More on Functions and Optional Arguments

Thomas Merkh

tmerkh55@gmail.com

Section 2, Week 6

April 30 2020
Overview

1. Functions and Parameters

2. Optional Arguments

3. Next Time
Last Time

Last time we discussed functions and modular programming. We will talk more about functions today. As a refresher:

- Functions are reusable and callable blocks of code
- Functions accept parameters or inputs, and may return a value
- Making small, independently testable chunks of code is useful in that they are reusable, easy to debug, and easy to understand
Last time we discussed functions and modular programming. We will talk more about functions today. As a refresher:

- Functions are reusable and callable blocks of code
- Functions accept parameters or inputs, and may return a value
- Making small, independently testable chunks of code is useful in that they are reusable, easy to debug, and easy to understand

Today, we will learn about optional parameters for functions, a little on how functions handle *bad inputs*, and continue to refine the idea that functions are abstract objects in C++. 
1 Functions and Parameters

2 Optional Arguments

3 Next Time
Function Types

Before getting started, I wanted to mention again that functions have *types*. One type that a function can take that a variable can not is a so-called *void*. A function that is declared as a `void` is simply a function which returns nothing.

- This does not mean that the function is not doing anything!
- Often, functions can change values without returning anything (more on this later).
- Similarly, functions can be used strictly for printing to the console or for debugging.
In the above code, we refer to the inputs of `divider` as *parameters*. These inputs are absolutely mandatory, in the sense that calling the function `divider(a);` or `divider();` will cause errors that terminate the program.

- Note that the type for each parameter must be specified in advance.
What do you think will happen if we try:

```cpp
double divider(double a, double b){
    return a/b;
}

int main() {
    double x = 5.5;
    cout << divider(x,0) << endl;
    return 0;
}
```

If you said, the program will throw an error and terminate, you’d be correct. Division by zero is not allowed, even in coding. This doesn’t seem like a problem, the error is right in front of us. In real programs, that won’t be the case, functions will throw errors and we won’t know which one did, where it was, or why. This motivates the need for a way to track where errors occur, especially in relation to providing functions bad parameters.
What do you think will happen if we try:

```cpp
double divider(double a, double b){
    return a/b;
}

int main() {
    double x = 5.5;
    cout << divider(x,0) << endl;
    return 0;
}
```

If you said, the program will throw an error and terminate, you’d be correct. Division by zero is not allowed, even in coding. This doesn’t seem like a problem, the error is right in front of us.

- In real programs, that won’t be the case, functions will throw errors and we won’t know which one did, where it was, or why
- This motivates the need for a way to track where errors occur, especially in relation to providing functions bad parameters
There are several potential fixes for this, such as carefully checking parameters *before* passing them to functions. Alternatively, we can expect that the passed in values will be valid, and check to make sure *after*. This can be done by:

```cpp
double divider(double a, double b) {
    assert(b != 0.0);
    return a/b;
}

int main() {
    double x = 5.5;
    double y1 = 9.8;
    cout << divider(x, y1) << endl;
    cout << divider(x, 0) << endl;
    return 0;
}
```

What do you think will happen here?
If you said that an error message would be thrown, indicating where the error occurred and why, you’d be correct. If these assert functions exist (note: don’t forget to #include <cassert>), then should we be using them before or after passing parameters to functions?
If you said that an error message would be thrown, indicating where the error occurred and why, you’d be correct. If these assert functions exist (note: don’t forget to include `<cassert>`), then should we be using them before or after passing parameters to functions?

My vote is *after*, meaning put the assert statement inside the function itself. Why? Because its less typing. We only need to write the assert statement once, rather than every single time we call the function.
1 Functions and Parameters

2 Optional Arguments

3 Next Time
Optional Arguments

Although it seems less than motivated at the moment, learning about *optional arguments* is important for understanding how more powerful objects (errhm.. classes?) work and are defined. Consider

```cpp
#include <cmath>

double hammer(double nail, double power = 10.0)
{
    return pow(nail, power);
}

int main()
{
    double x = 5.5;
    double y = hammer(x);
    double z = hammer(x, 5);
    return 0;
}
```

The new thing here is, we can call `hammer` with either one or two arguments and it works either way.
Default Parameters

The function \texttt{hammer} has an \textit{optional} parameter, \texttt{power}.

- If nothing is supplied, then \texttt{power} will default to the value 10
- Otherwise, we can supply any \texttt{double} we wish
The function `hammer` has an *optional* parameter, `power`.

- If nothing is supplied, then `power` will default to the value 10
- Otherwise, we can supply any `double` we wish

Now, life is full of rules. Optional parameters are no exception. Consider

```cpp
#include <cmath>

double hammerA(double nail, double power = 10.0) {
    return pow(nail, power);
}
double hammerB(double power = 10.0, double nail) {
    return pow(nail, power);
}

int main() {
    double x = 5.5;
    double y = hammerA(x);
    double z = hammerA(x, 5);
    double w = hammerB(x); // Why does this break?
    return 0;
}
```
This breaks because the function `hammerB` doesn’t know if `x` is intended as the optional parameter, or as the mandatory parameter. In short, we require that optional parameters come after mandatory parameters.

- If you zoned out, mandatory parameters *do not* have default values
- *Optional parameters* come last in the list, and are supplied with default values if none are provided
There is one other odd parameter-related thing you should know. Consider the code:

```cpp
#include <iostream>
using namespace std;
int function(int a, int b, int c)
{
    return a*b*c;
}

int function(int a, int b)
{
    return a*b;
}

int main()
{
    cout << function(2,3,4) << " \n" << function(2,3);
    return 0;
}
```
Example: two functions, two purposes

The function function here is declared and defined twice, but each time with a different set of parameters. Although this particular function could be written by simply having the optional parameter int c = 1, it goes to show that there are several way to have optional parameters.

- This is the default method of defining a so-called initializer for a class (discussed later)
- Neither way is necessarily better, but sometimes one method of having optional variables is more natural than the other
Why would we ever want to pass one function into another function as an input??
Why would we ever want to pass one function into another function as an input?? Try to guess what the following code does:

```cpp
auto benchmark = [](auto f)
{
    start_timer();
    f();
    cout << end_timer() << endl;
};

int main()
{
    Data_Type some_data;
    benchmark([&some_data]{some_computation();});
    return 0;
}
```

Although there are several things yet to be discussed going on here, this code essentially defines a benchmark function which times how long it takes to do a particular computation using whatever function you provide it.
Why would we ever want to pass one function into another function as an input?? Try to guess what the following code does:

```cpp
auto benchmark = [](auto f)
{
    start_timer();
    f();
    cout << end_timer() << endl;
};

int main()
{
    Data_Type some_data;
    benchmark([&some_data]{some_computation();});
    return 0;
}
```

Although there are several things yet to be discussed going on here, this code essentially defines a benchmark function which times how long it takes to do a particular computation using whatever function you provide it.
For this reason, functions can be thought of as themselves an abstract object in C++, something which can be passed around.

```cpp
int operation(int x, int y, std::function<int(int, int)> func)
{
    return func(x, y);
}
```
For this reason, functions can be thought of as themselves an abstract object in C++, something which can be passed around.

For completeness, I included a code snippet below showing how this works and the syntax required.

```cpp
int operation (int x, int y, std::function<int(int, int)> func) {
    return func(x, y);
}
```
For this reason, functions can be thought of as themselves an abstract object in C++, something which can be passed around.

For completeness, I included a code snippet below showing how this works and the syntax required.

```cpp
int operation (int x, int y, std::function<int (int, int)> func)
{
    return func(x,y);
}
```

Questions?
1 Functions and Parameters

2 Optional Arguments

3 Next Time
Parameter passing: reference vs value, Const specification

Have a nice weekend!