LEARN C++
Skill up with our free tutorials

6.1 — Compound statements (blocks)

ALEX AUGUST 5, 2021

A compound statement (also called a block, or block statement) is a group of *zero or more statements* that is treated by the compiler as if it were a single statement.

Blocks begin with a `{` symbol, end with a `}` symbol, with the statements to be executed being placed in between. Blocks can be used anywhere a single statement is allowed. No semicolon is needed at the end of a block.

You have already seen an example of blocks when writing functions, as the function body is a block:

```cpp
int add(int x, int y)
{ // start block
    return x + y;
} // end block (no semicolon)

int main()
{ // start block

    // multiple statements
    int value {}; // this is initialization, not a block
    add(3, 4);

    return 0;

} // end block (no semicolon)
```

Blocks inside other blocks

Although functions can't be nested inside other functions, blocks *can be* nested inside other blocks:

```cpp
int add(int x, int y)
{ // block
    return x + y;
} // end block

int main()
{ // outer block

    // multiple statements
    int value {};

    { // inner/nested block
        add(3, 4);
    } // end inner/nested block

    return 0;

} // end outer block
```

When blocks are nested, the enclosing block is typically called the outer block and the enclosed block is called the inner block or nested block.

Using blocks to execute multiple statements conditionally

One of the most common use cases for blocks is in conjunction with `if statements`. By default, an `if statement` executes a single statement if the condition evaluates to `true`. However, we can replace this single statement with a block of statements if we want multiple statements to execute when the condition evaluates to `true`.

For example:

```
#include <iostream>

int main()
{ // start of outer block
    std::cout << "Enter an integer: ";
    int value {};
    std::cin >> value;

    if (value >= 0)
    { // start of nested block
        std::cout << value << " is a positive integer (or zero)\n";
        std::cout << "Double this number is " << value * 2 << '\n';
    } // end of nested block
    else
    { // start of another nested block
        std::cout << value << " is a negative integer\n";
        std::cout << "The positive of this number is " << -value <<
'\n';
    } // end of another nested block

    return 0;
} // end of outer block
```

If the user enters the number 3, this program prints:

```
Enter an integer: 3
3 is a positive integer (or zero)
Double this number is 6
```

If the user enters the number -4, this program prints:

```
Enter an integer: -4
-4 is a negative integer
The positive of this number is 4
```

Block nesting levels

It is even possible to put blocks inside of blocks inside of blocks:

```
int main()
{ // block 1, nesting level 1
    std::cout << "Enter an integer: ";
    int value {};
    std::cin >> value;

    if (value >  0)
    { // block 2, nesting level 2
        if ((value % 2) == 0)
        { // block 3, nesting level 3
            std::cout << value << " is positive and
even\n";
        }
        else
        { // block 4, also nesting level 3
            std::cout << value << " is positive and
odd\n";
        }
    }

    return 0;
}
```

The nesting level (also called the nesting depth) of a function is the maximum number of nested blocks you can be inside at any point in the function (including the outer block). In the above function, there are 4 blocks, but the nesting level is 3 since in this program you can never be inside more than 3 blocks at any point.

The C++ standard says that C++ compilers should support 256 levels of nesting -- however not all do (e.g. as of the time of writing,

Visual Studio supports somewhere between 100 and 110).

Despite what C++ technically supports, it’s a good idea to keep your nesting level to 3 or less. Just as overly-long functions are good candidates for refactoring (breaking into smaller functions), overly-nested blocks are hard to read and are good candidates for refactoring (with the most-nested blocks becoming separate functions).

> **Best practice**
>
> Keep the nesting level of your functions to 3 or less. If your function has a need for more nested levels, consider refactoring your function into sub-functions.

Next lesson

6.2 User-defined namespaces and the scope resolution operator

Back to table of contents

Previous lesson

0.4 Converting between binary and decimal

DP N N FOUT

Newest

X

X