Functions, Function Prototypes, and Modular Programming

Thomas Merkh
tmerkh@math.ucla.edu

Section 2, Week 5

May 5, 2020
Overview

1. Functions
2. Function Prototypes
3. Modular Programming
4. Next Time
Last time we discussed the do-while loop and random numbers. As a refresher:

- Do-while loops always execute at least once, since they check the test condition *after* each iterate.
- Random number generation can be done by optionally “seeding” the generator and then calling `rand`.
Last time we discussed the do-while loop and random numbers. As a refresher:

- Do-while loops always execute at least once, since they check the test condition after each iterate
- Random number generation can be done by optionally “seeding” the generator and then calling rand

Today, we will cover possibly the most important coding principles and construct, period. That is, modular programming and functions.
1 Functions

2 Function Prototypes

3 Modular Programming

4 Next Time
Functions are ubiquitous throughout C++ and most coding languages. A function could be defined as a block of code which only executes when it is called. A few things about functions:

- Functions can accept parameters, input, or data
- Functions typically perform a certain action
- Functions can optionally return a value or object
- Functions are useful in that they may be reused and developed in isolation
Consider the following code:

```cpp
#include <iostream>
int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a, b;
    cout << "Enter two integers:" << endl;
    cin >> a >> b;
    cout << "Summation = " << add(a, b);
    return 0;
}
```

Here, there is a function `add` defined, where the code passes in two integers `a` and `b`, and the `add` function returns their sum. Notice: `add` has a type `int`. 
# Functions: An Example

Consider the following code:

```cpp
#include <iostream>

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

int main()
{
    int a, b;
    cout << "Enter two integers: \n" << endl;
    cin >> a >> b;
    cout << "Summation = " << add(a, b);
    return 0;
}
```

Here, there is a function add defined, where the code passes in two integers \(a\) and \(b\), and the add function returns their sum. Notice: add has a type int.
How do Functions Work?

The steps to making functions work in your program are:

- Properly declaring a function
- Defining a function and possibly what it will return
- Calling said function
How do Functions Work?

The steps to making functions work in your program are:

- Properly declaring a function
- Defining a function and possibly what it will return
- Calling said function

Declaring and defining a function is different, just as initializing a variable and declaring a variable are different.
How do Functions Work?

The steps to making functions work in your program are:

- Properly declaring a function
- Defining a function and possibly what it will return
- Calling said function

Declaring and defining a function is different, just as initializing a variable and declaring a variable are different.
The difference between declaring a function and defining what a function does can be demonstrated by the following:

```cpp
void myFunction(); // Function declaration

int main()
{
    myFunction(); // call the function
    return 0;
}

// Function definition
void myFunction()
{
    cout << "I just got executed!";
}
```
The difference between declaring a function and defining what a function does can be demonstrated by the following:

```c
void myFunction(); // Function declaration

int main() {
    myFunction(); // call the function
    return 0;
}

// Function definition
void myFunction() {
    cout << "I just got executed!";
}
```

We can see that the first line declares that there is a function `myFunction` which exists, and later in the program "what the function does" is provided.
Functions are building blocks of any useful C++ program, just like variables, classes, and so on. Just like these, functions also have a type, which is basically describing what type of object the function returns.

Consider the differences between:

```cpp
int sumInt(double a, double b) { return (int) (a+b); }
long sumLong(double a, double b) { return (long) (a+b); }
double sumDub(double a, double b) { return a+b; }
```

What do you think `sumInt(4.6, 0.5);` returns?
Functions are building blocks of any useful C++ program, just like variables, classes, and so on. Just like these, functions also have a type, which is basically describing what type of object the function returns. Consider the differences between:

```cpp
int sumInt(double a, double b){return (int) (a+b);}
long sumLong(double a, double b){return (long) (a+b);}
double sumDub(double a, double b){return a+b;}
```

What do you think `sumInt(4.6,0.5);` returns?

**Casting review:** What if `sumInt` was defined as

```cpp
int sumInt(double a, double b){return (int) a+b;}
```
1 Functions

2 Function Prototypes

3 Modular Programming

4 Next Time
Definition

A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of arguments.

- Looks just like a function definition except that it has no body, just as we saw in the previous
- The compiler will get upset if it comes across a function begin called before it has been declared
- The compiler won’t get upset if you declare it, then choose to define the function later in the code

Why would anyone actually do this though?
Usefulness of Function Prototyping

Function prototypes have advantages and drawbacks:

- In larger programs, sometimes it’s nice to have the `main` code block at the top of the actual program file. This can’t be done without prototyping, since all functions would have to come first.

- Classes in C++ are often large, contain many functions, and are split across several files. It is standard practice to have function prototypes only in a header file for a class, and function definitions in a separate `.cpp` file.

Perhaps none of this makes sense at this point, eh, the curriculum ordering in this class could use some work.
Recall:

```cpp
#include <iostream>

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

int main()
{
    int a, b;
    cout << "Enter two integers: '" << endl;
    cin >> a >> b;
    cout << "Summation = '" << add(a, b);
    return 0;
}
```

How can we rewrite this using a function prototype?
Example

Answer:

```cpp
#include <iostream>

int add(int x, int y); // function prototype

int main()
{
    int a, b;
    cout << "Enter two integers: " << endl;
    cin >> a >> b;
    cout << "Summation = " << add(a, b);
    return 0;
}

int add(int x, int y) // function definition
{
    return x + y;
}
```

We will revisit this as we develop more advanced concepts in C++ such as classes.

Thomas Merkh tmerkh@math.ucla.edu
May 5, 2020
#include <iostream>

int add(int x, int y); // function prototype

int main()
{
    int a, b;
    cout << "Enter two integers:"
        << endl;
    cin >> a >> b;
    cout << "Summation = " << add(a, b);
    return 0;
}

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

We will revisit this as we develop more advanced concepts in C++, such as classes.
+ Functions

- Function Prototypes

- Modular Programming

- Next Time
Modular Programming

Definition

Modular programming is a software design paradigm that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

- TLDR: Breaking your code up into many simple functions > having a few complicated blocks of code

Why? As your projects grow in size, testing individual components will become essential. Good organization equates to days, weeks, or months of your life saved.
Modular Programming

Just to really sell this paradigm, some pros:

- **Ease of Use**: This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.

- **Reusability**: It allows the user to reuse the functionality with a different interface without typing the whole program again.

- **Ease of Maintenance**: It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

One potential drawback: If a module (function) isn’t throwing an error but is slightly incorrect, it may be hard to find the bug since it may show up everywhere in one or many programs.
Remember that functions have types (and only a single type)? Can we prevent rewriting the same code multiple times, once for each type?

For example,

```
int sumInt(double a, double b);
long sumLong(double a, double b);
double sumDub(double a, double b)
```

**Question**

*Can we create a function-like thing which is general enough where we can specify the output type?*
That is exactly what this code does:

```cpp
#include <iostream>
using namespace std;

template <class dtype> dtype sumTemp(dtype x, dtype y)
{
    return x + y;
}

int main(void)
{
    int x1 = 10; int y1 = 1;
    long x2 = -110; long y2 = 10;
    double x3 = 10.5; double y3 = -0.3141592;

    cout << sumTemp<int>(x1, y1) << endl;
    cout << sumTemp<long>(x2, y2) << endl;
    cout << sumTemp<double>(x3, y3) << endl;
}
```
Bonus Topic: Templates

Although I won’t go into detail here, templates are useful when one has many functions which all have essentially the same functionality or structure, but could possibly have a well-defined meaning on different types of inputs. Some benefits include

- Templates even further improve the general reusability of a code (never write code twice)
- Less functions/less lines of code generally translates to easier debugging
- Passing that coding interview when you are looking for a job :)

Thomas Merkh tmerkh@math.ucla.edu

May 5, 2020 20 / 22
1. Functions

2. Function Prototypes

3. Modular Programming

4. Next Time
Things to do between now and then:

- Function parameters. Optional arguments. Procedures