

IDK1531 Advanced C++ Course

C++ Statements

Aleksandr Lenin

Tallinn University of Technology

February 5th, 2019

Basic Concepts

A C++ program is a set of text files that contain **statements**. They undergo translation to become an executable program.

Words having special meaning in C++ are **keywords**. Others can be used as **identifiers**.

Comments are ignored during translation.

```
// this is a single line comment
/*
   this is a multiline comment
*/
```

Certain characters in the program have to be represented with **escape sequences**.

The entities of a C++ program are the following.

Objects – a region of storage that has **size,alignment requirement,storage duration,lifetime,type,value**, and an optional **name**.

References – an alias to an already existing object or function.

Functions – entities that associate a compound statement (a function body) with a name and a list of parameters.

Types – restrict operations that are permitted for entities and provide semantic meaning to them.

Templates – a C++ entity that defines a **family** of classes, functions, or variables (C++14), an **alias** to a family of types (C++11), **concept** (C++20).

Namespaces – provide a method for preventing name conflicts in large projects.

Entities are introduced by **declarations** which associate them with **names** and define their properties.

A C++ source file consists of **statements**.

Statements are parts of the C++ program that are executed **sequentially**.

Statement Labels

Any statement can be **labelled**. There are 3 use cases for labeled statements.

- 1 Target for a `goto` unconditional jump

```
[attr] identifier : statement
```

- 2 Case label in a `switch` statement

```
[attr] case constexpr: statement
```

- 3 Default label in a `switch` statement

```
[attr] default: statement
```

A statement may have multiple labels.

A label declared inside a function is in the scope **everywhere** in that function, in all nested blocks, before and after its declaration.

Labels are ignored by unqualified name lookup. It means that a label can have the same name as any other entity in a program.

```
int x = 3;
x: hello:    // defined two labels for a single statement
std::cout << "Hello " << x << std::endl;
x-=1;
if (x==2) goto x;
else if (x==1) goto hello;
```

Output:

```
Hello 3
Hello 2
Hello 1
```

Common statement attributes provide a unified syntax for implementation-defined language extensions, such as GNU, IBM, Microsoft.

Attributes unknown to an implementation are ignored without causing an error.

If an attribute appears before the label, it applies to the label.

If an attribute appears after the label, but before the statement, it applies to the statement.

The most common standard attributes are given below.

[[noreturn]] – indicates that a function does not return

[[deprecated("reason")]] – indicates that the use of a name is deprecated for a specific reason. The ("reason") specification is optional. (C++14)

[[fallthrough]] – indicates that a fall through case labels is intended in switch statement. (C++17)

[[nodiscard]] – makes the compiler to issue a warning if the return value is discarded. (C++17)

[[maybe_unused]] – suppresses compiler warnings about unused entities. (C++17)

The C++ standard defines the following types of statements:

- 1 Expression statements
- 2 Compound statements
- 3 Selection statements
- 4 Iteration statements
- 5 Jump statements
- 6 Declaration statements
- 7 Try blocks

Expression Statements

An **expression statement** is an **expression** followed by a semicolon

```
[attr] [expression];
```

Most statements in a C++ program are expression statements, such as assignments or function calls.

An expression statement without an expression is called a **null statement**. It is often used to provide an empty body to a [for](#) loop.

Example

```
x = x + 1;
```

Compound Statements

Compound statements or **blocks** group a sequence of statements into a single statement.

Compound statements may be used when one statement is expected, but multiple statements need to be executed.

```
[attr] { [statement]... }
```

Example

```
{  
    int a = 2;           // declaration statement  
    a = a * 5;          // expression statement  
    std::cout << a;     // expression statement  
}  
                        // end of block, end of statement  
  
{ } // a valid empty statement
```

Every compound statement introduces its own **block scope**. Variables declared inside a block are destroyed at the closing brace in reverse order.

Example

```
{ // start of outer block
  { // start of inner block
    int x; // declaration statement
    std::string s; // declaration statement
    std::pair<int,int> point2D; // declaration statement
  } // end of inner block, point2D is destroyed, then s, then x

  int x; // name x is free to use
  std::string s; // name s is free to use
} // end of outer block
```

Selection statements execute a **specific branch** in the flow control, depending on the result of evaluation of a **condition**.

`if` statement conditionally executes another statement based on a run-time or compile-time condition.

```
[attr] if (condition) statement-true [ else statement-false ]
```

condition is one of

- an expression implicitly convertible to type `bool`
- a single non-array variable with a brace or equals initializer

statement-true and **statement-false** are any statements.

Example

```
int x = 2;
if (x < 0) x = 0;           // does pretty much nothing

if (int x = 2) std::cout << x;    // prints 2

if (int x=0) std::cout << x;
else std::cout << "Else" << std::endl; // prints "Else"

if (int x{3}) std::cout << x;     // prints 3

if (int x{0}) std::cout << x;
else std::cout << "Else" << std::endl; // prints "Else"
```

```
[attr] if [constexpr] ([init-statement] condition) statement-true [ else statement-false ]
```

init-statement is one of

- an expression statement
- a simple declaration

condition is one of

- an expression implicitly convertible to type `bool`
- a single non-array variable with a brace or equals initializer

If used with `constexpr`, the condition must be an expression implicitly convertible to a **constant expression** of type `bool`.

Statement-true and **statement-false** are any statements.

Example

```
int k = 10;
if (k-4; k > 5) std::cout << k;           // prints 10 -- Why?
if (k-=4; k > 5) std::cout << k;         // prints 5
if (int x = 2; x > 1) std::cout << x;    // prints 2
if (int x = 2; x > 2) std::cout << x;    // dead code
if (int x=2; int y=10) std::cout << x << " " << y; // prints 2 10
if (int x=2; int y{5}) std::cout << x << " " << y; // prints 2 5
```

`switch` statement transfers control to one of the conditional statements based on the value of a condition.

```
[attr] switch ([init-statement] condition) statement  
[attr] case constant-expression : statement  
[attr] default: statement
```

condition any of

- expression of integral or enumeration type
- class type implicitly convertible to integral or enumeration type
- declaration of a single non-array variable with a brace or equals initializer

init-statement one of

- an expression statement
- a declaration of one or more variables, optionally with an initializer

statement is any statement.

- **case**: and **default**: labels are permitted.
- **break**; statement has special meaning.

constant-expression of the same type as condition after implicit conversions.

A `switch` statement body may have an arbitrary number of `case:` labels as long as the constant-expressions are unique.

At most one `default:` label may be present.

If condition evaluates to the value of any of the constant-expressions, the control is transferred to the matched label.

If no match was found, and the `default:` label is present, the control is transferred to the statement labeled with the `default:` label.

The `break;` statement inside statement exits the `switch` statement.

Some compilers may issue a warning on fallthrough, unless attribute `[[fallthrough]]` is prepended immediately before the case label, this tells the compiler that the fallthrough is intentional.

Example

```
switch (2)
```

```
    std::cout << 2;    // does nothing
```

```
switch (2) {
```

```
    case 1: std::cout << 1;
```

```
    case 2: std::cout << 2;           // prints 2
```

```
    case 3: std::cout << 3;           // prints 3
```

```
    default: std::cout << "default"; // prints "default"
```

```
}
```

```
switch (2) {
```

```
    case 1: std::cout << 1; break;
```

```
    default: std::cout << "default"; // prints "default"
```

```
}
```

```
using QTY = enum { ONE,TWO,THREE }; QTY qty = TWO;
```

```
switch (qty) {
```

```
    case ONE: std::cout << 1; break;
```

```
    case TWO: std::cout << 2; break; // prints 2
```

```
    case THREE: std::cout << 3; break;
```

```
    default: std::cout << "default";
```

```
}
```

Example

```
switch (int x = 2) {  
    case 1: std::cout << 1; break;  
    case 2: std::cout << 2; break; //prints 2  
}  
  
switch (int x{2}) {  
    case 1: std::cout << 1; break;  
    case 2: std::cout << 2; break; //prints 2  
}  
  
switch (int x = 2, y = 3; x*y+10) {  
    case 12: std::cout << 12; break;  
    case 16: std::cout << 16; break; // prints 16  
    case 20: std::cout << 20; break;  
}
```

What is the problem with this code?

Example

```
switch (2) {  
  case 2:  
    int x = 2;  
    x = x + 3;  
    std::cout << x;  
    break;  
  case 3:  
    std::cout << 3;  
}
```

What is the problem with this code?

Example

```
switch (2) {  
  case 2:  
    int x = 2;  
    x = x + 3;  
    std::cout << x;  
    break;  
  case 3:  
    std::cout << 3;  
}
```

Jumping to `case 3`: would enter the scope of `x` without initializing it.

Transfers of control are not permitted to enter the scope of any variable.

Solution: the declaration of a variable has to be scoped in its compound statement.

Example

```
switch (2) {  
  case 2:  
    {  
      int x = 2;  
      x = x + 3;  
      std::cout << x;  
      break;  
    } // scope of x ends here  
  case 3:  
    std::cout << 3;  
}
```

Repetition Statements

Iteration statements repeatedly execute a statement.

`while` loop executes a statement repeatedly until the **condition** is implicitly convertible to boolean `false`.

The condition evaluation happens **before** every iteration is made.

If the condition is an expression, it is evaluated before every iteration, and the loop runs as long as the expression is evaluated to `true`. Once it is `false`, the loop is exited.

If the condition is a declaration, the initializer is evaluated before every iteration, and the loop runs as long as the declared variable is evaluated to `true`. Once it is `false`, the loop is exited.

```
[attr] while (condition) statement
```

condition is one of

- any expression implicitly convertible to type `bool`
- declaration of a single non-array variable with a brace or equals initializer

statement is any statement

Example

```
int x = 5;
while (x-- > 0)
    std::cout << x; // prints 43210
```

The `break;` statement in the body of the loop terminates the loop.

The `continue;` statement in the body of the loop transfers control to the end of the loop body.

The scope of variables declared in statement is limited to the `while` loop, as if it was a compound statement.

Example

```
int x = 5;
while (x-- > 0)
    int y;
// y goes out of scope
```

Example

```
int x = 5;
while (x-- > 0) {
    int y;
}
// y goes out of scope
```

If a condition is a declaration, the declared variable is in the scope of the body of `while` loop, and is **destroyed** and **recreated** in every iteration.

Example

```
std::string s = "Hello, World!";  
int i = 0;  
while (char c = s[i++])  
    std::cout << c;    // prints "Hello, World!"
```

`do-while` loop executes a statement repeatedly until the **condition** is implicitly convertible to boolean `false`.

The condition evaluation happens **after** every iteration is made.

```
[attr] do statement while (condition) ;
```

condition is any expression that is implicitly convertible to type `bool`.

statement is any statement.

Example

```
int x = 0;
do {
    std::cout << x;
    x++;
}
while ( x < 5 );    // prints 01234
```

The `break`; statement in the body of the loop terminates the loop.

The `continue`; statement in the body of the loop transfers control to the end of the loop body.

`for` loop executes **init-statement** once, then repeatedly executes the **statement** and **iteration expression** until the **condition** becomes implicitly convertible to boolean `false`.

```
[attr] for ( [init-statement] [condition] [iteration-expression] ) statement
```

init-statement is one of

- an expression statement
- a simple declaration

condition is one of

- an expression implicitly convertible to type `bool`. The expression is evaluated **before** every iteration.
- a declaration of a single non-array variable with brace or equals initializer. The initializer is evaluated **before** every iteration.

iteration-expression is any expression which is executed **after** every iteration of the loop.

statement is any statement.

Names declared by the `init`–statement as well as names declared by `condition` are in the scope of `statement`.

`break`; inside `statement` terminates the loop

`continue`; inside `statement` will execute `iteration`–`expression`.

Empty `condition` results in an infinite loop equivalent to `while(1)`.

The scope of variables declared inside `statement` is limited to the body of the `for` loop.

Example

```
for(int x=0; x<10; x+=3) {  
    std::cout << x;      // will print 0369  
}  
// x goes out of scope
```

Example

```
std::string s = "Hello, World!";  
for (int i=0; char c = s[i]; i++) {  
    std::cout << c;      // prints "Hello, World!"  
}
```

Example

```
std::string s = "Hello, World!";  
for(int i=0; char c = s[i++]; ) { // note the empty iteration-expression!  
    std::cout << c;              // prints "Hello, World!"  
}
```

Range based for loop operates over a **range** of values executing one iteration for every element in the range.

```
[attr] for ( [init-statement] range-declaration : range-expression ) statement
```

init-statement (C++20) is any of

- an expression statement
- a simple declaration

range-declaration is a declaration of a **named variable** whose type is equivalent to the type of the element represented by **range-expression**.

range-expression is any expression representing a range of elements – i.e., **array**, **brace initialized list**, **iterable object**.

statement is any statement.

Example

```
std::vector<int> vec{1,2,3,4};  
for (int i : vec)  
    std::cout << i; // prints 1234  
  
for (int i : {1,2,3,4})  
    std::cout << i; // prints 1234
```

Example

```
std::string s = "Hello, World!";  
for(char c : s )  
    std::cout << c; // prints "Hello, World!"  
  
for(char c : std::string("Hello,World!"))  
    std::cout << c; // prints "Hello, World!"
```

Example

C++20:

```
for(std::string s = "Hello, World!"; char c : s )  
    std::cout << c; // prints "Hello, World!"
```

The **range-declaration** may be a **structured binding declaration**

Example

```
std::map<int,std::string> dict = { {1,"Apples"}, {2,"Grapes"}, {3,"Oranges"} };  
for ( auto&& [key,value] : dict ) // access by forwarding reference  
    std::cout << key << " : " << value << std::endl;
```

Output:

```
1 : Apples  
2 : Grapes  
3 : Oranges
```


Jump Statements

`break`; terminates `for`, ranged `for`, `do-while` loop or `switch` statements.

`continue` skips the current iteration in `for`, ranged `for`, `do-while` loops and proceeds to the next one. In `for` loops, the iteration-expression is executed before proceeding with the next iteration.

`return` terminates the current function and returns an optional value to its caller.

```
[attr] return statement
```

statement is any of

- expression implicitly convertible to the return type of the enclosing function
- brace-enclosed list of initializers

`goto` statement performs an unconditional jump to a label.

```
[attr] goto label;
```

Declaration statements introduce one or more identifiers into a block.

Try Blocks

try blocks associate one or more exception handlers with a statement and allow to catch exceptions thrown by it.

```
try compound-statement handler-sequence
```

Example

```
try {  
    throw std::runtime_error("Fault");  
} catch (const std::runtime_error& e) {  
    std::cerr << "Exception: " << e.what() << std::endl;  
} catch (...) {  
    std::cerr << "Catch-all handler" << std::endl;  
}
```



THANK YOU
FOR
YOUR
ATTENTION
ANY QUESTIONS?