（3）括号通过'='向左赋值给cats
最终：将cats设置成 240!
```

2）while 循环：

（1）通用格式：

```
while(test-condition)
{
    body...
}
```

（2）执行原理：

首先，程序计算圆括号内的测试条件（test-condition）表达式。如果该表达式为 true，则执行循环体中的语句。与 for 循环一样，循环体也由一条语句或两个花括号定义的语句块组成。执行完循环体后，程序返回测试条件，对它进行重新评估。如果该条件为非零，则再次执行循环体。测试和执行将一直进行下去，直到测试条件为 false 为止（参见图 5.3）。显然，如果希望循环最终能够结束，循环体中的代码必须完成某种影响测试条件

（3）设置类型别名：

方法1：通过预处理器

```
#define BYTE char
//预处理器在编译程序时用char替换所有的BYTE，从而使BYTE成为char的别名
```

方法2：使用c++关键字 typedef 创建别名

```
typedef char BYTE;
//使 BYTE 成为 char 的别名
```

3）do-while 循环：

通用格式：

```
do
{
    ......
} while(test-expression)p;
```

执行原理：

前面已经学习了 for 循环和 while 循环。第 3 种 C++循环是 do while，它不同于另外两种循环，因为它是出口条件（exit condition）循环。这意味着这种循环将首先执行循环体，然后再判定测试表达式，决定是否应继续执行循环。如果条件为 false，则循环终止；否则，进入新一轮的执行和测试。这样的循环通常至少执行一次，因为其程序流必须经过循环体后才能到达测试条件。下面是其句法：

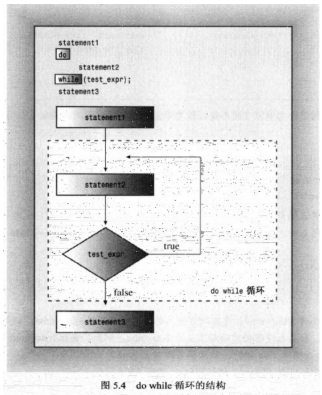


图 5.4 do while 循环的结构

4）基于范围的 for 循环：

C++11 新增了一种循环：基于范围（range-based）的 for 循环。这简化了一种常见的循环任务：对数组（或容器类，如 vector 和 array）的每个元素执行相同的操作，如下例所示：

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
    cout << x << std::endl;
```

其中，x 最初表示数组 prices 的第一个元素。显示第一个元素后，不断执行循环，而 x 依次表示数组的其他元素。因此，上述代码显示全部 5 个元素，每个元素占据一行。总之，该循环显示数组中的每个值。

要修改数组的元素，需要使用不同的循环变量语法：

```
for (double &x : prices)
    x = x * 0.80;          //20% off sale
```

符号&表明 x 是一个引用变量，这个主题将在第 8 章讨论。就这里而言，这种声明让接下来的代码能够修改数组的内容，而第一种语法不能。

还可结合使用基于范围的 for 循环和初始化列表：

```
for (int x : {3, 5, 2, 8, 6})
    cout << x << " ";
cout << '\n';
```

5）循环与文本输入：

（1）cin 与 cin.get(ch)：

> [1]提出问题：

```
// textin1.cpp -- reading chars with a while loop
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int count = 0;      // use basic input
    cout << "Enter characters; enter # to quit:\n";
    cin >> ch;          // get a character
    while (ch != '#')   // test the character
    {
        cout << ch;     // echo the character
        ++count;        // count the character
        cin >> ch;      // get the next character
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

下面是该程序的运行情况：

```
Enter characters; enter # to quit:
see ken run#really fast
seekenrun
9 characters read
```

上面的做法合情合理。但为什么程序在输出时省略了空格呢？原因在 cin。读取 char 值时，与读取其他基本类型一样，cin 将忽略空格和换行符。因此输入中的空格没有被回显，也没有被包括在计数内。

更为复杂的是，发送给 cin 的输入被缓冲。这意味着只有在用户按下回车键后，他输入的内容才会被发送给程序。这就是在运行该程序时，可以在#后面输入字符的原因。按下回车键后，整个字符序列将被发送给程序，但程序在遇到#字符后将结束对输入的处理。

[2]改进方法，并提出使用cin.get(ch)解决问题：

> 方法：

```
char ch;        //  不可以省略，否则就是未定义变量ch
cin.get(ch);    //  读取输入中的下一个字符(即使它是空格)，然后将其赋给ch
```

> 例证：

```cpp
// textin2.cpp -- using cin.get(char)
#include <iostream>
int main()
{
    using namespace std;
    char ch;
    int count = 0;
    cout << "Enter characters; enter # to quit:\n";

    cin.get(ch);        // use the cin.get(ch) function
    while (ch != '#')
    {
        cout << ch;
        ++count;
        cin.get(ch);    // use it again
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

（2）文件尾（EOF）条件：

> 原理：如果检测到文件尾的EOF，fail( )成员函数返回true；否则返回false！

```cpp
// textin3.cpp -- reading chars to end of file
#include <iostream>
using namespace std;
int main()
{
    char ch;
    int count = 0;
    cin.get(ch);        // attempt to read a char
    while (cin.fail() == false)  // test for EOF
    {
        cout << ch;     // echo character
        ++count;
        cin.get(ch);    // attempt to read another char
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

ps： windows系统上以ctrl+Z模拟EOF条件

6）二维数组与嵌套循环：

> 过于简单，略之！