Statements and flow control

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (`;`), and are executed in the same order in which they appear in a program.

一个简单的c++语句是程序中的每一条单独的指令，就像前面小节中看到的变量声明和表达式一样。它们总是以分号(';')结尾，并且执行的顺序与它们在程序中出现的顺序相同。

But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

但是程序并不局限于语句的线性序列。在这个过程中，程序可能会重复一段代码，或者做出决定并进行分叉。为此，c++提供了流控制语句，用于指定程序在何时以及在何种情况下必须执行的操作。

Many of the flow control statements explained in this section require a generic (sub)statement as part of its syntax. This statement may either be a simple C++ statement, -such as a single instruction, terminated with a semicolon (`;`) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {}:

本节中解释的许多流控制语句都需要通用(子)语句作为其语法的一部分。该语句可以是简单的c++语句，例如以分号(';')结束的单个指令，也可以是复合语句。复合语句是一组语句(每条语句都以自己的分号结束)，但它们都被组合在一个块中，用花括号括起来:{}:

```
{ statement1; statement2; statement3; }
```

The entire block is considered a single statement (composed itself of multiple substatements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

整个块被认为是单个语句(由多个子语句组成)。只要泛型语句是流控制语句语法的一部分，它可以是简单语句，也可以是复合语句。

Selection statements: if and else

The `if` keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

if关键字用于执行语句或块，当且仅当条件满足时。它的语法是:

```
if (condition) statement
```

Here, `condition` is the expression that is being evaluated. If this `condition` is true, `statement` is executed. If it is false, `statement` is not executed (it is simply ignored), and the program continues right after the entire selection statement.

这里，condition是要计算的表达式。如果此条件为真，则执行语句。如果为false，则不执行语句(它被简单地忽略)，程序将在整个选择语句之后继续执行。

For example, the following code fragment prints the message `(x is 100)`, only if the value stored in the `x` variable is indeed 100:

例如，以下代码片段只在x变量中存储的值确实是100时打印消息(x是100):

```
if (x == 100)
  cout << "x is 100";
```

If `x` is not exactly 100, this statement is ignored, and nothing is printed.

如果x不等于100，则忽略此语句，不打印任何内容。

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces (`{}`), forming a block:

如果你想在条件满足时包含多个要执行的语句，这些语句应该用大括号({})括起来，形成一个块:

```
if (x == 100)
{
   cout << "x is ";
   cout << x;
}
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

和往常一样，代码中的缩进和换行没有作用，所以上面的代码相当于:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with `if` can also specify what happens when the condition is not fulfilled, by using the `else` keyword to introduce an alternative statement. Its syntax is:

带有if的选择语句还可以通过使用else关键字引入替代语句来指定在条件未满足时将发生什么。它的语法是:

`if (condition) statement1 else statement2`

where `statement1` is executed in case condition is true, and in case it is not, `statement2` is executed.

在case condition为真时执行statement1，在case condition为假时执行statement2。

For example:

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

This prints `x is 100`, if indeed x has a value of 100, but if it does not, and only if it does not, it prints `x is not 100` instead.

如果x的值确实为100，则输出x is 100，但如果不是100，仅当不是100时，输出x is not 100。

Several if + else structures can be concatenated with the intention of checking a range of values. For example:

可以连接多个if + else结构以检查值的范围

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: `{}`.

它通过连接两个if-else结构输出x是正、负还是零。同样，通过将它们分组到用大括号括起来的块中:'{}'，也可以在每个case中执行多个语句。

Iteration statements (loops)

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords `while`, `do`, and `for`.

循环将语句重复一定次数，或者在条件满足时重复语句。它们由关键词while、do和for引入。

The while loop

The simplest kind of loop is the while-loop. Its syntax is:

最简单的循环是while循环。它的语法是:

```
while (expression) statement
```

The while-loop simply repeats `statement` while `expression` is true. If, after any execution of `statement`, `expression` is no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

while循环只是重复语句while表达式为真。如果在任何语句执行后，表达式不再为真，则循环结束，程序在循环结束后继续执行。例如，让我们看看使用while循环的倒计时:

```
// custom countdown using while
#include <iostream>
using namespace std;

int main ()
{
  int n = 10;

  while (n>0) {
    cout << n << ", ";
    --n;
  }

  cout << "liftoff!\n";
}
```

The first statement in `main` sets n to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition `n>0` (that n is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (`n>0`) remains being true.

main中的第一个语句将n的值设置为10。这是倒数的第一个数字。然后开始while循环:如果这个值满足条件n>0 (n大于0)，则执行条件后面的块，并一直重复，直到条件(n>)为真。

The whole process of the previous program can be interpreted according to the following script (beginning in `main`):

上一个程序的整个过程可以按照下面的脚本(从main开始)来解释:

1. `n` is assigned a value
2. The

   ```
   while
   ```

   condition is checked (

   ```
   n>0
   ```

). At this point there are two possibilities:
   - condition is true: the statement is executed (to step 3)
   - condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   `cout << n << ", ";--n;` (prints the value of `n` and decreases `n` by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:
   print `liftoff!` and end the program.

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever. In this case, the loop includes `--n`, that decreases the value of the variable that is being evaluated in the condition (`n`) by one - this will eventually make the condition (`n>0`) false after a certain number of loop iterations. To be more specific, after 10 iterations, `n` becomes 0, making the condition no longer true, and ending the while-loop.

使用while-loops要考虑的一件事是，循环应该在某个点结束，因此语句应该以某种方式改变条件中检查的值，从而迫使它在某个点变成false。否则，循环将永远循环下去。在本例中，循环包含——n，使条件(n)中正在计算的变量的值减少1——这将最终使条件(n>0)在经过一定次数的循环迭代后为假。更具体地说，在10次迭代之后，n变成0，使条件不再为真，结束while循环。

Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count (if interested, see `sleep_for` for a countdown example with delays).

请注意，这个循环的复杂性对于计算机来说是微不足道的，因此整个倒计时将立即执行，计数元素之间没有任何实际的延迟(如果感兴趣，请参阅sleep_for的延迟倒计时示例)。

The do-while loop

A very similar loop is the do-while loop, whose syntax is:

一个非常类似的循环是do-while循环，其语法是

```
do statement while (condition);
```

It behaves like a while-loop, except that `condition` is evaluated after the execution of `statement` instead of before, guaranteeing at least one execution of `statement`, even if `condition` is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

它的行为类似于while循环，除了' condition '是在' statement '执行后而不是在' statement '之前计算，确保' statement '至少执行一次，即使' condition '从未实现。例如，下面的示例程序回显用户输入的任何文本，直到用户输入再见:

```
// echo machine
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str;
  do {
    cout << "Enter text: ";
    getline (cin,str);
    cout << "You entered: " << str << '\n';
  } while (str != "goodbye");
}
```

The do-while loop is usually preferred over a while-loop when the `statement` needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering goodbye, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

当语句需要至少执行一次时，例如当检查循环结束的条件是在循环语句本身内确定时，通常首选do-while循环而不是while循环。在前面的例子中，块中的用户输入将决定循环是否结束。因此，即使用户想通过输入goodbye来尽快结束循环，循环中的块也需要至少执行一次，以提示输入，而条件实际上只能在执行之后才能确定。

The for loop

The `for` loop is designed to iterate a number of times. Its syntax is:

for循环被设计为多次迭代。它的语法是:

`for (initialization; condition; increase) statement;`

Like the while-loop, this loop repeats `statement` while `condition` is true. But, in addition, the for loop provides specific locations to contain an `initialization` and an `increase` expression, executed before the loop begins the first time, and after each iteration, respectively. Therefore, it is especially useful to use counter variables as `condition`.

与while循环类似，此循环重复语句while条件为真。但是，另外，for循环提供了包含初始化表达式和递增表达式的特定位置，分别在第一次循环开始之前和每次迭代之后执行。因此，使用计数器变量作为条件特别有用。

It works in the following way:

1. `initialization` is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. `condition` is checked. If it is true, the loop continues; otherwise, the loop ends, and `statement` is skipped, going directly to step 5.
3. `statement` is executed. As usual, it can be either a single statement or a block enclosed in curly braces `{ }`.
4. `increase` is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.

Here is the countdown example using a for loop:

```
// countdown using a for loop
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, `for (;n<10;)` is a loop without *initialization* or *increase* (equivalent to a while-loop); and `for (;n<10;++n)` is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with `true` as condition (i.e., an infinite loop).

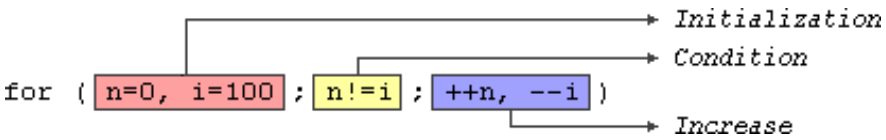
for循环中的三个字段是可选的。它们可以保留为空，但在所有情况下，它们之间的分号都是必需的。例如，For (;n<10;)是一个没有初始化或递增的循环(相当于while循环);for (;n<10;++n)是一个带有递增的循环，但没有初始化(可能是因为变量在循环之前已经初始化了)。没有条件的循环等价于条件为true的循环(即无限循环)。

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (`,`): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables, initializing and increasing both:

由于每个字段都在循环生命周期的特定时间内执行，因此可以将多个表达式作为任何初始化、条件或语句执行。不幸的是，这些不是语句，而是简单的表达式，因此不能用块代替。然而，作为表达式，它们可以使用逗号操作符(，):该操作符是一个表达式分隔符，可以在通常只需要一个表达式的情况下分隔多个表达式。例如，使用它，For循环可以处理两个计数器变量，初始化和递增两个变量:

```
for ( n=0, i=100 ; n!=i ; ++n, --i )
{
   // whatever here...
}
```

This loop will execute 50 times if neither `n` or `i` are modified within the loop:



`n` starts with a value of 0, and `i` with 100, the condition is `n!=i` (i.e., that `n` is not equal to `i`). Because `n` is increased by one, and `i` decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both `n` and `i` are equal to 50.

n以0开始，i以100开始，条件是m !=i(即n不等于i)。因为每次迭代n增加1,i减少1，当n和i都等于50时，循环的条件将在第50次迭代后变为假。

Range-based for loop

The for-loop has another syntax, which is used exclusively with ranges:

for循环还有另一种语法，专门用于范围:

```
for ( declaration : range ) statement;
```

This kind of for loop iterates over all the elements in `range`, where `declaration` declares some variable able to take the value of an element in this range. Ranges are sequences of elements, including arrays, containers, and any other type supporting the functions begin and end; Most of these types have not yet been introduced in this tutorial, but we are already acquainted with at least one kind of range: strings, which are sequences of characters.

这种for循环遍历' range '中的所有元素，其中' declaration '声明了某个变量，该变量可以接受该范围内元素的值。范围是元素的序列，包括数组、容器和任何其他支持begin和end函数的类型;大多数类型在本教程中还没有介绍，但我们已经熟悉了至少一种范围:字符串，即字符序列。

An example of range-based for loop using strings:

```
// range-based for loop
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string str {"Hello!"};
  for (char c : str)
  {
    cout << "[" << c << "]";
  }
  cout << '\n';
}
```

补充：实际测试，要c++11才可以支持

Note how what precedes the colon (`:`) in the for loop is the declaration of a `char` variable (the elements in a string are of type `char`). We then use this variable, `c`, in the statement block to represent the value of each of the elements in the range.

注意，在for循环中，冒号(:)前面是char变量的声明(字符串中的元素是char类型的)。然后在语句块中使用这个变量c来表示范围内每个元素的值。

This loop is automatic and does not require the explicit declaration of any counter variable.

这个循环是自动的，不需要显式声明任何计数器变量。

Range based loops usually also make use of type deduction for the type of the elements with `auto`. Typically, the range-based loop above can also be written as:

基于范围的循环通常也会使用auto来推断元素的类型。通常，上面基于范围的循环也可以写成:

```
for (auto c : str)
  cout << "[" << c << "]";
```

Here, the type of `c` is automatically deduced as the type of the elements in `str`.

在这里，c的类型被自动推断为str中元素的类型。

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

跳转语句允许通过执行到特定位置的跳转来改变程序的流程。

The break statement

`break` leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

' break '会离开一个循环，即使它的结束条件没有满足。它可以用来结束一个无限循环，或者迫使它在自然结束之前结束。例如，让我们在它自然结束前停止倒计时:

```
// break loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--)
  {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
}
```

The continue statement

The `continue` statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

continue语句导致程序跳过当前迭代中循环的其余部分，就好像已经到达语句块的末尾，导致它跳转到下一个迭代的开始。例如，让我们跳过倒数中的第5项:

```
// continue loop example
#include <iostream>
using namespace std;

int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
    cout << n << ", ";
  }
  cout << "liftoff!\n";
}
```

The goto statement

`goto` allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

Goto允许绝对跳转到程序中的另一个点。这种无条件跳转忽略嵌套级别，并且不会导致任何自动堆栈展开。因此，它是一个需要谨慎使用的特性，最好在同一个语句块中使用，特别是在存在局部变量的情况下。

The destination point is identified by a *label*, which is then used as an argument for the `goto` statement. A *label* is made of a valid identifier followed by a colon (`:`).

目标点由标签标识，然后用作goto语句的参数。标签由一个有效的标识符和一个冒号(:)组成。

`goto` is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using goto:

goto通常被认为是一个低级特性，在通常与c++一起使用的现代高级编程范例中没有特定的用例。但是，只是作为一个例子，这里有一个版本的倒计时循环使用goto:

```
// goto loop example
#include <iostream>
using namespace std;

int main ()
{
  int n=10;
mylabel:
  cout << n << ", ";
  n--;
  if (n>0) goto mylabel;
  cout << "liftoff!\n";
}
```

Another selection statement: switch.

The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating `if`-`else` statements, but limited to constant expressions. Its most typical syntax is:

switch语句的语法有点奇怪。它的目的是在许多可能的常量表达式中检查一个值。它类似于连接if-else语句，但仅限于常量表达式。它最典型的语法是:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

It works in the following way: `switch` evaluates `expression` and checks if it is equivalent to `constant1`; if it is, it executes `group-of-statements-1` until it finds the `break` statement. When it finds this `break` statement, the program jumps to the end of the entire `switch` statement (the closing brace).

它的工作方式如下:switch计算expression并检查它是否等价于constant1;如果是，则执行group-of-statements-1直到找到break语句。当它找到这个break语句时，程序跳转到整个switch语句的末尾(右大括号)。

If expression was not equal to `constant1`, it is then checked against `constant2`. If it is equal to this, it executes `group-of-statements-2` until a `break` is found, when it jumps to the end of the switch.

如果expression不等于constant1，则根据constant2检查它。如果它等于这个，它执行group-of-statements-2，直到找到一个break，然后跳转到开关的末端。

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the `default:` label, if it exists (since it is optional).

最后，如果表达式的值不匹配前面指定的任何常量(可能有任意数量的常量)，则程序执行default: label后面包含的语句，如果它存在(因为它是可选的)。

Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

下面的两个代码片段具有相同的行为，演示了等价于switch语句的if-else:

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

```
if (x == 1) {
  cout << "x is 1";
}
else if (x == 2) {
  cout << "x is 2";
}
else {
  cout << "value of x unknown";
}
```

The `switch` statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that `break` statements are needed after each group of statements for a particular label. If `break` is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as `break`) is reached.

switch语句有一种从早期的第一个C编译器继承而来的有点奇怪的语法，因为它使用标签而不是块。在最典型的用法中(如上所示)，这意味着对于一个特定的标签，在每组语句之后都需要break语句。如果不包含break，则会执行该case下的所有语句(包括任何其他标签下的语句)，直到到达switch块或跳转语句(如break)的末尾。

If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing `x is 1`, and would instead continue executing the statements in case two (thus printing also `x is 2`). It would then continue doing so until a `break` statement is encountered, or the end of the `switch` block. This makes unnecessary to enclose the statements for each case in braces `{}`, and can also be useful to execute the same group of statements for different possible values. For example:

如果上面的示例缺乏break语句后,第一组的情况下,这个程序不会自动跳转到开关组结束后打印x = 1,并将继续执行语句,以防两个(因此也打印x = 2)。它会继续这样做,直到遇到break语句,或开关块的末端。这样就没有必要在大括号{}中包含每种情况的语句，对于不同的可能值执行同一组语句也很有用。例如:

```
switch (x) {
  case 1:
  case 2:
  case 3:
    cout << "x is 1, 2 or 3";
    break;
  default:
    cout << "x is not 1, 2 nor 3";
  }
```

Notice that `switch` is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.

注意switch仅限于将其求值表达式与常量表达式标签进行比较。不能将变量用作标签或范围，因为它们不是有效的c++常量表达式。

To check for ranges or values that are not constant, it is better to use concatenations of `if` and `else if` statements.

要检查非常量的范围或值，最好使用if和else if语句的串联。