7.7 — Default arguments

BY ALEX ON AUGUST 6TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

A **default argument** is a default value provided for a function parameter. If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used. If the user does supply an argument for the parameter, the user-supplied argument is used.

Because the user can choose whether to supply a specific argument value, or use the default, a parameter with a default value provided is often called an **optional parameter**.

Consider the following program:

```
1  void printValues(int x, int y=10) // 10 is the default argument, y is now an optional paramete
2  {
3      std::cout << "x: " << x << '\n';
4      std::cout << "y: " << y << '\n';
5  }
6
7  int main()
8  {
9      printValues(1); // y will use default argument 10
10     printValues(3, 4); // y will use user-supplied argument 4
11 }
```

This program produces the following output:

```
x: 1
y: 10
x: 3
y: 4
```

In the first function call, the caller did not supply an argument for y, so the function used the default value of 10. In the second call, the caller did supply a value for y, so the user-supplied value was used.

Default arguments are an excellent option when the function needs a value that the user may or may not want to override. For example, here are a few function prototypes for which default arguments might be commonly used:

```
1  void openLogFile(std::string filename="default.log");
2  int rollDie(int sides=6);
3  void printStringInColor(std::string str, Color color=COLOR_BLACK); // Color is an enum
```

Multiple default arguments

A function can have multiple default arguments:

```
1  void printValues(int x=10, int y=20, int z=30)
2  {
3      std::cout << "Values: " << x << " " << y << " " << z << '\n';
4  }
```

Given the following function calls:

```
1  printValues(1, 2, 3);
2  printValues(1, 2);
3  printValues(1);
4  printValues();
```

The following output is produced:

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

Note that it is impossible to supply an argument for parameter z without also supplying arguments for parameters x and y. This is because C++ does not support a function call syntax such as `printValues(,,3)`. This has two major consequences:

1) All default arguments must be for the rightmost parameters. The following is not allowed:

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

2) If more than one default argument exists, the leftmost default argument should be the one most likely to be explicitly set by the user.

Default arguments can only be declared once

Once declared, a default argument can not be redeclared. 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.

```
void printValues(int x, int y=10);

void printValues(int x, int y=10) // error: redefinition of default argument
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

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 (particularly if it's in a header file).

in foo.h:

```
#ifndef FOO_H
#define FOO_H
void printValues(int x, int y=10);
#endif
```

in main.cpp:

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

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

int main()
{
    printValues(5);

    return 0;
}
```

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

Rule: If the function has a forward declaration, put the default argument there. Otherwise, put them in the function definition.

Default arguments and function overloading

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

```
1 void print(std::string string);
2 void print(char ch=' ');
```

If the user were to call `print()`, it would resolve to `print(' ')`, which would print a space.

However, it is important to note that optional parameters do NOT count towards the parameters that make the function unique. Consequently, the following is not allowed:

```
1 void printValues(int x);
2 void printValues(int x, int y=20);
```

If the caller were to call `printValues(10)`, the compiler would not be able to disambiguate whether the user wanted printValues(int) or printValues(int, 20) with the default value.

Summary

Default arguments provide a useful mechanism to specify parameters that the user may optionally provide values for. They are frequently used in C++, and you'll see them a lot in future lessons.


7.8 -- Function Pointers


Index


7.6 -- Function overloading

C++ TUTORIAL | PRINT THIS POST

50 comments to 7.7 — Default arguments

terapty
February 5, 2020 at 8:08 am · Reply

>However, it is important to note that optional parameters do NOT count towards the parameters that make the function unique.

Just as a sidenote, forward declaring or defining functions with signatures that only differ by the TYPE of last default parameters is still ambiguous.
And it makes sense, the compiler should tell you it's impossible to make any call to the function unambiguously if you only specify the non-"defaultable" arguments when you call it (the compiler would have to "guess" which overloaded function you're trying to call).

Example, just to clarify...

In foo.h:

```
#ifndef FOO_H
#define FOO_H
    void printValues(int x, char c='c');
    void printValues(int x, int y=10);
#endif
```

In foo.c:

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

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

void printValues(int x, char c) {
    std::cout << "x: " << x << '\n';
    std::cout << "c: " << c << '\n';
}

int main() {
    printValues(5);
    return 0;
}
```

Compiling with gcc version 7.4.0 :
foo.c:15:18: error: call of overloaded ‘printValues(int)’ is ambiguous

Then it procedes to list all ambiguous candidates (in all the different headers that declare the function I suppose, but haven't tested that out).
I think (if the author intends to update this page with this) it should be added to the last section that deals with function overloading.

P.S. : This site is awesome. Well-written, lots of technical detail, there's also some basics on how to use the standard library and even some custom-made implementations of known algorithms.
It not only gives you the tools to get started programming fast, it also gives you lots of C++-specific details to make sure you leverage all of its benefits and avoid its pitfalls! Nice work!

Samira Ferdi
August 26, 2019 at 7:16 pm · Reply

Hi, Alex and Nascardriver!

This is if string passed by value

```
1 void openLogFile(std::string filename="default.log");
```

But, is this if I passed string by reference?

```
1 void openLogFile(const std::string &filename = "default.log");
```

nascardriver
August 27, 2019 at 3:53 am · Reply

Yes, that's what the `&` means.

Red Lightning
April 26, 2019 at 5:44 am · Reply

"A default argument is an default value provided for a function parameter. If the user does not supply an explicit argument for a parameter with a default argument, the default value will be used. If the user does supply an argument for the parameter, the user-supplied argument is used."
Syntax error: "a default" instead of "an default"

Alex
April 28, 2019 at 4:29 pm · Reply

Fixed! Thanks for pointing this out.

Zha Weihua
April 4, 2019 at 5:01 pm · Reply

Default arguments can only be declared once

Once declared, a default argument can not be redeclared. 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.

About this conclusion, I saw an inverse example. May I think it's a C++ defect and we should avoid to use it?

refer to
https://stackoverflow.com/questions/44818513/c-adding-and-redefinition-of-default-arguments-in-real-world

```
#include <iostream>

void foo(int a, int b, int c = -1)
{
   std::cout << "foo(" << a << ", " << b << ", " << c << ")\n";
}

int main()
{
   foo(1, 2);   // output: foo(1, 2, -1)
```

```
    // error: does not use default from surrounding scope
    //void foo(int a, int b = 0, int c);

    void foo(int a, int b, int c = 30);
    foo(1, 2);   // output: foo(1, 2, 30)

    // error: we cannot redefine the argument in the same scope
    // void foo(int a, int b, int c = 35);

    // has a default argument for c from a previous declaration
    void foo(int a, int b = 20, int c);
    foo(1);      // output: foo(1, 20, 30)

    void foo(int a = 10, int b, int c);
    foo();       // output: foo(10, 20, 30)

    {
        // in inner scopes we can completely redefine them
        void foo(int a, int b = 4, int c = 8);
        foo(2);  // output: foo(2, 4, 8)
    }

    return 0;
}
```

nascardriver
April 5, 2019 at 1:35 am · Reply

Hi!

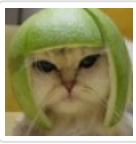
This is not a defect, it's explicitly allowed.
"[...] default arguments can be added in later declarations of a function in the same scope."

I agree, this should be avoided as it requires a reader to read the preceding code to understand a function call (As opposed to reading a single declaration).

nascardriver
March 15, 2019 at 11:14 am · Reply

Hi Alex!

They're called default arguments, not parameters.



Alex
March 19, 2019 at 4:53 pm · Reply

Indeed. Nomenclature changed. Thanks for pointing out the inaccuracy.

magaji::hussaini
December 22, 2018 at 1:48 am · Reply

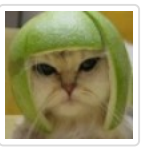
Hi alex I think a note should added that only copy initialisation work with default parameters because I got a compile time error when I tried uniform/direct initialisations.

```
1 void printValues(int x, int y{5}); //invalid
2
3 void printValues(int x, int y(5)); //also not valid
```

```
4
5 void printValues(int x, int y=5);  //the only valid syntax
```

ST.James
August 2, 2018 at 5:19 pm · Reply

In the challenge section in lesson 6 (the blackjack game), we're almost sure that each time we call shuffleDeck, we will be passing in a deck of cards (&deck); how do we make "&deck" a default parameter?



Alex
August 2, 2018 at 9:42 pm · Reply

You can't. Default parameters must be compile-time constants, and deck is a variable.

Peter Baum
May 11, 2018 at 12:26 pm · Reply

1. Is there any good reason why C++ does not support more flexible default values such as

```
1  printValues(,,3)
```

and

```
1 void printValue(int x=10, int y);
```

?

2. It seems strange that “Once declared, a default parameter can not be redeclared.” even when the declarations are exactly the same. Is there any good reason why this is not allowed?

3. In the function overload example:

```
1 void printValues(int x);
2 void printValues(int x, int y=20);
```

the statement is made that “... the compiler would not be able to disambiguate...” such a situation. Note that it could easily do this if the requirement is to write

```
1 printValues(10,)
```

if the second function were desired.

nascardriver
May 13, 2018 at 2:09 am · Reply

Hi Peter!

1. I don't think there is. The boost library has a feature to explicitly use a default parameter when calling a function. I like your first suggestions, but I don't like the second one.

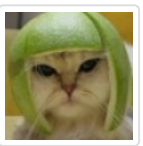
2. Either you declare the parameter with the same default value twice, which is redundant, or you declare them with different default values, which doesn't make sense.

Peter Baum
May 13, 2018 at 3:21 pm · Reply

Hi nascardriver,
Regarding 2... if they had different default values maybe the compiler could point out the conflict. That would take care of the redundancy too, because the programmer could use that feature to verify that the code for each section was consistent.

Not a big deal in any case.

Thanks.



Alex
May 19, 2018 at 8:41 am · Reply

This is a pretty common question. There's some conversation about this topic **here**.

I think the short is that they could have, but it would have made the language more complex, and they didn't feel that was worth the tradeoffs.

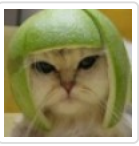
I do wonder if you could work around this by using **std::optional**, which was introduced in C++17.

nascardriver
May 20, 2018 at 7:53 am · Reply

@std::optional can be used, but you'd have to set the default value in the function body which isn't where default values should be located in my opinion.
If there is a better way I'd love to know it.

```
1  #include <iostream>
2  #include <optional>
3  #include <string>
4
5  void printWaldo(std::optional<std::string> optShirtColor, double dbX, double dbY)
6  {
7    std::string strShirtColor{ optShirtColor.value_or("red") };
8
9    std::cout << "Waldo is wearing a " << strShirtColor << " shirt and is located a
10 }
11
12 int main(void)
13 {
14   printWaldo(std::nullopt, 12.3, 45.6);
15   printWaldo("blue", 78.9, 1.2);
16
17   return 0;
18 }
```

Output
Waldo is wearing a red shirt and is located at 12.3, 45.6
Waldo is wearing a blue shirt and is located at 78.9, 1.2



Alex
May 20, 2018 at 1:30 pm · Reply

I agree. I'm not aware of a way to solve this, but I'm also not that familiar with some of the C++17 stuff, so maybe someone who is knows of a good way to resolve.

jayu
February 17, 2018 at 11:56 pm · Reply

char name[10];

//I want to enter default value of name like "name= jayu"how I can do that please send me syntax of this.jayupipaliya198@gmail.com

nascardriver
February 18, 2018 at 2:20 am · Reply

Hi Jayu!

```
const char *szName{ "jayu" }; // Immutable (Can't be modified)
// Or
char arrName[4]{ "jayu" }; // Mutable (Can be modified)
```

Trevor
February 17, 2018 at 1:57 am · Reply

I should probably try it, but I would think that if a function has default arguments and is called from a different CPP file, then the default values would need to be in the function prototype. My thinking is that the default values are inserted by the compiler as it compiles the calling code, otherwise the linker would need to differentiate between calls with all the arguments and calls with missing arguments which require the default parameters.
This would raise the interesting (and scary) prospect of being able to define different default parameter values in different CPP files by using different function prototypes for the same function! However I would not recommend using this - just include the required argument value in the call.
Trevor

Alex
February 19, 2018 at 10:55 am · Reply

I believe you are correct. This is generally avoided by putting your function prototypes in a header and #including that header wherever you need access to the function. That way you only have a single function prototype.

Marcus
January 23, 2018 at 8:11 am · Reply

Is there an explicit way to use the default value?
e.g printValues(-1, *, -1) where '*' means that I would like to use the default value.
Output:
Values: -1 20 -1

nascardriver
January 23, 2018 at 8:59 am · Reply

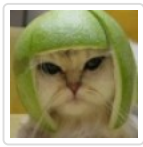
Hi Marcus!
The only way to use default arguments is not setting them at all.

Marcus

January 23, 2018 at 9:23 am · Reply

;(

Thank you nascardriver!



Alex
January 24, 2018 at 8:22 pm · Reply

Short answer: no. Default values can only be invoked for the rightmost parameters in a call, and are invoked by not passing an argument for that parameter in the function call.

Nakamura
January 2, 2018 at 1:59 pm · Reply

How does it work for map? I have a map "std::map<int,std::string> map" , how will it work for optional parameter ?

nascardriver
January 3, 2018 at 12:32 am · Reply

Hi Nakamura!
An std::map can be initialized with empty curly braces

```
#include <iostream>
#include <map>
#include <string>

void fn(std::map<int,std::string> mp = { })
{
    std::cout << mp.size() << std::endl;
}

int main()
{
  fn({{ 4, "Hello" }});
  fn();

  return 0;
}
```

Output

```
1
0
```

nikos-13
July 8, 2017 at 7:00 am · Reply

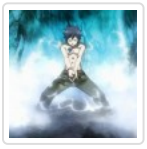
"void printStringInColor(std::string, Color color=COLOR_BLACK); // Color is an enum"
You forgot the parameter's, of type std::string, name.

Alex
July 8, 2017 at 9:18 pm · Reply

Fixed. Even though providing the name of the parameters is optional in a function prototype (only the type is required), it's a good practice to do so.

nikos-13
July 10, 2017 at 6:20 am · Reply

Oh, I didn't know that...



blaze077
May 19, 2017 at 4:18 pm · Reply

"Note that it is impossible to supply a user-defined value for z without also supplying a value for x and y."
This function refers to printValues(int x, int y, int z) so I think the sentence should be:
"Note that it is impossible to supply a user-defined value for x without also supplying a value for y and z."
Thanks as always.

Alex
May 19, 2017 at 7:39 pm · Reply

Lesson fixed. Thanks!

Kess
June 1, 2017 at 6:13 am · Reply

Actually, I believe it was correct in it's former version, as in order to provide the value only for x you could just write printValues(3);
and the function would be called with 3, 20 and 30 as its arguments (20 and 30 being the default values for y and z)

"Note that it is impossible to supply a user-defined value for x without also supplying a value for y and z."
Would mean that I would have to write printValues(3, 20, 30); While possible, it's easier and still correct to write printValues(3);

"Note that it is impossible to supply a user-defined value for z without also supplying a value for x and y."
Would mean that I cannot write printValues(,,3); which is the intended outcome of that particular part of the lesson. Also, the next sentence in the explanation makes it clear that the intention was to point out that you cannot set z while leaving x and y to their default values without at least explicitly set them (to their default value, if you need them to be that way).

Thanks for these top notch lessons ;)
Have a good day

Alex
June 1, 2017 at 10:04 pm · Reply

Agreed -- I've updated and clarified the text to be clearer that I was talking about arguments, not parameters. :)

Akshay Chavan
January 30, 2017 at 7:00 am · Reply

In foo.h

```
void printvalues(int x, int y=10);
```

Shouldn't it be

```
void printValues(int x, int y=10);
```

Alex
January 30, 2017 at 5:54 pm · Reply

Yes, fixed.

Matt
November 19, 2016 at 11:02 am · Reply

At the end of section "Default parameters can only be declared once", you wrote:

"Rule: If the function has a forward declaration, put the default parameters there. Otherwise, put them on the function definition."

Did you mean to write "on", or did you mean "in"?

Alex
November 19, 2016 at 12:29 pm · Reply

Definitely "in". You'd think I didn't proofread this stuff. I guess I just must be horrible at it. :)

Kattencrack Kledge
July 22, 2016 at 11:43 pm · Reply

Little typo on one of the function prototypes:

```
int rollDie(int sides=6);
```

I'm pretty sure you meant rollDice than rollDie XD

Alex
July 30, 2016 at 3:30 pm · Reply

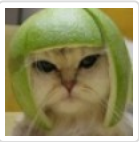
Nope. Die is actually the singular of dice, and in this case, we are just rolling one die, not multiple dice. Though enough people don't know this that I suspect it will be acceptable to use dice in both situations in the near future (if we haven't hit that point already).

Jim
May 22, 2016 at 8:57 pm · Reply

Alex,
This code uses the same name and parameters except one has a capital V and the other uses a lower case v.
Why wouldn't the compiler catch this? Or is this a typo?

[code]
void printvalues(int x, int y=10);

void printValues(int x, int y=10) // error: redefinition of default parameter
{
  std::cout << "x: " << x << '\n';
  std::cout << "y: " << y << '\n';
}



Alex
May 22, 2016 at 10:07 pm · Reply

Typo. It's fixed now, thanks for noticing.

Avneet
August 24, 2015 at 12:41 am · Reply

Alex, is there any way to forward declare a function that has default pararmeters in it...?
[code]
#include <iostream>

void val(int, int);

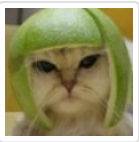
int main()
{
  int value = 10;
  val(value);
  return 0;
}

void val(int nval, int val2 = 20)
{
  std::cout << nval << "\n";
  std::cout << val2 << "\n";
}

This program gives an error at line 8:

"Fire.cpp|8|error: too few arguments to function 'void val(int, int)'|"

and if I remove an int, compiler says that it is unable to find a function with only 1 int parameter.



Alex
August 24, 2015 at 12:15 pm · Reply

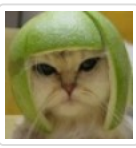
Put the default parameter in the forward declaration:

```
1  void val(int, int = 20);
2
3  int main()
4  {
5      int value = 10;
6      val(value);
7      return 0;
8  }
9
10 void val(int nval, int val2)
11 {
```

```
12      std::cout << nval << "\n";
13      std::cout << val2 << "\n";
14  }
```

kekie
June 10, 2014 at 2:34 pm · Reply

Shouldn't they be called 'Default arguments'?



Alex
November 11, 2015 at 4:15 pm · Reply

They often are.

Sphingine
August 19, 2011 at 10:51 pm · Reply

Can i give d definition as:

void printvalues(int nvalue1=0, int nvalue2=2)
{

cout<<"1st : "<< nvalue1<< endl;
cout<<"2st : "<< nvalue2<< endl;

}

and call the function as:

printvalues(,3);

zingmars
August 20, 2011 at 7:23 am · Reply

No you can't.

dave
October 30, 2009 at 4:50 am · Reply

I must say this site is awesome and clean looking..specially how the code is displayed....looks neat! keep up good work and i am doing last min craming when i googled default params...so wish me luck! :)