# 7.7 — Default parameters

BY ALEX ON AUGUST 6TH, 2007 | LAST MODIFIED BY ALEX ON JULY 8TH, 2017

A **default parameter** (also called an **optional parameter** or a **default argument**) is a function parameter that has a default value provided to it. If the user does not supply a value for this parameter, the default value will be used. If the user does supply a value for the default parameter, the user-supplied value is used instead of the default value.

Consider the following program:

```
void printValues(int x, int y=10)
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}

int main()
{
    printValues(1); // y will use default parameter of 10
    printValues(3, 4); // y will use user-supplied value 4
}
```

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 parameters 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 parameters might be commonly used:

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

**Multiple default parameters**

A function can have multiple default parameters:

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

Given the following function calls:

```
printValues(1, 2, 3);
printValues(1, 2);
printValues(1);
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 parameters must be the rightmost parameters. The following is not allowed:

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

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

**Default parameters can only be declared once**

Once declared, a default parameter can not be redeclared. That means for a function with a forward declaration and a function definition, the default parameter 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 parameter
{
    std::cout << "x: " << x << '\n';
    std::cout << "y: " << y << '\n';
}
```

Best practice is to declare the default parameter 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 parameter for function printValues() because the main.cpp #includes foo.h, which has the forward declaration that defines the default parameter.

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

**Default parameters and function overloading**

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

```
void print(std::string string);
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 default parameters do NOT count towards the parameters that make the function unique. Consequently, the following is not allowed:

```
void printValues(int x);
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 parameters 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**


## Share this:


Facebook Twitter Google Pinterest



C++ TUTORIAL | PRINT THIS POST


## 38 comments to 7.7 — Default parameters

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

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

and

```
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:

```
void printValues(int x);
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

```
printValues(10,)
```

if the second function were desired.

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