2.2 — Function return values (value-returning functions)

ALEX JANUARY 2, 2024

Consider the following program:

```
#include <iostream>

int main()
{
    // get a value from the user
    std::cout << "Enter an integer: ";
    int num{};
    std::cin >> num;

    // print the value doubled
    std::cout << num << " doubled is: " << num * 2 << '\n';

    return 0;
}
```

This program is composed of two conceptual parts: First, we get a value from the user. Then we tell the user what double that value is.

Although this program is trivial enough that we don't need to break it into multiple functions, what if we wanted to? Getting an integer value from the user is a well-defined job that we want our program to do, so it would make a good candidate for a function.

So let's write a program to do this:

```
// This program doesn't work
#include <iostream>

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

int main()
{
    getValueFromUser(); // Ask user for input

    int num{}; // How do we get the value from getValueFromUser() and use it to initialize this variable?

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

    return 0;
}
```

While this program is a good attempt at a solution, it doesn't quite work.

When function `getValueFromUser` is called, the user is asked to enter an integer as expected. But the value they enter is lost when `getValueFromUser` terminates and control returns to `main`. Variable `num` never gets initialized with the value the user entered, and so the program always prints the answer `0`.

What we're missing is some way for `getValueFromUser` to return the value the user entered back to `main` so that `main` can make use of that data.

Return values

When you write a user-defined function, you get to determine whether your function will return a value back to the caller or not. To return a value back to the caller, two things are needed.

First, your function has to indicate what type of value will be returned. This is done by setting the function's **return type**, which is the type that is defined before the function's name. In the example above, function `getValueFromUser` has a return type of `void` (meaning no value will be returned to the caller), and function `main` has a return type of `int` (meaning a value of type `int` will be returned to the caller). Note that this doesn't determine what specific value is returned -- it only determines what type of value will be returned.

Related content

We explore functions that return `void` further in the next lesson (2.3 -- Void functions (non-value returning functions) (https://www.learncpp.com/cpp-tutorial/void-functions-non-value-returning-functions/)).

Second, inside the function that will return a value, we use a **return statement** to indicate the specific value being returned to the caller. The specific value returned from a function is called the **return value**. When the return statement is executed, the function exits immediately, and the return value is copied from the function back to the caller. This process is called **return by value**.

Let's take a look at a simple function that returns an integer value, and a sample program that calls it:

```
#include <iostream>

// int is the return type
// A return type of int means the function will return some integer value to the caller (the specific value is not specified
here)
int returnFive()
{
    // the return statement indicates the specific value that will be returned
    return 5; // return the specific value 5 back to the caller
}

int main()
{
    std::cout << returnFive() << '\n'; // prints 5
    std::cout << returnFive() + 2 << '\n'; // prints 7

    returnFive(); // okay: the value 5 is returned, but is ignored since main() doesn't do anything with it

    return 0;
}
```

When run, this program prints:

```
5
7
```

Execution starts at the top of `main`. In the first statement, the function call to `returnFive` is evaluated, which results in function `returnFive` being called. Function `returnFive` returns the specific value of `5` back to the caller, which is then printed to the console via `std::cout`.

In the second function call, the function call to `returnFive` is evaluated, which results in function `returnFive` being called again. Function `returnFive` returns the value of `5` back to the caller. The expression `5 + 2` is evaluated to produce the result `7`, which is then printed to the console via `std::cout`.

In the third statement, function `returnFive` is called again, resulting in the value `5` being returned back to the caller. However, function `main` does nothing with the return value, so nothing further happens (the return value is ignored).

Note: Return values will not be printed unless the caller sends them to the console via `std::cout`. In the last case above, the return value is not sent to `std::cout`, so nothing is printed.

Tip

When a called function returns a value, the caller may decide to use that value in an expression or statement (e.g. by using it to initialize a variable, or sending it to `std::cout`) or ignore it (by doing nothing else). If the caller ignores the return value, it is discarded (nothing is done with it).

Fixing our challenge program

With this in mind, we can fix the program we presented at the top of the lesson:

```
#include <iostream>

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

    return input; // return the value the user entered back to the caller
}

int main()
{
    int num { getValueFromUser() }; // initialize num with the return value of getValueFromUser()

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

    return 0;
}
```

When this program executes, the first statement in `main` will create an `int` variable named `num`. When the program goes to initialize `num`, it will see that there is a function call to `getValueFromUser()`, so it will go execute that function. Function `getValueFromUser`, asks the user to enter a value, and then it returns that value back to the caller (`main`). This return value is used as the initialization value for variable `num`.

Compile this program yourself and run it a few times to prove to yourself that it works.

Revisiting main()

You now have the conceptual tools to understand how the `main` function actually works. When the program is executed, the operating system

makes a function call to `main`. Execution then jumps to the top of `main`. The statements in `main` are executed sequentially. Finally, `main` returns an integer value (usually `0`), and your program terminates. The return value from `main` is sometimes called a **status code** (also sometimes called an **exit code**, or rarely a **return code**), as it is used to indicate whether the program ran successfully or not.

By definition, a status code of `0` means the program executed successfully.

Best practice

Your `main` function should return the value `0` if the program ran normally.

A non-zero status code is often used to indicate failure (and while this works fine on most operating systems, strictly speaking, it's not guaranteed to be portable).

For advanced readers

The C++ standard only defines the meaning of 3 status codes: 0, EXIT_SUCCESS, and EXIT_FAILURE. 0 and EXIT_SUCCESS both mean the program executed successfully. EXIT_FAILURE means the program did not execute successfully.

EXIT_SUCCESS and EXIT_FAILURE are preprocessor macros defined in the <cstdlib> header:

```
#include <cstdlib> // for EXIT_SUCCESS and EXIT_FAILURE

int main()
{
    return EXIT_SUCCESS;
}
```

If you want to maximize portability, you should only use 0 or EXIT_SUCCESS to indicate a successful termination, or EXIT_FAILURE to indicate an unsuccessful termination.

We cover the preprocessor and preprocessor macros in lesson 2.10 -- Introduction to the preprocessor (https://www.learncpp.com/cpp-tutorial/introduction-to-the-preprocessor/).

C++ disallows calling the `main()` function explicitly.

As an aside...

C does allow `main()` to be called explicitly, so some C++ compilers will allow this for compatibility reasons.

For now, you should also define your `main()` function at the bottom of your code file, below other functions, and avoid calling it explicitly.

A value-returning function that does not return a value will produce undefined behavior

A function that returns a value is called a **value-returning function**. A function is value-returning if the return type is anything other than `void`.

A value-returning function must return a value of that type (using a return statement), otherwise undefined behavior will result.

Related content

We discuss undefined behavior in lesson 1.6 -- Uninitialized variables and undefined behavior (https://www.learncpp.com/cpp-tutorial/uninitialized-variables-and-undefined-behavior/).

Here's an example of a function that produces undefined behavior:

```
#include <iostream>

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

    // note: no return statement
}

int main()
{
    int num { getValueFromUserUB() }; // initialize num with the return value of getValueFromUserUB()

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

    return 0;
}
```

A modern compiler should generate a warning because `getValueFromUserUB` is defined as returning an `int` but no return statement is provided. Running such a program would produce undefined behavior, because `getValueFromUserUB()` is a value-returning function that does not return a value.

In most cases, compilers will detect if you've forgotten to return a value. However, in some complicated cases, the compiler may not be able to properly determine whether your function returns a value or not in all cases, so you should not rely on this.

Best practice

Make sure your functions with non-void return types return a value in all cases.

Failure to return a value from a value-returning function will cause undefined behavior.

Function main will implicitly return 0 if no return statement is provided

The only exception to the rule that a value-returning function must return a value via a return statement is for function `main()`. The function `main()` will implicitly return the value `0` if no return statement is provided. That said, it is best practice to explicitly return a value from `main`, both to show your intent, and for consistency with other functions (which will exhibit undefined behavior if a return value is not specified).

Functions can only return a single value

A value-returning function can only return a single value back to the caller each time it is called.

Note that the value provided in a return statement doesn't need to be literal -- it can be the result of any valid expression, including a variable or even a call to another function that returns a value. In the `getValueFromUser()` example above, we returned a variable `input`, which held the number the user input.

There are various ways to work around the limitation of functions only being able to return a single value, which we'll cover in future lessons.

The function author can decide what the return value means

The meaning of the value returned by a function is determined by the function's author. Some functions use return values as status codes, to indicate whether they succeeded or failed. Other functions return a calculated or selected value. Other functions return nothing (we'll see examples of these in the next lesson).

Because of the wide variety of possibilities here, it's a good idea to document your function with a comment indicating what the return values mean. For example:

```
// Function asks user to enter a value
// Return value is the integer entered by the user from the keyboard
int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value the user entered back to the caller
}
```

Reusing functions

Now we can illustrate a good case for function reuse. Consider the following program:

```
#include <iostream>

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

    int y{};
    std::cout << "Enter an integer: ";
    std::cin >> y;

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}
```

While this program works, it's a little redundant. In fact, this program violates one of the central tenets of good programming: **Don't Repeat Yourself** (often abbreviated **DRY**).

Why is repeated code bad? If we wanted to change the text "Enter an integer:" to something else, we'd have to update it in two locations. And what if we wanted to initialize 10 variables instead of 2? That would be a lot of redundant code (making our programs longer and harder to understand), and a lot of room for typos to creep in.

Let's update this program to use our `getValueFromUser` function that we developed above:

```
#include <iostream>

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

    return input;
}

int main()
{
    int x{ getValueFromUser() }; // first call to getValueFromUser
    int y{ getValueFromUser() }; // second call to getValueFromUser

    std::cout << x << " + " << y << " = " << x + y << '\n';

    return 0;
}
```

This program produces the following output:

```
Enter an integer: 5
Enter an integer: 7
5 + 7 = 12
```

In this program, we call `getValueFromUser` twice, once to initialize variable `x`, and once to initialize variable `y`. That saves us from duplicating the code to get user input, and reduces the odds of making a mistake. Once we know `getValueFromUser` works, we can call it as many times as we desire.

This is the essence of modular programming: the ability to write a function, test it, ensure that it works, and then know that we can reuse it as many times as we want and it will continue to work (so long as we don't modify the function -- at which point we'll have to retest it).

Best practice

Follow the DRY best practice: "don't repeat yourself". If you need to do something more than once, consider how to modify your code to remove as much redundancy as possible. Variables can be used to store the results of calculations that need to be used more than once (so we don't have to repeat the calculation). Functions can be used to define a sequence of statements we want to execute more than once. And loops (which we'll cover in a later chapter) can be used to execute a statement more than once.

As an aside...

The opposite of DRY is WET ("Write everything twice").

Conclusion

Return values provide a way for functions to return a single value back to the function's caller.

Functions provide a way to minimize redundancy in our programs.

Quiz time

Question #1

Inspect (do not compile) each of the following programs. Determine what the program will output, or whether the program will generate a compiler error.

Assume you have "treat warnings as errors" turned off.

1a)

```
#include <iostream>

int return7()
{
    return 7;
}

int return9()
{
    return 9;
}

int main()
{
    std::cout << return7() + return9() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

1b)

```
#include <iostream>

int return7()
{
    return 7;

    int return9()
    {
        return 9;
    }
}

int main()
{
    std::cout << return7() + return9() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

1c)

```
#include <iostream>

int return7()
{
    return 7;
}

int return9()
{
    return 9;
}

int main()
{
    return7();
    return9();

    return 0;
}
```

Show Solution (javascript:void(0))

1d)

```
#include <iostream>

int getNumbers()
{
    return 5;
    return 7;
}

int main()
{
    std::cout << getNumbers() << '\n';
    std::cout << getNumbers() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

1e)

```
#include <iostream>

int return 5()
{
    return 5;
}

int main()
{
    std::cout << return 5() << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

1f) Extra credit: Will the following program compile?

```
#include <iostream>

int returnFive()
{
    return 5;
}

int main()
{
    std::cout << returnFive << '\n';

    return 0;
}
```

Show Solution (javascript:void(0))

Question #2

What does “DRY” stand for, and why is it a useful practice to follow?

Show Solution (javascript:void(0))

Next lesson
2.3 Void functions (non-value returning functions)

Back to table of contents

Previous lesson
2.1 Introduction to functions

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

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

376 COMMENTS

Newest