11.5 — Default arguments

ALEX[1] NOVEMBER 10, 2024

A **default argument** is a default value provided for a function parameter. For example:

```
void print(int x, int y=10) // 10 is the default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

When making a function call, the caller can optionally provide an argument for any function parameter that has a default argument. If the caller provides an argument, the value of the argument in the function call is used. If the caller does not provide an argument, the value of the default argument is used.

Consider the following program:

```
#include <iostream>

void print(int x, int y=4) // 4 is the default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    print(1, 2); // y will use user-supplied argument 2
    print(3); // y will use default argument 4, as if we had called print(3, 4)

    return 0;
}
```

This program produces the following output:

```
x: 1
y: 2
x: 3
y: 4
```

In the first function call, the caller supplied explicit arguments for both parameters, so those argument values are used. In the second function call, the caller omitted the second argument, so the default value of `4` was used.

Note that you must use the equals sign to specify a default argument. Using parenthesis or brace initialization won't work:

```
void foo(int x = 5);   // ok
void goo(int x ( 5 )); // compile error
void boo(int x { 5 }); // compile error
```

Perhaps surprisingly, default arguments are handled by the compiler at the call site. In the above example, when the compiler sees `print(3)`, it will rewrite this function call as `print(3, 4)`, so that the number of arguments matches the number of parameters. The rewritten function call then works as per usual.

Key insight

Default arguments are inserted by the compiler at site of the function call.

Default arguments are frequently used in C++, and you'll see them a lot in code you encounter (and in future lessons).

When to use default arguments

Default arguments are an excellent option when a function needs a value that has a reasonable default value, but for which you want to let the caller override if they wish.

For example, here are a couple of function prototypes for which default arguments might be commonly used:

```
int rollDie(int sides=6);
void openLogFile(std::string filename="default.log");
```

Author's note

Because the user can choose whether to supply a specific argument value or use the default value, a parameter with a default value provided is sometimes called an **optional parameter**. However, the term *optional parameter* is also used to refer to several other types of parameters (including parameters passed by address, and parameters using `std::optional`), so we recommend avoiding this term.

Default arguments are also useful in cases where we need to add a new parameter to an existing function. If we add a new parameter without a default argument, it will break all existing function calls (which aren't supplying an argument for that parameter). This can result in a lot of updating of existing function calls (and may not even be possible if you don't own the calling code). However, if we add a new parameter with a default argument instead, all existing function calls will still work (as they will use the default argument for the parameter), while still letting new calls to the function specify an explicit argument if desired.

Multiple default arguments

A function can have multiple parameters with default arguments:

```cpp
#include <iostream>

void print(int x=10, int y=20, int z=30)
{
    std::cout << "Values: " << x << " " << y << " " << z << '\n';
}

int main()
{
    print(1, 2, 3); // all explicit arguments
    print(1, 2); // rightmost argument defaulted
    print(1); // two rightmost arguments defaulted
    print(); // all arguments defaulted

    return 0;
}
```

The following output is produced:

```
Values: 1 2 3
Values: 1 2 30
Values: 1 20 30
Values: 10 20 30
```

C++ does not (as of C++23) support a function call syntax such as `print(,,3)` (as a way to provide an explicit value for `z` while using the default arguments for `x` and `y`. This has three major consequences:

1. In a function call, any explicitly provided arguments must be the leftmost arguments (arguments with defaults cannot be skipped).

For example:

```cpp
void print(std::string_view sv="Hello", double d=10.0);

int main()
{
    print();           // okay: both arguments defaulted
    print("Macaroni"); // okay: d defaults to 10.0
    print(20.0);       // error: does not match above function (cannot skip argument
for sv)

    return 0;
}
```

2. If a parameter is given a default argument, all subsequent parameters (to the right) must also be given default arguments.

The following is not allowed:

```cpp
void print(int x=10, int y); // not allowed
```

Rule

If a parameter is given a default argument, all subsequent parameters (to the right) must also be given default arguments.

3. If more than one parameter has a default argument, the leftmost parameter should be the one most likely to be explicitly set by the user.

Default arguments can not be redeclared, and must be declared before use

Once declared, a default argument can not be redeclared in the same translation unit. That means for a function with a forward declaration and a function definition, the default argument can be declared in either the forward declaration or the function definition, but not both.

```
#include <iostream>

void print(int x, int y=4); // forward declaration

void print(int x, int y=4) // compile error: redefinition of default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

The default argument must also be declared in the translation unit before it can be used:

```
#include <iostream>

void print(int x, int y); // forward declaration, no default argument

int main()
{
    print(3); // compile error: default argument for y hasn't been defined yet

    return 0;
}

void print(int x, int y=4)
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

The best practice is to declare the default argument in the forward declaration and not in the function definition, as the forward declaration is more likely to be seen by other files and included before use (particularly if it's in a header file).

in foo.h:

```
#ifndef FOO_H
#define FOO_H
void print(int x, int y=4);
#endif
```

in main.cpp:

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

void print(int x, int y)
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    print(5);

    return 0;
}
```

Note that in the above example, we're able to use the default argument for function `print()` because *main.cpp* #includes *foo.h*, which has the forward declaration that defines the default argument.

Best practice

If the function has a forward declaration (especially one in a header file), put the default argument there. Otherwise, put the default argument in the function definition.

Default arguments and function overloading

Functions with default arguments may be overloaded. For example, the following is allowed:

```
#include <iostream>
#include <string_view>

void print(std::string_view s)
{
    std::cout << s << '\n';
}

void print(char c = ' ')
{
    std::cout << c << '\n';
}

int main()
{
    print("Hello, world"); // resolves to print(std::string_view)
    print('a');            // resolves to print(char)
    print();               // resolves to print(char)

    return 0;
}
```

The function call to `print()` actually calls `print(char)`, which acts as if the user had explicitly called `print(' ')`.

Now consider this case:

```
void print(int x);                  // signature print(int)
void print(int x, int y = 10);      // signature print(int, int)
void print(int x, double y = 20.5); // signature print(int, double)
```

Default values are not part of a function's signature, so these function declarations are differentiated overloads.

Related content

We discuss function overload differentiation in lesson 11.2 -- Function overload differentiation (https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/)[2]

Default arguments can lead to ambiguous matches

Default arguments can easily lead to ambiguous function calls:

```
void foo(int x = 0)
{
}

void foo(double d = 0.0)
{
}

int main()
{
    foo(); // ambiguous function call

    return 0;
}
```

In this example, the compiler can't tell whether `foo()` should resolve to `foo(0)` or `foo(0.0)`.

Here's a slightly more complex example:

```
void print(int x);                  // signature print(int)
void print(int x, int y = 10);      // signature print(int, int)
void print(int x, double y = 20.5); // signature print(int, double)

int main()
{
    print(1, 2);   // will resolve to print(int, int)
    print(1, 2.5); // will resolve to print(int, double)
    print(1);      // ambiguous function call

    return 0;
}
```

For the call `print(1)`, the compiler is unable to tell whether this resolve to `print(int)`, `print(int, int)`, or `print(int, double)`.

In the case where we mean to call `print(int, int)` or `print(int, double)` we can always explicitly specify the second parameter. But what if we want to call `print(int)`? It's not obvious how we can do so.

Default arguments don't work for functions called through function pointers

Advanced

We cover this topic in lesson 20.1 -- Function Pointers (https://www.learncpp.com/cpp-tutorial/function-pointers/)[3]. Because default arguments are not considered using this method, this also provides a workaround to call a function that would otherwise be ambiguous due to default arguments.

Next lesson
11.6 Function templates

4

Back to table of contents

5

Previous lesson
11.4 Deleting functions

6

7

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

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[9] are connected to your provided email address.

130 COMMENTS

Newest

Diddy
June 20, 2025 3:04 am PDT

WOW!

0 Reply

vitrums
March 21, 2025 3:29 am PDT

In the paragraph praising default arguments as a convenient mechanism for an introduction of the new parameters without breaking the existing code you say: "This can result in a lot of updating of existing

function calls (and may not even be possible if you don't own the calling code)" . I think it's in contradiction with the following quote: "Default arguments are inserted by the compiler at site of the function call" . It makes perfect sense that we don't have to modify the existing **source code**. It makes no sense however that linker can continue to use the old **object files**. Unless by "compiler" you mentioned it in a broad sense, and hence the linker is perfectly capable of doing such arguments substitution. Can you clarify this moment?

0 Reply

here4cpp

December 25, 2024 7:40 am PST

I have a question here.

I know that if I have something like

```
int f(int x);
float f(int x);
```

then obviously it will show a compilation error due to ambiguity after function overloading.

but we see that in the examples like the last one you showed here:

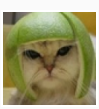
```
void print(int x);                  // signature print(int)
void print(int x, int y = 10);      // signature print(int, int)
void print(int x, double y = 20.5); // signature print(int, double)

int main()
{
    print(1, 2);   // will resolve to print(int, int)
    print(1, 2.5); // will resolve to print(int, double)
    print(1);      // ambiguous function call

    return 0;
}
```

So in this case, even though the three print functions do have ambiguity, but since its not obvious, as long as a call is not made which causes ambiguity, until then the c++ compiler (like g++ that I've used till now) does not show any error and just compiles the ambiguous things without trying to explicitly check for exceptional ambiguous cases.

How come it doesn't even give any warning on such contexts? I would like to believe that if a compiler is capable of finding non-complete branching in if-else and give a warning in case of return in functions, then it should be able to check things like these with ease right? After all, I think if-else branching seems somewhat complicated.

I want to know about the compiler behavior behind this.

0 Reply



Alex Author

Reply to here4cpp [10] January 1, 2025 10:28 pm PST

Probably because the C++ standard doesn't require it, and the compiler authors didn't think it was worth the time to detect and warn against.

There are lots of other cases in C++ where you can do something that's potentially ambiguous (or even outright incorrect) and the compiler will only warn you or error out when you actually try to invoke that thing. It simplifies things if the compiler only has to check what you are doing, not what you could be doing.

2 Reply

here4cpp

Reply to Alex [11] January 2, 2025 2:42 am PST

Thanks, got it!

0 Reply

NordicPeace

December 9, 2024 11:34 pm PST

This question could be quite different from the rest : Which is the best practice to make your code clutter-less - a) Forward declaring all functions and letting (int main) at the top, b) letting all the functions at the top and then finally calling them in (int main), c) keeping a separate file (main.cpp) and defining all implementations in (other.cpp/others.h).

I am talking about bigger programs with a lot of functions and (if,else statements).
I do want to hear your thoughts on smaller programs as well like for instance a simple - lo-hi game which prolly not required a separate file.

0 Reply

Alex Author

Reply to NordicPeace [12] December 14, 2024 1:35 pm PST

Generally, I see:

- main.cpp: contains the main() function (typically at the bottom), initialization functions to set up the application state and 3rd party libraries, and main application loop.
- everything else in .cpp/.h files

For small programs you may not need the .cpp/.h files.

0 Reply

NordicCat

Reply to Alex [13] January 13, 2025 5:14 am PST

What does main application loop means it terms of a game? (pls give me a simple example)

0 Reply

NordicPeace

Reply to Alex [13] December 14, 2024 11:03 pm PST

thank you!

0 Reply

NordicPeace

Reply to NordicPeace [12] December 9, 2024 11:49 pm PST

https://softwareengineering.stackexchange.com/questions/129327/how-to-keep-a-big-and-complex-software-product-maintainable-over-the-years

I have searched it on the internet and found this, I guess it's unrelated to my problem.

0 Reply

ahj jh

November 21, 2024 4:11 pm PST

```
1 | void print(int x, int y=4) // compile error: redefinition of default argument
```

My brother you are missing a semi-colon.

0 Reply

Alex Author

Reply to ahj jh [14] November 25, 2024 4:19 pm PST

Nope, this is not a forward declaration (which ends in a semicolon) -- it is the header for the function definition, and the function definition follows starting on the next line.

0 Reply

Vitalik

August 15, 2024 12:55 pm PDT

If the function has a forward declaration (especially one in a header file), put the default argument there. Otherwise, put the default argument in the function definition.

Perhaps we should just leave the advice that it is better to define the default value in the .h file?

It's just that you wrote before that it's better to do it in .h, but you give both options as the best option :).

0 Reply

Alex Author

Reply to Vitalik [15] August 17, 2024 10:01 pm PDT

Then readers will be confused what to do when a forward declaration is declared in a code file. :)

2 Reply

Shakir
July 1, 2024 7:27 pm PDT

Regarding one of the final sections:

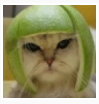
" Default values are not part of a function's prototype, so the above functions are all considered distinct (meaning the above will compile). However, such functions can lead to potentially ambiguous function calls. "

I know that most people would understand this, but as I was a bit rusty/slow I was a bit confused. At first I had the misconception that this means that the function would differentiate to print(int x) only.

At the risk of excess verbosity, perhaps you could link explicitly to Chapter 11.2 as a reminder of how functions are differentiated? I admit this is probably a niche problem.

Last edited 11 months ago by Shakir

0 Reply



Alex Author
Reply to Shakir [16] July 4, 2024 1:57 pm PDT

Linked.

0 Reply

Old Coder
May 12, 2024 11:14 pm PDT

Alex, under the heading **Default arguments can not be redeclared**, you write that

"Once declared, a default argument can not be redeclared (in the same file). That means for a function with a forward declaration and a function definition, the default argument can be declared in either the forward declaration or the function definition, but not both."

Before I'd read that far, I tried essentially the same thing that @Vlad did earlier (declaring the default in a definition, but trying to use the default earlier in a source file than the definition appears). GCC (v10.2.1) failed to compile.

I'd suggest a rewording to clarify that a default put in the function definition can't be used unless it appears earlier in the source file than the code which tries to use the default value.

0 Reply

Alex Author
Reply to Old Coder [17] May 13, 2024 12:06 pm PDT

Section amended to include this point. Thanks for the feedback!

0 Reply

Swaminathan R

May 4, 2024 12:41 pm PDT

Default arguments don't work for functions called through function pointers

Isn't this topic better placed in https://www.learncpp.com/cpp-tutorial/function-pointers/ ? Because no one knows what a pointer is yet, so people will skip this anyway, and by the time they know pointers, they probably won't even think of this possibility.

0 Reply

Alex Author

Reply to Swaminathan R [18] May 8, 2024 12:00 pm PDT

Agreed. Moved to the lesson on function pointers. Thanks for the feedback.

0 Reply

Phargelm

April 4, 2024 8:36 am PDT

> When the compiler encounters a call to a function with one or more default arguments, it rewrites the function call to include the default arguments. This process happens at compile-time, and thus **can only be applied to functions that are called at compile time**.

But functions that are called at compile time should be either constexpr functions, or consteval functions. Or I misunderstand the phrase above.

1 Reply

Alex Author

Reply to Phargelm [19] April 4, 2024 2:11 pm PDT

Changed wording to "and thus can only be applied to functions that can be resolved at compile time."

1 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/function-overload-differentiation/
3. https://www.learncpp.com/cpp-tutorial/function-pointers/

4. https://www.learncpp.com/cpp-tutorial/function-templates/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/deleting-functions/
7. https://www.learncpp.com/default-arguments/
8. https://www.learncpp.com/cpp-tutorial/introduction-to-function-overloading/
9. https://gravatar.com/
10. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-605693
11. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-606090
12. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-605075
13. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-605253
14. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-604397
15. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-600919
16. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-599108
17. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-597038
18. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-596679
19. https://www.learncpp.com/cpp-tutorial/default-arguments/#comment-595427
20. https://g.ezoic.net/privacy/learncpp.com