2.6 — Whitespace and basic formatting

BY ALEX ON JUNE 1ST, 2007 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 2ND, 2020

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
}
```

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.

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:

```
// 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*.

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

2.7 -- Forward declarations and definitions

Index

2.5 -- Why functions are useful, and how to use them effectively

C++ TUTORIAL | PRINT THIS POST

84 comments to 2.6 — Whitespace and basic formatting

« Older Comments 1 2

Ryan
February 2, 2020 at 12:20 am · Reply

Very minor typo! At the bottom of the page, it says "modern IDEs typically contains an automatic formatting feature", where 'contains' should be 'contain'.

nascardriver
February 2, 2020 at 1:17 am · Reply

Fixed, thanks!

Jimmy Hunter
January 24, 2020 at 2:01 pm · Reply

I love this tutorial!

It is so excellent, I want to help, even with this miniscule correction.

> One exception where the C++ compiler does pays attention

Should be "pay" not "pays".

You indicated in an earlier post that you fixed it. But somehow it is back.

nascardriver
January 25, 2020 at 3:17 am · Reply

Fixed for real this time, thanks!

Etienne
October 28, 2019 at 10:10 am · Reply

This is best short and sweet style guide I've read so far. Thank you! I used your style in class and the professor was very impressed.

Anastasia
August 8, 2019 at 7:09 am · Reply

Hi!
Are 'mixed' braces styles (e.g. k&r, stroustrup, "One True Brace Style", mozilla, etc.) acceptable to use? I mean, I know that it's the personal preference and consistency that count, but for those who have neither one nor the other yet?

And same question about indentation, many styles use different amount of spaces from conventional 4, if it doesn't have a big impact on readability, is it okay to use them?

And in particular, is stroustrup style (opening braces are broken from functions, attached otherwise, tabbed indent of 5 spaces) often used?

nascardriver
August 8, 2019 at 7:30 am · Reply

> Are 'mixed' braces styles [...] acceptable to use?
Yes, as long as you're consistent.

> And same question about indentation
Yes.

> is stroustrup style [...] often used?
I don't pay a lot of attention to formatting unless I'm reviewing code. I don't recall seeing this style. I'd say mixed braces and anything but 2 or 4 spaces is rather rare.

Which formatting you use during development is completely up to you. If the project you're working on enforces a specific formatting, you can still write the code how you're used to and in the end auto-format everything with the demanded settings.

Anastasia
August 8, 2019 at 8:13 am · Reply

Thank you, I know that it's not so important, just trying to set my formatter to some (more or less) acceptable style I like and stick with it. There are so many different formatting practices, it's easy to get lost.

> I don't recall seeing this style. I'd say mixed braces and anything but 2 or 4 spaces is rather rare.
That's a shame. I hoped they were more common.

ISSOtm
April 11, 2019 at 1:40 pm · Reply

About braces-on-newline being better due to an indentation mismatch, that's also provided by braces-on-same-line, example

```
int bar(int foo) {
    if(foo != 0) {
        frobble();
        return 0;
    return 1;
}
```

Indentation still makes it obvious which brace is missing. Further, IDEs have features to highlight the corresponding brace when selecting one.

For this reason, the "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." claim is in my opinion invalid.

Bob
April 5, 2019 at 10:06 am · Reply

In the comments, you state that the operator should be placed at the end of the current line, but in the actual article you state the opposite.

Which is correct?

Alex
April 7, 2019 at 10:54 am · Reply

The article reflects the more current best practice.

Louis Cloete
February 7, 2019 at 4:34 pm · Reply

Just below the first example: "Even the last two statements with the newlines in them compiles just fine."

I see only one statement with a newline in it (the last one).

Alex
February 7, 2019 at 9:11 pm · Reply

Thanks! Fixed.

Shxbl
January 5, 2019 at 12:35 pm · Reply

```
#include "pch.h"
#include <iostream>

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;
}

int main()
{
    std::cout <<  (5, 4) << std::endl;
    return 0;
}
```

why i'm getting error

nascardriver
January 6, 2019 at 5:23 am · Reply

A function can only be defined once. If all functions have different names, your program compiles.

Pork and Beans
February 28, 2018 at 10:19 am · Reply

Hello Alex!

Thank you as always for providing your amazing lessons.

I found a tiny grammatical error that I'm sure you'll like to correct. It is near the beginning of this lesson, after the second code sample in the following text:

"One exception where the C++ compiler does pays attention..."

It should be "pay" instead of "pays".

I feel that you would have wanted to know this.

May you be right where you want to be.

Have a great day.

Alex
February 28, 2018 at 4:26 pm · Reply

Fixed. Thanks!

Linyuan
February 5, 2018 at 6:40 am · Reply

I love this Tutorial from this website. I've been looking for more than 20+ different books and webs that teaching people how to code in C++.

and this is the best i found until now.

Tyler
January 10, 2018 at 2:42 pm · Reply

I've been coding for about 15 years now, reading up these pages as a refresher on C++ as I have not used it in several years. I have to say I very strongly disagree with Basic Formatting #1. I'm an extremely adamant believer in "tabs for indentation, spaced for alignment". Your reasoning for using spaces was to allow the indentation to stay the same between editors, there is no reason for this to be an advantage, maintaining alignment with code like that displayed in Basic Formatting #4 would not suffer from using tabs, you'd use a tab to indent to the scope level, then spaces to align further, in the first case for example for Basic Formatting #4 it double be 1 tab and then 4 spaces. Changing the tab size will then still maintain alignment and this way has many advantages over using only spaces that I would prefer not spend hours writing about.

Honestly most of the people who argue for exclusively using spaces almost always come off as overly pretentious, and many of the most popular arguments for using spaces exclusively are plain and simply false or work with tabs as well. Instead of coming off as a pretentious coder version of a grammar nazi, at least discuss the alternatives instead of forcing users down a route that is highly debated. If pure space coding was entirely superior, there would be no argument, the fact that there is shows immediately that there are benefits to the other that are being ignored. Coding with spaces only is literally one of the most painful times coding I've ever dealt with, the pure added tedium dealing with the negative consequences of a system based on using more characters to solve a trivial problem is enough that my productivity goes down approximately 30% when dealing with using pure spaces. I'm more productive with keyboard navigation through code and having 24 spaces versus a handful of tabs and then a handful of spaces for additional alignment is just about the most irritating thing in the entirety of programming. I dislike dealing with 4 or 5 levels of indentation of spaces more than I get annoyed at debugging memory allocation/deallocation problems.

Alex
January 10, 2018 at 5:59 pm · Reply

To be clear, I'm not trying to advocate a position in the eternal tabs vs spaces debate. Personally, I use tabs, but many companies and style guides recommend spaces.

I've rewritten recommendation 1 to try to make this a little more clear.

Porter
April 29, 2017 at 4:16 pm · Reply

I find it more readable to do:

```
int main() {
return 0;
}
```

Rather than:

```
int main()
{
return 0;
}
```

Just a personal preference.

Alex
April 29, 2017 at 8:54 pm · Reply

A lot of people agree with you because that style is more compact, and you can fit more code on the screen.

Personally, I like having my opening and closing braces at the same level of indentation. It makes it easier to see the indentation structure of a function or class and generally makes reading more pleasant.

It's less important which style you choose/prefer and more important that you're consistent.

Richard
April 14, 2017 at 7:22 pm · Reply

BASIC FORMATTING point one, paragraph two. Do you have the words 'space' and 'tab' the wrong way around?

Alex
April 16, 2017 at 11:58 am · Reply

Nope it's correct as written. Most IDEs treat a tab as 3 or 4 spaces, but let you configure it to whatever you like.

Georges Theodosiou
January 19, 2017 at 7:07 am · Reply

My dear c++ Teacher,
Please comment following comment:
Regarding 2nd easier to read example, I see more easier following:

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

Indeed comments are below statements.
Is it disadvantage?
With regards and friendship.

Alex
January 19, 2017 at 11:33 am · Reply

Generally, comments not placed on the same line as the code are placed before the line, not after. This is because the comment normally provides some sort of enlightening information about the line to come -- and you want to know that information before you try and understand the line, not after.

I've never seen anybody place comments after the lines. That's not to say it never happens, but it's certainly not best practice, and I would not recommend it.

Georges Theodosiou
January 18, 2017 at 6:05 am · Reply

My dear c++ Teacher,
Please let me say that in "Basic formatting", 4th recommendation, in program, in 2nd statement, text is not aligned with the previous line. In the following program is aligned:

```
#include <iostream>
int main()
{
    std::cout << "This is a really, really, really, really, really, really, really, " <<
        "really long line" << std::endl; // one extra indentation for continuation line

    std::cout << "This is another really, really, really, really, really, really, really, "
                 "really long line" << std::endl; // text aligned with the previous line fo

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

Also in 5th recommendation, in program, edl are not preceded by std::.
With regards and friendship.

Alex
January 18, 2017 at 4:43 pm · Reply

Thanks for pointing out these discrepancies. They should be fixed now.

Georges Theodosiou
January 18, 2017 at 5:38 am · Reply

My dear c++ Teacher,
please let me send you program with your second snippet and output capability.

```
#include <iostream>

int add1(int x, int y) { return x + y; }

int add2(int x, int y) {
    return x + y; }

int add3(int x, int y)
{    return x + y; }

int add4(int x, int y)
{
    return x + y;
}
int main()
{
    std::cout << add1(1,2) << std::endl;
    std::cout << add2(3,4) << std::endl;
    std::cout << add3(5,6) << std::endl;
    std::cout << add4(7,8) << std::endl;
    return 0;
}
```

With regards and friendship.

Leander
November 29, 2016 at 9:01 am · Reply

Love these sort of guidelines. As a new programmer it's hard to tell what is considered good practice and what is frowned upon even if it won't prevent your program from functioning correctly.

« Older Comments 1 2