11.5 — Default arguments

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.

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.

Multiple default arguments

A function can have multiple parameters with default arguments:

```
#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:

```
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:

```
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

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.

```
#include <iostream>

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

void print(int x, int y=4) // 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 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()` acts as if the user had explicitly called `print(' ')`, which resolves to `print(char)`.

Now consider this case:

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

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. For example:

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

In the last case, the compiler is unable to tell whether `print(1)` should resolve to `print(int)` or one of the two functions where the second parameter has a default value. The result is an ambiguous function call.

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

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 can be resolved at compile time.

In future lesson 20.1 -- Function Pointers, we cover function pointers, which are pointers that can point to functions. When a function is called through a function pointer, it is resolved at runtime. In this case, there is no rewriting of the function call to include default arguments.

Key insight

Because the resolution happens at runtime, default arguments are not resolved when a function is called through a function pointer.

This means that we can use a function pointer to disambiguate a function call that would otherwise be ambiguous due to default arguments. In the following example, we show two ways to do this:

```
#include <iostream>

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

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

int main()
{
//    print(1); // ambiguous function call

    // Deconstructed method
    using vnptr = void(*)(int); // define a type alias for a function pointer to a
void(int) function
    vnptr pi { print }; // initialize our function pointer with function print
    pi(1); // call the print(int) function through the function pointer

    // Concise method
    static_cast<void(*)(int)>(print)(1); // call void(int) version of print with
argument 1

    return 0;
}
```

Related content

See lesson 20.1 -- Function Pointers for more information on function pointers.

Summary

Default arguments provide a useful mechanism to specify values for parameters that the user may or may not want to override. They are frequently used in C++, and you’ll see them a lot in future lessons.