Expanding your program beyond just main()

CS 10A – FUNCTIONS AND VARIABLES

Allen Zhao – SRJC CS 10A

Introduction to Functions

- All commands used in C++ up to this point have taken place inside main(). After a while, a program becomes too big and too complex to be handled by main() alone.
- Commands can be grouped together to form code blocks called functions. Functions exist outside main(), and can be used multiple times for common operations.
- Functions greatly help with organizing your code and reducing the number of lines needed to write a program.
- Functions can also handle inputs and outputs if needed.
- Generally, the mark of a function in C/C++ is parentheses following a string name.

Rules for Writing Functions in C/C++

- int main() is a function that exists pre-defined in almost every language, serving as the primary execution block.
- Functions are written outside the main() block, either before or after it. Naming rules are the same as they are for variables.
- However, because the compiler reads top to bottom, all custom made functions must be declared before they are used inside the main() block.
- How, where, and when functions can be used can get fairly nuanced, but for now, any function declared in the same file as your main() block can be used freely within main().
- Just like variables, functions have types as well, which determines what kind of data the function can output.

Functions Before main()

```
void hello()
{           // Unlike loops and logic statements, you must always enclose the commands of a function inside brackets.
            cout << “Hello World” << endl;
}           // Functions written before main() are both declared and defined at the same time.

int test()
{           // Any number of lines can be executed as part of the function, top to bottom.
            return 1;
}           // Functions written before main() do NOT execute unless main() explicitly calls for them.

int main()
{
            hello();    // Function call, can take place any line after the function of that name has been declared
            test();     // Another function call

            return 0;
}
```

Functions After main()

```
void hello(); // This is known as a function prototype. It tells the compiler that a function named hello() exists somewhere in the code.
int test();   // If you want to define your functions behind the main() block, you must declare the prototype before main() in order to use it.

int main()
{
        hello();
        test();
        return 0;
}

void hello()
{
        cout << “Hello World” << endl;
}       // Writing the function before main(), prototype or otherwise, does not execute the function. It must be called by main() to run at all.

int test()
{       // Any number of lines can be executed as part of the function, top to bottom.
        return 1;
}       // A function immediately ends with a return statement.
```

Return Values (aka Function Outputs)

- You probably have noticed that all functions start with a type as part of their definition. This is their defined return value, also known as the function's output.
- main() has a return type of int, so therefore, the function block returns some integer value after the block finishes.
  - The return 0 line we've been using all this time is just that: main() outputting a 0 back to the system when it finishes.
- Functions can use most variable types as a return type and also one more: void.
  - Void is exactly what it sounds like – nothing. A void function just executes something but does not give a value back once finished.
  - As such, void functions do not contain a return statement.
- Executing a return statement immediately terminates the function.

Input Parameters

- A function isn’t just necessarily a static block of commands to execute. It can accept inputs too.
- Each item/variable a function can accept as an input is known as a parameter.
- A function can any number of input parameters.
- Input parameters are declared like variables, but they are given their own name and can only be used locally. They’re essentially placeholders for the real values.
- A function with inputs does not have to have an output and vice-versa.
- All parameters declared must be part of the prototype too.

Defining Input Parameters - Examples

```
int add(int a, int b)                    // Each parameter is separated by a comma
{
        return a+b;
}

void test(string str)                    // Parameters are placeholders to be used within the function
{
        cout << str << endl;
}

double average(double x[], int size)     // You can pass arrays as parameters, but cannot use sizeof() on the parameter
{
        double sum = 0;
        for(int i = 0; i < size; i++)
                sum += x[i];
        return sum/size;
}
```

Utilizing Functions

```
// Use the functions as defined on the previous slide.
int main()
{
        // The function is treated as its returned value when used.
        cout << add(3, 4) << endl;     // Prints out 7 to the console

        // Make sure that parameters and their types for both input and output are consistent
        test("Hello World");

        double arr[] = {1, 2, 3, 4, 5, 6};
        double avg = average(arr, sizeof(arr)/sizeof(arr[0]));
        // Returned values from a function must be stored or used right away to be useful

        return 0;
}
```

Common Misconceptions

- A function parameter is a variable that serves as a placeholder for use within the function ONLY. It does not need to be declared before the function prototype.
- If you declare a function parameter and give it the same name as another variable that exists outside the function, they will NOT be the same variable.
- Never declare two variables of the same name in one program just to force your program to work. Remember that the point of a function (for now) is to pass VALUES from one variable to another, not the variables themselves.

Suggested Coding Habits for Functions

- A function can be as short as one command. Since function names can be comments as much as variable names are, it's not a bad idea to have a bunch of one line functions so that your code can read like plain language.
- While it takes up more lines, I recommend defining functions after the main() block. It allows whoever is reading your code to quickly find the main() block instead of having to scroll past a bunch of custom functions first.
- Outside this class's scope, but once you learn how to incorporate multiple files into one program, you should define and categorize your functions in those external files as opposed to the same file where main() is written.

Variable Scope

- With custom functions, WHERE you declare a variable will now be critical to where you can use it.
- The scope of a variable determine which parts of a program can access a variable at any given time. There are multiple aspects of determining the variable's scope.
- For this class, we'll just cover global vs. local variables.
  - A global variable is a variable that can be accessed by any function at any time.
  - A local variable is a variable that can be accessed only by the function where it was originally declared.

Local vs. Global Variables

```
int x, y;                       // Any variable declared outside of all functions is global
int main()
{
        int i, j = x-y;         // Variables declared inside a function are local to that function
        x += foo(i, j);         // Local variable values can be passed to other functions
        return x;
}

int foo(int a, int b)           // Parameter variables are considered local
{
        int m = x+y;
        int n = a+b;            // Passing a value is not the same thing as passing a variable
        return m+n;
}
```

Suggested Coding Habits for Variable Control

- Because global variables can be accessed anywhere, anytime, this can introduce some weird bugs in larger projects or whenever array manipulations are needed, and thus are generally discouraged.
- By default, you should declare all of your variables locally. Move a variable to global if and only if you have a concrete reason for doing so.
- Temporary placeholder variables are often useful as global variables. It’s like a communal trash bin.
- Critical data should always be in local variables.

Efficiency vs. Memory Trade-Off

Less Efficient, Less Memory

```
int sum(int arr[], int size)
{
		int sum = 0;
		for(int i = 0; i < size; i++)
				sum += arr[i];
		return sum;
}

int main()
{
		int input[] = {1, 2, 3, 4, 5, 6};
		// Would have to recalculate every time
		for(int i = 0; i < sum(input[], sizeof(input)/sizeof(input[0])); i++)
				cout << sum(input[], sizeof(input)/sizeof(input[0]));
		return 0;
}
```

More Efficient, More Memory

```
int sum(int arr[], int size)
{
		int sum = 0;
		for(int i = 0; i < size; i++)
				sum += arr[i];
		return sum;
}

int main()
{
		int input[] = {1, 2, 3, 4, 5, 6};
		int x = sum(input[], sizeof(input)/sizeof(input[0]));
		// Calculate only once
		for(int i = 0; i < x; i++)
				cout << x;
		return 0;
}
```