1.8 — Whitespace and basic formatting

ALEX JUNE 14, 2022

Whitespace is a term that refers to characters that are used for formatting purposes. In C++, this refers primarily to spaces, tabs, and newlines. The C++ compiler generally ignores whitespace, with a few minor exceptions (when processing text literals). For this reason, we say that C++ is a whitespace-independent language.

Consequently, the following statements all do the exact same thing:

```
std::cout << "Hello world!";

std::cout               <<            "Hello
world!";

        std::cout <<        "Hello world!";

std::cout
    << "Hello world!";
```

Even the last statement that is split over two lines compiles just fine.

The following functions all do the same thing:

```
int add(int x, int y) { return x + y;
}

int add(int x, int y) {
    return x + y; }

int add(int x, int y)
{    return x + y; }

int add(int x, int y)
{
    return x + y;
}
```

One exception where the C++ compiler does pay attention to whitespace is inside quoted text, such as `"Hello world!"`.

```
"Hello world!"
```

is different than:

```
"Hello     world!"
```

and each prints out exactly as you'd expect.

Newlines are not allowed in quoted text:

```
std::cout << "Hello
     world!"; // Not
allowed!
```

Quoted text separated by nothing but whitespace (spaces, tabs, or newlines) will be concatenated:

```
std::cout << "Hello "
     "world!"; // prints "Hello
world!"
```

Another exception where the C++ compiler pays attention to whitespace is with // comments. Single-line comments only last to the end of the line. Thus doing something like this will get you in trouble:

```
std::cout << "Hello world!"; // Here is a single-line
comment
this is not part of the comment
```

Basic formatting

Unlike some other languages, C++ does not enforce any kind of formatting restrictions on the programmer (remember, trust the programmer!). Many different methods of formatting C++ programs have been developed throughout the years, and you will find disagreement on which ones are best. Our basic rule of thumb is that the best styles are the ones that produce the most readable code, and provide the most consistency.

Here are our recommendations for basic formatting:

1. It's fine to use either tabs or spaces for indentation (most IDEs have a setting where you can convert a tab press into the appropriate number of spaces). Developers who prefer spaces tend to do so because it makes the formatting self-describing -- code that is spaced using spaces will always look correct regardless of editor. Proponents of using tabs wonder why you wouldn't use the character designed to do indentation for indentation, especially as you can set the width to whatever your preference is. There's no right answer here -- and debating it is like arguing whether cake or pie is better. It ultimately comes down to personal preference.

Either way, we recommend you set your tabs to 4 spaces worth of indentation. Some IDEs default to 3 spaces of indentation, which is fine too.

2. There are two acceptable styles for function braces.

The Google C++ style guide recommends putting the opening curly brace on the same line as the statement:

```
int main()
{
}
```

The justification for this is that it reduces the amount of vertical whitespace (you aren't devoting an entire line to nothing but the opening curly brace), so you can fit more code on a screen. More code on a screen makes the program easier to understand.

However, we prefer the common alternative, where the opening brace appears on its own line:

```
int
main()
{
}
```

This enhances readability, and is less error prone since your brace pairs should always be indented at the same level. If you get a compiler error due to a brace mismatch, it's very easy to see where.

3. Each statement within curly braces should start one tab in from the opening brace of the function it belongs to. For example:

```
int main()
{
    std::cout << "Hello world!\n"; // tabbed in one tab (4 spaces)
    std::cout << "Nice to meet you.\n"; // tabbed in one tab (4
spaces)
}
```

4. Lines should not be too long. Typically, 80 characters is the maximum length a line should be. If a line is going to be longer, it should be split (at a reasonable spot) into multiple lines. This can be done by indenting each subsequent line with an extra tab, or if the lines are similar, by aligning it with the line above (whichever is easier to read).

```
int main()
{
    std::cout << "This is a really, really, really, really, really, really, really, "
        "really long line\n"; // one extra indentation for continuation line

    std::cout << "This is another really, really, really, really, really, really, really, "
                 "really long line\n"; // text aligned with the previous line for continuation
line

    std::cout << "This one is short\n";
}
```

This makes your lines easier to read. On modern wide-screen monitors, it also allows you to place two windows with similar code side by side and compare them more easily.

Best practice

Your lines should be no longer than 80 chars in length.

Tip

Many editors have a built-in feature (or plugin/extension) that will show a line (called a “column guide”) at a given column (e.g. at 80 characters), so you can easily see when your lines are getting too long. To see if your editor supports this, do a search on your editor’s name + “Column guide”.

5. If a long line is split with an operator (eg. << or +), the operator should be placed at the beginning of the next line, not the end of the current line

```
std::cout << 3 +
4
    + 5 + 6
    * 7 * 8;
```

This helps make it clearer that subsequent lines are continuations of the previous lines, and allows you to align the operators on the left, which makes for easier reading.

6. Use whitespace to make your code easier to read by aligning values or comments or adding spacing between blocks of code.

Harder to read:

```
cost = 57;
pricePerItem =
24;
value = 5;
numberOfItems =
17;
```

Easier to read:

```
cost          =
57;
pricePerItem  =
24;
value         =
5;
numberOfItems =
17;
```

Harder to read:

```
std::cout << "Hello world!\n"; // cout lives in the iostream library
std::cout << "It is very nice to meet you!\n"; // these comments make the code hard to
read
std::cout << "Yeah!\n"; // especially when lines are different lengths
```

Easier to read:

```
std::cout << "Hello world!\n";                  // cout lives in the iostream
library
std::cout << "It is very nice to meet you!\n";  // these comments are easier to
read
std::cout << "Yeah!\n";                         // especially when all lined up
```

Harder to read:

```
// cout lives in the iostream library
std::cout << "Hello world!\n";
// these comments make the code hard to read
std::cout << "It is very nice to meet
you!\n";
// especially when all bunched together
std::cout << "Yeah!\n";
```

Easier to read:

```cpp
// cout lives in the iostream library
std::cout << "Hello world!\n";

// these comments are easier to read
std::cout << "It is very nice to meet
you!\n";

// when separated by whitespace
std::cout << "Yeah!\n";
```

We will follow these conventions throughout this tutorial, and they will become second nature to you. As we introduce new topics to you, we will introduce new style recommendations to go with those features.

Ultimately, C++ gives you the power to choose whichever style you are most comfortable with, or think is best. However, we highly recommend you utilize the same style that we use for our examples. It has been battle tested by thousands of programmers over billions of lines of code, and is optimized for success. One exception: If you are working in someone else's code base, adopt their styles. It's better to favor consistency than your preferences.

Automatic formatting

Most modern IDEs will help you format your code as you type it in (e.g. when you create a function, the IDE will automatically indent the statements inside the function body).

However, as you add or remove code, or change the IDE's default formatting, or paste in a block of code that has different formatting, the formatting can get messed up. Fixing the formatting for part or all of a file can be a headache. Fortunately, modern IDEs typically contain an automatic formatting feature that will reformat either a selection (highlighted with your mouse) or an entire file.

For Visual Studio users

In Visual Studio, the automatic formatting options can be found under Edit > Advanced > Format Document and Edit > Advanced > Format Selection.

For Code::Blocks users

In Code::Blocks, the automatic formatting options can be found under Right mouse click > Format use AStyle.

For easier access, we recommend adding a keyboard shortcut to auto-format the active file.

There are also external tools that can be used to automatically format code. clang-format (https://clang.llvm.org/docs/ClangFormat.html) is a popular one.

Best practice

Using the automatic formatting feature is highly recommended to keep your code's formatting style consistent.

Next lesson
1.9 Introduction to literals and operators

Back to table of contents

Previous lesson
1.7 Keywords and naming identifiers

B U URL INLINE CODE C++ CODE BLOCK HELP!