Functions: Basics

CS 1044

What is a function?

- **From your perspective so far:** “Magic” names that provide some sort of calculation, you don’t have to understand the implementation just call them

- **Basic Definition:** A function is a sequence of statements that have been grouped together and given a name. ex. `pow(), exp()`

- **More info:** Each function is essentially a small program, with its own declarations and statements

Why Define Functions?

- **Increase readability:** Breaks down your program into smaller, more manageable pieces
- **Reduce repetition:** If you have the same or similar looking code in a lot of places, move it into its own function and call it from those places instead
- **Provide external access:** If the code you're writing is a library that you expect others to use, you would provide functions that they would be expected to call

```
int main()
{
    int a, b, c, d, e, f;

    // Unnecessary repetition computing
    // when computing the average, often
    // error prone.
    cout << (a + b) / 2;
    cout << (c + d) / 2;
    cout << (e + f) / 2;

    // or

    // This is easier to read, and it's
    // harder to make a mistake.
    cout << average(a, b);
    cout << average(c, d);
    cout << average(e, f);

    return 0;

}
```

```
// The control flow of the program and what
// each function does is clear from the
// function names increasing readability.

int main()
{
    // Read the input.
    int data = read_file(filename);

    // Process the data somehow.
    int output = process_data(data);

    print_output(output);

    return 0;

}
```

```
// The advantage of this becomes clearer
// when dealing with more complex programs
// with multiple calls to the same function.

int main()
{
    // Read the input.
    int data1 = read_file(filename1);
    int data2 = read_file(filename2);
    int data3 = read_file(filename3);

    // Process the data somehow.
    int output1 = process_data(data1);
    int output2 = process_data(data2);
    int output3 = process_data(data3);

    print_output(output1);
    print_output(output2);
    print_output(output3);

    return 0;
}
```

Defining Functions

```
type name(parameters)
{
    body;
}
```

This line is the function's **header** or **signature**

- ***type***: The type of the result that the function returns
- ***parameters***: Zero or more input values; include the **type** and **name**, just like variable declarations
- ***body***: Code executed when the method is called

The `main` function

- You've already been writing functions in your programs; every C++ program must have a `main` function

```
int main()
{
    // stuff
    return 0;
}
```

- What is its return type? What are its parameters?

`void` Functions

```
void name(arguments)
{
    body;
}
```

- Functions with return type `void` do not return a value
- Think of it representing an **action** – it **does something** but doesn't necessarily compute a result

```c
// Our example from before print_output
// is a good candidate for a void function.

int main()
{
    // Read the input.
    int data = read_file(filename);

    // Process the data somehow.
    int output = process_data(data);

    print_output(output);

    return 0;

}
```

```
// Lets write print_output
// as an example of a void function.


void print_output(int out)
{
    cout << out << endl;

}
```

void return type, no values will be returned

one argument or parameter: an **int**

I can use the parameter like any other variable

Declaring vs. Defining

- For historical reasons, C++ is pretty **lazy** when it comes to scanning your program for functions
- You can’t call a function at a point in your code earlier than you declare or define it

Declaring vs. Defining

```
int main()
{
    cout << average(100, 200);
    return 0;
}


double average(int a, int b)
{
    return (a + b) / 2;
}
```

Error: `average` not defined

WTF? It's right here!

Declaring vs. Defining

- We could get around this particular problem by **switching the order** of the two functions
- But, what if I have two functions **A** and **B** – **A** calls **B** and **B** calls **A**
- I **can't** define each function before the other one

Prototypes

- In C++, we can declare a function without defining it by writing its **prototype**
- The prototype looks the same as the function header: return type, name, and parameters
- But, it's followed by a **semicolon** instead of the body
- Hint to the compiler: "Here's what the function looks like for anyone who wants to call it – the definition will come later"

Prototypes

```
double average(int a, int b);

int main()
{
    cout << average(100, 200);
    return 0;
}

double average(int a, int b)
{
    return (a + b) / 2;
}
```

All good!

Function Organization

- As programs get larger, **organizing your functions** becomes important for readability
- Here's a good organizational plan:
  - `#include`s and such
  - Prototypes for all your functions, except `main`
  - Definition of your `main` function
  - Definition of your other functions

`return` Statement

- `return` exits the function immediately and passes control back to whomever called it
- Usually at the end of a function, but can also come in the middle if you need to bail out early
- If the function is not `void`, you must have an expression after `return` that represents the value to pass back

Decomposition

- Break a problem down into manageable **subproblems**
- **Stepwise refinement:**
  - Divide problem **A** into subproblems **A.1**, **A.2**...
  - Can/should **A.1**, **A.2**, ... be broken down further?
  - If so, repeat
- Need for **reuse** is also a good way to identify subproblems