6.1 — Compound statements (blocks)

BY ALEX ON JUNE 18TH, 2007 | LAST MODIFIED BY NASCARDRIVER ON JANUARY 4TH, 2020

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:

```
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:

```
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
```

We'll talk more about `if statements`, including the use of blocks, in lesson **5.2 -- If statements**.

Block nesting levels

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

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

    if (value >  0)
    { // nesting level 2
        if ((value % 2) == 0)
        { // nesting level 3
            std::cout << value << " is positive and even\n";
        }
        else
        { // 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 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 you can never be inside more than 3 blocks at any point.

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 functions are also 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, consider refactoring.

6.2 -- User-defined namespaces

Index

0.4 -- Converting between binary and decimal

C++ TUTORIAL | PRINT THIS POST

50 comments to 6.1 — Compound statements (blocks)

Benur21
July 29, 2019 at 7:11 am · Reply

Can I use compound statements alone, outside any if, function, etc?
Eg:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";
    int value;
    std::cin >> value;
    {
      std::cout << value << " is a positive integer (or zero)" << std::endl;
      std::cout << "Double this number is " << value * 2 << std::endl;
    }
    return 0;
}
```

nascardriver
July 29, 2019 at 8:14 am · Reply

Yes.
If you think you need to do this to make your code more tidy, use a function instead.

Charan
December 14, 2019 at 12:07 am · Reply

But doesn't it raise the concerns about scope of a variable as shown in one of the comments.(Copying the code from @Alireza's comment)

Eg: