2.3 — Introduction to function parameters and arguments

BY ALEX ON JANUARY 25TH, 2015 | LAST MODIFIED BY NASCARDRIVER ON FEBRUARY 1ST, 2020

In the previous lesson, we learned that we could have a function return a value back to the function's caller. We used that to create a modular *getValueFromUser* function that we used in this program:

```cpp
#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input;
}

int main()
{
    int num { getValueFromUser() };

    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

However, what if we wanted to put the output line into its own function as well? You might try something like this:

```cpp
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // added return statement to return input back to the caller
}

// This function won't compile
// The void return type means the function won't return a value to the caller
void printDouble()
{
    std::cout << num << " doubled is: " << num * 2 << '\n';
}

int main()
{
    int num { getValueFromUser() };

    printDouble();

    return 0;
}
```

This won't compile, because function *printDouble* doesn't know what identifier *num* is. You might try defining num as a variable inside function printDouble():

```cpp
void printDouble()
```

```
{
    int num{}; // we added this line
    std::cout << num << " doubled is: " << num * 2 << '\n';
}
```

While this addresses the compiler error and makes the program compile-able, the program still doesn't work correctly (it always prints "0 doubled is: 0"). The core of the problem here is that function *printDouble* doesn't have a way to access the value the user entered.

We need some way to pass the value of variable *num* to function *printDouble* so that *printDouble* can use that value in the function body.

Function parameters and arguments

In many cases, it is useful to be able to pass information *to* a function being called, so that the function has data to work with. For example, if we wanted to write a function to add two numbers, we need some way to tell the function which two numbers to add when we call it. Otherwise, how would the function know what to add? We do that via function parameters and arguments.

A **function parameter** is a variable used in a function. Function parameters work almost identically to variables defined inside the function, but with one difference: they are always initialized with a value provided by the caller of the function.

Function parameters are defined in the function declaration by placing them in between the parenthesis after the function identifier, with multiple parameters being separated by commas.

Here are some examples of functions with different numbers of parameters:

```
// This function takes no parameters
// It does not rely on the caller for anything
void doPrint()
{
    std::cout << "In doPrint()\n";
}

// This function takes one integer parameter named x
// The caller will supply the value of x
void printValue(int x)
{
    std::cout << x  << '\n';
}

// This function has two integer parameters, one named x, and one named y
// The caller will supply the value of both x and y
int add(int x, int y)
{
    return x + y;
}
```

An **argument** is a value that is passed *from* the caller *to* the function when a function call is made:

```
doPrint(); // this call has no arguments
printValue(6); // 6 is the argument passed to function printValue()
add(2, 3); // 2 and 3 are the arguments passed to function add()
```

Note that multiple arguments are also separated by commas.

How parameters and arguments work together

When a function is called, all of the parameters of the function are created as variables, and the value of each of the arguments is *copied* into the matching parameter. This process is called **pass by value**.

For example:

```cpp
#include <iostream>

// This function has two integer parameters, one named x, and one named y
// The values of x and y are passed in by the caller
void printValues(int x, int y)
{
    std::cout << x << '\n';
    std::cout << y << '\n';
}

int main()
{
    printValues(6, 7); // This function call has two arguments, 6 and 7

    return 0;
}
```

When function *printValues* is called with arguments *6* and *7*, *printValues*'s parameter *x* is created and initialized with the value of *6*, and *printValues*'s parameter *y* is created and initialized with the value of *7*.

This results in the output:

```
6
7
```

Note that the number of arguments must generally match the number of function parameters, or the compiler will throw an error. The argument passed to a function can be any valid expression (as the argument is essentially just an initializer for the parameter, and initializers can be any valid expression).

Fixing our challenge program

We now have the tool we need to fix the program we presented at the top of the lesson:

```cpp
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // added return statement to return input back to the caller
}

void printDouble(int value)
{
    std::cout << value << " doubled is: " << value * 2 << '\n';
}

int main()
{
    int num { getValueFromUser() };

    printDouble(num);

    return 0;
}
```

In this program, variable *num* is first initialized with the value entered by the user. Then, function *printDouble* is called, and the value of argument *num* is copied into the *value* parameter of function *printDouble*. Function *printDouble* then uses the value of parameter *value*.

Using return values as arguments

In the above problem, we can see that variable *num* is only used once, to transport the return value of function *getValueFromUser* to the argument of the call to function *printDouble*.

We can simplify the above example slightly as follows:

```
#include <iostream>

int getValueFromUser() // this function now returns an integer value
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // added return statement to return input back to the caller
}

void printDouble(int value)
{
    std::cout << value << " doubled is: " << value * 2 << '\n';
}

int main()
{
    printDouble(getValueFromUser());

    return 0;
}
```

Now, we're using the return value of function *getValueFromUser* directly as an argument to function *printDouble*!

Although this program is more concise (and makes it clear that the value read by the user will be used for nothing else), you may also find this "compact syntax" a bit hard to read. If you're more comfortable sticking with the version that uses the variable instead, that's fine.

A warning about function argument order of evaluation

The C++ specification does not define whether arguments are matched with parameters in left to right order or right to left order. When copying values, order is of no consequence. However, if the arguments are function calls, then this can be problematic:

```
someFunction(a(), b()); // a() or b() may be called first
```

If the architecture evaluates left to right, a() will be called before b(). If the architecture evaluates right to left, b() will be called before a(). This may or may not be of consequence, depending on what a() and b() do.

If it is important that one argument evaluate first, you should explicitly define the order of execution, like so:

```
int avar{ a() }; // a() will always be called first
int bvar{ b() }; // b() will always be called second

someFunction(avar, bvar); // it doesn't matter whether avar or bvar are copied first because th
```

Warning

The C++ specification does not define whether function calls evaluate arguments left to right or right to left. Take care not to make function calls where argument order matters.

How parameters and return values work together

By using both parameters and a return value, we can create functions that take data as input, do some calculation with it, and return the value to the caller.

Here is an example of a very simple function that adds two numbers together and returns the result to the caller:

```
#include <iostream>

// add() takes two integers as parameters, and returns the result of their sum
// The values of x and y are determined by the function that calls add()
int add(int x, int y)
{
    return x + y;
}

// main takes no parameters
int main()
{
    std::cout << add(4, 5) << '\n'; // Arguments 4 and 5 are passed to function add()
    return 0;
}
```

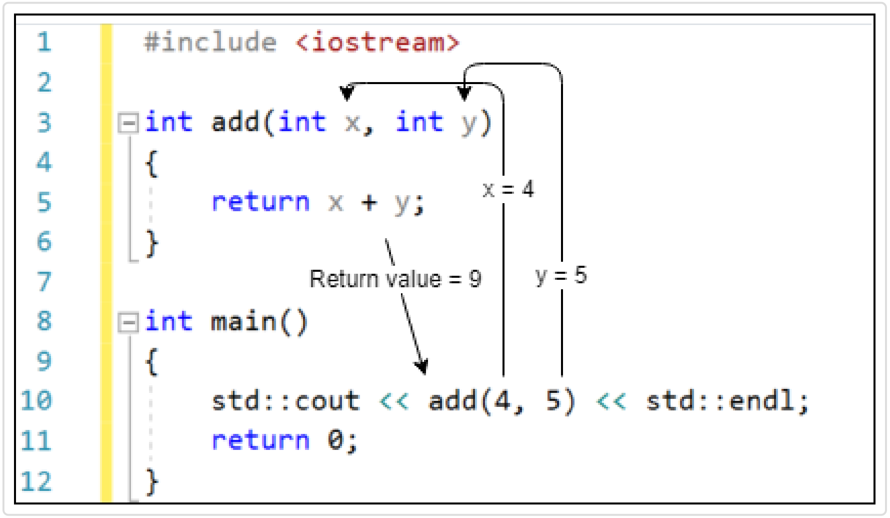
Execution starts at the top of *main*. When `add(4, 5)` is evaluated, function *add* is called, with parameter *x* being initialized with value *4*, and parameter *y* being initialized with value *5*.

The *return statement* in function *add* evaluates *x* + *y* to produce the value *9*, which is then returned back to *main*. This value of *9* is then sent to *std::cout* to be printed on the console.

Output:

```
9
```

In pictorial format:



```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int main()
{
    std::cout << add(4, 5) << std::endl;
    return 0;
}
```

More examples

Let’s take a look at some more function calls:

```
#include <iostream>

int add(int x, int y)
{
    return x + y;
}

int multiply(int z, int w)
{
    return z * w;
}

int main()
{
    std::cout << add(4, 5) << '\n'; // within add() x=4, y=5, so x+y=9
    std::cout << add(1 + 2, 3 * 4) << '\n'; // within add() x=3, y=12, so x+y=15

    int a{ 5 };
    std::cout << add(a, a) << '\n'; // evaluates (5 + 5)

    std::cout << add(1, multiply(2, 3)) << '\n'; // evaluates 1 + (2 * 3)
    std::cout << add(1, add(2, 3)) << '\n'; // evaluates 1 + (2 + 3)

    return 0;
}
```

This program produces the output:

```
9
15
10
7
6
```

The first statement is straightforward.

In the second statement, the arguments are expressions that get evaluated before being passed. In this case, *1 + 2* evaluates to *3*, so *3* is copied to parameter *x*. *3 * 4* evaluates to *12*, so *12* is copied to parameter *y*. *add(3, 12)* resolves to *15*.

The next pair of statements is relatively easy as well:

```
    int a{ 5 };
    std::cout << add(a, a) << '\n'; // evaluates (5 + 5)
```

In this case, *add()* is called where the value of *a* is copied into both parameters *x* and *y*. Since *a* has value *5*, *add(a, a)* = *add(5, 5)*, which resolves to value *10*.

Let's take a look at the first tricky statement in the bunch:

```
    std::cout << add(1, multiply(2, 3)) << '\n'; // evaluates 1 + (2 * 3)
```

When the function *add* is executed, the program needs to determine what the values for parameters *x* and *y* are. *x* is simple since we just passed it the integer *1*. To get a value for parameter *y*, it needs to evaluate *multiply(2, 3)* first. The program calls *multiply* and initializes *z = 2* and *w = 3*, so *multiply(2, 3)* returns the integer value *6*. That return value of *6* can now be used to initialize the *y* parameter of the *add* function. *add(1, 6)* returns the integer *7*, which is then passed to std::cout for printing.

Put less verbosely:
add(1, multiply(2, 3)) evaluates to *add(1, 6)* evaluates to *7*

The following statement looks tricky because one of the parameters given to *add* is another call to *add*.

```
    std::cout << add(1, add(2, 3)) << '\n'; // evaluates 1 + (2 + 3)
```

But this case works exactly the same as the prior case. add(2, 3) resolves first, resulting in the return value of *5*. Now it can resolve add(1, 5), which evaluates to the value *6*, which is passed to std::cout for printing.

Less verbosely:
add(1, add(2, 3)) evaluates to *add(1, 5)* => evaluates to *6*

Conclusion

Function parameters and return values are the key mechanisms by which functions can be written in a reusable way, as it allows us to write functions that can perform tasks and return retrieved or calculated results back to the caller without knowing what the specific inputs or outputs are ahead of time.

Quiz time

Question #1

What's wrong with this program fragment?

```
void multiply(int x, int y)
{
    return x * y;
}

int main()
{
    std::cout << multiply(4, 5) << '\n';
    return 0;
}
```

Show Solution

Question #2

What two things are wrong with this program fragment?

```
int multiply(int x, int y)
{
    int product{ x * y };
}

int main()
{
    std::cout << multiply(4) << '\n';
    return 0;
}
```

Show Solution

Question #3

What value does the following program print?

```
#include <iostream>

int add(int x, int y, int z)
{
    return x + y + z;
}

int multiply(int x, int y)
{
    return x * y;
}

int main()
{
    std::cout << multiply(add(1, 2, 3), 4) << '\n';
    return 0;
}
```

Show Solution

Question #4

Write a function called doubleNumber() that takes one integer parameter and returns twice the value passed in.

Show Solution

Question #5

5) Write a complete program that reads an integer from the user, doubles it using the doubleNumber() function you wrote in the previous quiz, and then prints the doubled value out to the console.

Show Solution

2.4 -- Introduction to local scope

Index

2.2 -- Function return values

C++ TUTORIAL | PRINT THIS POST

492 comments to 2.3 — Introduction to function parameters and arguments

« Older Comments 1 ... 6 7 8

Vitaliy Sh.
January 31, 2020 at 9:12 pm · Reply

Hi Sires!

```
// There is two diagrams: one above, and one below of this text ("However,...). There is no
"However, what if we wanted to put the output line into its own function as well? You might


// <em>in</em>, to match style of "An argument is..." definition.
"A function parameter is a variable used in a function."


// std::cout << "In doPrint()\n";
// std::cout << x << '\n';
// To show the need of separate '\n' in some cases, yet demonstrate embed-ment.
"Here's some examples of functions with different numbers of parameters:"


// x and y are initialized with value of 6 and 7 respectively.
"parameter x is created and assigned the value of 6, and ... y is..."

```

```

//      architecture
"If the architectures evaluates left to right, a() will be called before b()."


//        or may not be of
//    may be or not be of
"This may or not be of consequence,"


// Please, add a whitespace line between @bvar definition and someFunction() call.
"If it is important that one argument evaluate first, you should explicitly define the orde


// : instead of .   ?
"Here is an example of a very simple function that adds two numbers together and returns th


// add() instead of add,
// : instead of .   ?
"The following statement looks tricky because one of the parameters given to add is another


// Q #1:
//                        snippet?
"What's wrong with this program fragment?"


// Q #5:
//         to solve a quiz #4,
"you wrote for quiz answer 4,"




// It's true that people who do things like this
// are dissapear sometimes without a trace?
printInteger(
                doubleInteger(
                        getUserInteger()
                )
);
```

Also, can your please change 4 "before std:cin" { 0 }s to { }; here?

nascardriver

February 1, 2020 at 3:19 am · Reply

Thanks for your suggestions! I integrated most of them into the lesson.

Tony98

January 24, 2020 at 1:46 pm · Reply

How about using multiple functions? Is this correct or not? Thank you for your work!

```
void printValue(int printResult)
{
    std::cout << "The result is: " << printResult;
}

```

```
int doubleNumber(int valueDouble)
{
    return 2 * valueDouble;
}

int readInteger()
{
    int value{};
    std::cin >> value;
    return value;
}

void enterInput()
{
    std::cout << "Please enter an integer: ";
}

int main()
{
    enterInput();
    int result{ doubleNumber(readInteger()) };
    printValue(result);

    return 0;
}
```

nascardriver
January 25, 2020 at 3:16 am · Reply

Yes it's correct, good job!

prince
January 18, 2020 at 3:13 am · Reply

In our fixing our challenge program
why does the int value parameter gets it's value from the getValueFromUser funtion?
and why does the int value named like that? both int and value is a reserve keyword right?

and can you please explain the use of the int value parameter that you used?

nascardriver
January 18, 2020 at 3:23 am · Reply

The return value of `getValueFromUser` is stored in `num` (Line 19). `num` is then passed to `printDouble` (Line 21).
"value" is not a reserved identifier, the syntax highlighting on learncpp makes it seem like it is, but it isn't.
We use a parameter because we want to transport a value from one function to another.

prince
January 18, 2020 at 4:17 am · Reply

one last question. in the quiz number 3
what will be the value of int z?

nascardriver

January 18, 2020 at 4:20 am · Reply

3, it gets it from `main`:

```
add(1, 2, 3)
//  x  y  z
```

Kermit

January 9, 2020 at 12:37 am · Reply

```
#include <iostream>

using namespace std;
/*
int getInputFromUser()                          // This asks for user input
{
    cout << "Enter an integer: ";
    int input{};
    cin >> input;
    return input;
}

// Parameters are like variables but one difference they are always initialized with a valu
void printDouble(int value)                      // This function has one integer parameter
{

    cout << value << " Doubled is: " << value * 2 << '\n';
}


int main()
{
//    int num{ getInputFromUser() };                // setting up the return value in variab
    printDouble(getInputFromUser());             // Calling the function printDouble() with
    return 0;
}
*/

int add(int x, int y)
{
    return x + y;
}

int multiply(int z, int w)
{
    return z * w;
}

int userInput()
{
    cout << " Enter an integer: ";
    int input{};
    cin >> input;
    return input;
}

int doubleNumber(int dob)
```

```
{
    return dob * 2;
}

int main()
{
    int ad{ userInput() };
    int ad1{ userInput() };
    cout << " " << ad << " + " << ad1 << " = " << add(ad, ad1) << '\n';
    cout << " 4 + 5 = " << add(4, 5) << '\n';
    cout << " 1 + 2, 3 * 4 = " << add(1 + 2, 3 * 4) << '\n';

    int a{ 5 };
    cout << " 5 + 5 = " << add(a, a) << '\n';
    cout << " " << add(1, multiply(2, 3)) << '\n';
    cout << " " << add(1, add(2, 3)) << '\n';
    cout << " Doubled that no is: " << doubleNumber(userInput()) << '\n';
    return 0;
}
```

so far

nascardriver
January 9, 2020 at 4:17 am · Reply

So far so good. You can re-use variable names if they are in different functions. `multiply` doesn't see what's inside `add`.

abdul khaleqh
December 30, 2019 at 2:01 am · Reply

```
//the challenge modified
#include<iostream>
using namespace std;
int input(int x)//using parameters instead of local variables
{
    cout<<"enter a input:"<<endl;
    cin>>x;
    return x;
}
void doubled(int x)
{
    cout<<"the input is:"<<'\n'<<x<<'\n'<<"the doubled is :"<<'\n'<<x*2;

}
int main()
{
    int x=input(x);//passing values to function parameters using variable x
    doubled(x);
    return 0;
}
```

nascardriver
December 30, 2019 at 5:48 am · Reply

- Don't use "using namespace".
- Don't use `std::endl` unless you need to.
- Initialize variables with brace initialization.
- If your program prints anything, the last thing it prints should be a line feed ('\n').

Centriax
December 23, 2019 at 7:15 am · Reply

When should you be using "int" and when should you be using "void" and how do they differentiate from each other?

nascardriver
December 23, 2019 at 7:39 am · Reply

This is covered in lesson 2.2.

« Older Comments 1 ... 6 7 8