Modern C++ - Statements

6. Statements

Statements, control structures

Statements, constrol structures are the smallest standalone elements in an *imperative programming language* commanding the computer to carry out some actions. As C++ has roots from the imperative C language, most of the statements are inherited from there.

Expression statement

An expression followed by a *semicolon* (;) is an expression-statement.

```
printf("Hello world\n")
```

is an expression (type is *int*, value is 12, as printf returns with the number of bytes about to print). It is valid (but unusual) to write this:

```
printf("Hello world\n") + 3
```

Most cases we just use the *side effect* of the expression and make it as a statement.

```
printf("Hello world\n");
```

Null statement

Null statement is a statement without effect. It is a single semicolon.

```
1 if ( x < 10 )
2   ;
3 else
4   std::cout << "else branch";
```

Compound statement

Compound statement (sometimes called as *block* is to create a single statement from multiply statements. A body of a function is a compound statement too.

```
1 if ( x < 10 )
2 {
3   ;
4 }
5 else
6 {
7   std::cout << "compund statement";
8   std::cout << "in the else branch";
9 }
```

Its a good idea to use compound statements in every control structure.

If statement

The if (selection) statement is a control statement. It has two forms.

```
if (expression) statement
if (expression) statement1; else statement2;
```

The statements may (and suggested to be) compound statements.

Dangling ifs are matched to the closest if statement:

```
1 if ( x < 10 )
2   if ( y > 5 )
3     std::cout << "x < 10 and y > 5";
4 else
5   std::cout << "x < 10 and y <= 5";
```

is equivalent of:

```
1 if ( x < 10 )
2 {
3   if ( y > 5 )
4     std::cout << "x < 10 and y > 5";
5   else
6     std::cout << "x < 10 and y <= 5";
7 }
```

and different from this:

```
1 if ( x < 10 )
```

```
2 {
3   if ( y > 5 )
4     std::cout << "x < 10 and y > 5";
5 }
6 else
7   std::cout << "x >= 10";
```

Sometimes if and else statements appear in a specific form:

```
 1 if ( x < 10  &&  y > 5 )
 2 {
 3   std::cout << "x < 10 and y > 5";
 4 }
 5 else if ( x < 10  &&  y <= 5 )
 6 {
 7   std::cout << "x < 10 and y <= 5";
 8 }
 9 else if ( x >= 10  &&  y > 5 )
10 {
11   std::cout << "x >= 10 and y > 5";
12 }
13 else if ( x >= 10  &&  y <= 5 )
14 {
15   std::cout << "x >= 10 and y <= 5";
16 }
17 else
18 {
19   printf("impossible");
20 }
```

Switch (selection) statement

The switch statement is an alternative selection statement.

```
switch (expression) statement
```

The statement almost always is a compound statement with labelled statements.

```
1 int day_of_week;
2 //...
3 switch ( day_of_week )
```

```
 4 {
 5   default: std::cout << "Udefined";  break;
 6   case  2: std::cout << "Monday";    break;
 7   case  3: std::cout << "Tuesday";   break;
 8   case  4: std::cout << "Wednesday"; break;
 9   case  5: std::cout << "Thursday";  break;
10   case  6: std::cout << "Friday";    break;
11   case  1:
12   case  7: std::cout << "Week-end";  break;
13 }
```

The statements will *fall through* to the next case. That is why it is suggested to use the *break* statements, to pass the control to the next statement after the switch.

While (iteration) statement

```
while ( expression ) statement
```

The while loop checks condition for truthfulness *before* executing the *statement*.

```
 1 struct list_type
 2 {
 3   int       value;
 4   list_type *next;
 5 };
 6 // ...
 7 list_type *ptr = first;
 8 while ( NULL != ptr  )
 9 {
10   std::cout << ptr->value << " ";
11   ptr = ptr->next;
12 }
```

Do-while (iteration) statement

```
do statement while ( expression ) ;
```

The do-while loop is different from the while loop since the *statement* will be executed at least ones. The conditional expression is checked only *after* the statement.

The do-while statement is equivalent of:

```
statement
while ( expression )
  statement
```

For (iteration) statement

```
for ( opt-expr-1 ; opt-expr-2 ; opt-expr-3 ) statement
for ( declaration; opt-expr-2 ; opt-expr-3 ) statement
for ( range-declaration : range-expression ) statement (C++11)
```

where

1. opt-expr-1 is an optional initialization expression, executed once before the first iteration. In C++ opt-expr-1 can be replaced by a *declaration*.
2. opt-expr-2 is an optional control expression, evaluated before every iteration, and the *statement* is executed only when the control expression is true.
3. opt-expr-3 is an optional iteration expression, executed every time after the *statement*.

Therefore,

```
for ( e1 ; e2 ; e3 ) s;
```

is equivalent of

```
{
  e1;
  while ( e2 )
  {
    s;
    e3;
  }
}
```

The three expressions are all optionals. If the second expression is missing, then it is equivalent of *true*. The *infinite loop* is usually written in teh following way:

```
for( ; ; ) statement
```

You can still leave this tool with a *return* or *break* statement.

Since C++11 we can use *range for* statement to iterate on a certain interval of values:

```
1 std::vector<int> v = { 0, 1, 2, 3 };
2 for ( int i : v )
3 {
4   std::cout << v[i] << " ";
5 }
6 // i is not visible here.
```

In C++11 we frequently use type deduction for the declaration of the range variable using **auto** keyword.

```
1 std::vector<int> v = { 0, 1, 2, 3 };
2 for ( auto i : v )
3 {
4   std::cout << v[i] << " ";
5 }
6 // i is not visible here.
```

Break and Continue (jump) in loops

The break statement in a loop or a switch statement jumps to the following statement.

```
 1 int t[10];
 2 // ...
 3 for ( int i = 0; i < 10; ++i )
 4 {
 5   if ( t[i] < 0 )
 6   {
 7     std::cout << "negative found");
 8     break;
 9   }
10   std::cout << "do something with non-negatives";
11 }
12 // break jumps to here
```

The continue statement will jump over the rest of the loop.

```
1 int t[10];
2 // ...
3 for ( int i = 0; i < 10; ++i )
4 {
```

```
 5   if ( t[i] < 0 )
 6   {
 7     std::cout << "negative found";
 8     continue;
 9   }
10   std::cout << "do something with non-negatives";
11   // ...
12   // continue jumps to here
13 }
```

Return (jump) statement

```
return;
return expr;
```

The return statements returns from the actually executed function to the caller. There might be multiply return statements in a function.

Returning from main will finish the program execution.

```
 1 int find_first_negative( int t[], int length)
 2 {
 3   for ( int i = 0; i < length; ++i )
 4   {
 5     if ( t[i] < 0 )
 6     {
 7        std::cout << "negative found";
 8       return t[i];
 9     }
10   }
11   return 0;
12 }
```

Goto (jump) statement

```
goto label ;
```

where label is an identifier. Don’t use the goto statement.