1.2 — Comments

A **comment** is a programmer-readable note that is inserted directly into the source code of the program. Comments are ignored by the compiler and are for the programmer's use only.

In C++ there are two different styles of comments, both of which serve the same purpose: to help programmers document the code in some way.

Single-line comments

The `//` symbol begins a C++ single-line comment, which tells the compiler to ignore everything from the `//` symbol to the end of the line. For example:

```
std::cout << "Hello world!"; // Everything from here to the end of the line is
ignored
```

Typically, the single-line comment is used to make a quick comment about a single line of code.

```
std::cout << "Hello world!\n"; // std::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
```

Having comments to the right of a line can make both the code and the comment hard to read, particularly if the line is long. If the lines are fairly short, the comments can simply be aligned (usually to a tab stop), like so:

```
std::cout << "Hello world!\n";                // std::cout lives in the iostream
library
std::cout << "It is very nice to meet you!\n"; // this is much easier to read
std::cout << "Yeah!\n";                       // don't you think so?
```

However, if the lines are long, placing comments to the right can make your lines really long. In that case, single-line comments are often placed above the line it is commenting:

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

// this is much easier to read
std::cout << "It is very nice to meet you!\n";

// don't you think so?
std::cout << "Yeah!\n";
```

Author's note

In this tutorial series, our examples fall into one of the following categories:

- Full programs (those with a `main()` function). These are ready to be compiled and run.
- Snippets (small pieces) of code, such as the statements above. We use these to demonstrate specific concepts in a concise manner.

We don't intend for you to compile snippets. But if you'd like to, you'll need to turn them into a full program. Typically, that program will look something like this:

```
#include <iostream>

int main()
{
    // Replace this line with the snippet(s) of code you'd like to compile

    return 0;
}
```

Multi-line comments

The `/*` and `*/` pair of symbols denotes a C-style multi-line comment. Everything in between the symbols is ignored.

```
/* This is a multi-line comment.
   This line will be ignored.
   So will this one. */
```

Since everything between the symbols is ignored, you will sometimes see programmers "beautify" their multi-line comments:

```
/* This is a multi-line comment.
 * the matching asterisks to the left
 * can make this easier to read
 */
```

Multi-line style comments can not be nested. Consequently, the following will have unexpected results:

```
/* This is a multi-line /* comment */ this is not inside the comment */
// The above comment ends at the first */, not the second */
```

When the compiler tries to compile this, it will ignore everything from the first `/*` to the first `*/`. Since `this is not inside the comment */` is not considered part of the comment, the compiler will try to compile it. That will inevitably result in a compile error.

This is one place where using a syntax highlighter can be really useful, as the different coloring for comment should make clear what's considered part of the comment vs not.

Warning

Don’t use multi-line comments inside other multi-line comments. Wrapping single-line comments inside a multi-line comment is okay.

Proper use of comments

Typically, comments should be used for three things. First, for a given library, program, or function, comments are best used to describe *what* the library, program, or function, does. These are typically placed at the top of the file or library, or immediately preceding the function. For example:

```
// This program calculates the student's final grade based on their test and homework
scores.

// This function uses Newton's method to approximate the root of a given equation.

// The following lines generate a random item based on rarity, level, and a weight
factor.
```

All of these comments give the reader a good idea of what the library, program, or function is trying to accomplish without having to look at the actual code. The user (possibly someone else, or you if you’re trying to reuse code you’ve previously written) can tell at a glance whether the code is relevant to what he or she is trying to accomplish. This is particularly important when working as part of a team, where not everybody will be familiar with all of the code.

Second, *within* a library, program, or function described above, comments can be used to describe *how* the code is going to accomplish its goal.

```
/* To calculate the final grade, we sum all the weighted midterm and homework scores
    and then divide by the number of scores to assign a percentage, which is
    used to calculate a letter grade. */

// To generate a random item, we're going to do the following:
// 1) Put all of the items of the desired rarity on a list
// 2) Calculate a probability for each item based on level and weight factor
// 3) Choose a random number
// 4) Figure out which item that random number corresponds to
// 5) Return the appropriate item
```

These comments give the user an idea of how the code is going to accomplish its goal without having to understand what each individual line of code does.

Third, at the statement level, comments should be used to describe *why* the code is doing something. A bad statement comment explains *what* the code is doing. If you ever write code that is so complex that needs a comment to explain *what* a statement is doing, you probably need to rewrite your statement, not comment it.

Here are some examples of bad line comments and good statement comments.

Bad comment:

```
// Set sight range to 0
sight = 0;
```

Reason: We already can see that sight is being set to 0 by looking at the statement

Good comment:

```
// The player just drank a potion of blindness and can not see anything
sight = 0;
```

Reason: Now we know why the player's sight is being set to 0

Bad comment:

```
// Calculate the cost of the items
cost = quantity * 2 * storePrice;
```

Reason: We can see that this is a cost calculation, but why is quantity multiplied by 2?

Good comment:

```
// We need to multiply quantity by 2 here because they are bought in pairs
cost = quantity * 2 * storePrice;
```

Reason: Now we know why this formula makes sense.

Programmers often have to make a tough decision between solving a problem one way, or solving it another way. Comments are a great way to remind yourself (or tell somebody else) the reason you made one decision instead of another.

Good comments:

```
// We decided to use a linked list instead of an array because
// arrays do insertion too slowly.

// We're going to use Newton's method to find the root of a number because
// there is no deterministic way to solve these equations.
```

Finally, comments should be written in a way that makes sense to someone who has no idea what the code does. It is often the case that a programmer will say "It's obvious what this does! There's no way I'll forget about this". Guess what? It's *not* obvious, and you *will* be amazed how quickly you forget. :) You (or someone else) will thank you later for writing down the what, how, and why of your code in human language. Reading individual lines of code is easy. Understanding what goal they are meant to accomplish is not.

Related content

We discuss commenting for variable declaration statements in lesson 1.7 -- Keywords and naming identifiers.

Best practice

Comment your code liberally, and write your comments as if speaking to someone who has no idea what the code does. Don't assume you'll remember why you made specific choices.

Author's note

Throughout the rest of this tutorial series, we'll use comments inside code blocks to draw your attention to specific things, or help illustrate how things work (while ensuring the programs still compile). Astute readers will note that by the above standards, most of these comments are horrible. :) As you read through the rest of the tutorials, keep in mind that the comments are serving an intentional educational purpose, not trying to demonstrate what good comments look like.

Commenting out code

Converting one or more lines of code into a comment is called **commenting out** your code. This provides a convenient way to (temporarily) exclude parts of your code from being included in your compiled program.

To comment out a single line of code, simply use the // style comment to turn a line of code into a comment temporarily:

Uncommented out:

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

Commented out:

```
//    std::cout << 1;
```

To comment out a block of code, use // on multiple lines of code, or the /* */ style comment to turn the block of code into a comment temporarily.

Uncommented out:

```
std::cout << 1;
std::cout << 2;
std::cout << 3;
```

Commented out:

```
//    std::cout << 1;
//    std::cout << 2;
//    std::cout << 3;
```

or

```
/*
    std::cout << 1;
    std::cout << 2;
    std::cout << 3;
*/
```

There are quite a few reasons you might want to do this:

1. You're working on a new piece of code that won't compile yet, and you need to run the program. The compiler won't let you compile the code if there are compiler errors. Commenting out the code that won't compile will allow the program to compile so you can run it. When you're ready, you can uncomment the code, and continue working on it.
2. You've written new code that compiles but doesn't work correctly, and you don't have time to fix it until later. Commenting out the broken code will ensure the broken code doesn't execute and cause problems until you can fix it.
3. To find the source of an error. If a program isn't producing the desired results (or is crashing), it can sometimes be useful to disable parts of your code to see if you can isolate what's causing it to not work correctly. If you comment out one or more lines of code, and your program starts working as expected (or stops crashing), odds are whatever you last commented out was part of the problem. You can then investigate why those lines of code are causing the problem.
4. You want to replace one piece of code with another piece of code. Instead of just deleting the original code, you can comment it out and leave it there for reference until you're sure your new code works properly. Once you are sure your new code is working, you can remove the old commented out code. If you can't get your new code to work, you can always delete the new code and uncomment the old code to revert to what you had before.

Commenting out code is a common thing to do while developing, so many IDEs provide support for commenting out a highlighted section of code. How you access this functionality varies by IDE.

For Visual Studio users

You can comment or uncomment a selection via Edit menu > Advanced > Comment Selection (or Uncomment Selection).

For Code::Blocks users

You can comment or uncomment a selection via Edit menu > Comment (or Uncomment, or Toggle comment, or any of the other comment tools).

For VS Code users

You can comment out a selection by pressing ctrl-k-c, and uncomment out a selection by pressing ctrl-k-u.

Tip

If you always use single line comments for your normal comments, then you can always use multi-line comments to comment out your code without conflict. If you use multi-line comments to document your code, then commenting-out code using comments can become more challenging.

If you do need to comment out a code block that contains multi-line comments, you can also consider using the `#if 0` preprocessor directive, which we discuss in lesson 2.10 -- Introduction to the preprocessor.

Summary

- At the library, program, or function level, use comments to describe *what*.
- Inside the library, program, or function, use comments to describe *how*.
- At the statement level, use comments to describe *why*.