COMP 1633: Intro to CS II

Functions in C++

Charlotte Curtis

January 17, 2024

Where we left off

- `cout` and `cin`
- Debugging with `gdb`
- Constants
- Type casting
- Basic compiling and arithmetic

```
#include <iostream>
using namespace std;
#define PI 3.14159

int main() {
  double r;
  cout << "Enter the radius: ";
  cin >> r;
  cout << "The area is " << PI * r * r << endl;
  return 0;
}
```

Today's topics

- Predefined functions in C++
- Function calls
- Declaring and defining functions
- Separate compilation (briefly)

> *Textbook Sections 4.1-4.5*

Program execution

- The requisite `main` function is the program entry point
- Code in `main` is executed one line at a time from top to bottom
- `main` should be reserved for the program's main logic, with various sub-tasks delegated to other functions

```
int main() {
    double a, b;
    cout << "Enter the two side lengths of a right angle triangle: ";
    cin >> a >> b;
    cout << "The hypotenuse length is " << hypoteneuse(a, b) << endl;
    return 0;
}
```

Predefined functions

- C++ includes a standard library, such as `<iostream>`
  - Caution: Many libraries say C++ 11 or similar, but we are using C++ 98
- To use a library, include its **header file** at the top of your program:

```
#include <iostream>
```

- Just like Python's `import`, this now gives access to all the functions in `iostream`

> *Remember how you needed to access Python functions with `.` syntax, such as `random.randint`? C++ is similar, using `::` instead of `.`, and the `using namespace std` statement lets you skip the `std::` part.*

`<cmath>` functions

Function	Description
`fabs(x)`	Absolute value of `x`
`pow(x, y)`	`x` to the power of `y`
`sqrt(x)`	Square root of `x`
`ceil(x)`	Smallest integer greater than or equal to `x`
`floor(x)`	Largest integer less than or equal to `x`
`round(x)`	Nearest integer to `x`
`sin(x)`	Sine of `x` (and other trig functions)

Data type of output depends on input!

Review: Calling functions 1/2

Given a function `pow(int x, int y)` that returns x^y, what output do you think the following code will produce? Assume the code is part of a complete program.

A. No output

B. 8

C. 9

D. 27

E. Error

```
int x = 2;
int y = 3;
cout << pow(y, x) << endl;
```

Review: Calling functions 2/2

Given a function `pow(int x, int y)` that returns x^y, what output do you think the following code will produce? Assume the code is part of a complete program.

A. No output

B. 8

C. 9

D. 27

E. Error

```
int x = 2;
int y = 3;
pow(x, y);
cout << pow << endl;
```

Main takeaway: function calls

General form is similar to Python:

```
return_val = function_name(argument1, argument2, ...);
```

- Whatever is returned from the function is assigned to `return_val`
- Arguments are passed to the function in the parentheses
- Data type and order matters, names do not!
- Arguments can be literals, variables, or expressions

Example: formatted output

```python
print(f"Total: ${bill:.2f}")
```

```cpp
cout.precision(2);
cout << fixed;
cout << "Total: $" << bill << endl;
```

- Calling `cout.precision(n)` sets the number of decimal places to `n`
- `fixed` prints a decimal like `308.24`
  - alternatively, `scientific` prints a decimal like `3.08e+02`
- These are called format flags

Formatting output: field width

```python
print(f"Total:    ${bill:8.2f}")
print(f"With GST: ${bill*1.05:8.2f}")
```

```cpp
#include <iomanip>
cout.precision(2);
cout << fixed;
cout << "Total: $" << setw(8)
     << bill << endl;
cout << "With GST: $" << setw(8)
     << bill*1.05 << endl;
```

- The `iomanip` library provides more functions for formatting output
- `setw(n)` sets the field width to `n` characters
- Unlike `precision`, `setw` only affects the next output

Remember abstraction?

- Abstraction lets us hide the **implementation** from the **interface**
- To *call* a function, we only need to know:
  - What values to pass in
  - What it will return
  - A general idea of what it does
- Similarly, you can *write* a function without ever knowing how it will be used
- This lets us break big problems into smaller ones, and reuse useful bits of code

Declaring functions

- Just like variables, C++ requires functions to be **declared** before they are used
- This tells the compiler that the function exists and how it behaves
- Similar to a function header in Python:

```
def func_name(args) -> return_type
```

```
return_type func_name(args);
```

- A function declaration is also called a **prototype**
- All function declarations must be placed before `main`, and ideally in a separate **header file**

Defining functions

```python
def func_name(args) -> return_type:
    # function body
    return return_val
```

```javascript
return_type func_name(args) {
    // function body
    return return_val;
}
```

- The declaration and function header are almost identical except:
  - No semicolon after the function header
  - Variable names are required

`void` functions

What if you don't want to return anything?

```python
def say_hello() -> None:
    print("Hello!")
```

```cpp
void say_hello() {
    cout << "Hello!" << endl;
}
```

- `void` is an explicit return type that means "no return value"
- The `return` statement is **optional**
- Otherwise, just like Python!

> *Caution: Python's return types are just a suggestion, while in C++ they are strictly enforced.*

Complete program example with functions

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

double hypoteneuse(double a, double b); // function declaration

int main() {
  double a, b;
  cout << "Enter the two side lengths of a right angle triangle: ";
  cin >> a >> b;
  cout << "The hypotenuse length is " << hypoteneuse(a, b) << endl;
  return 0
}

double hypoteneuse(double a, double b) { // function definition
    return sqrt(a*a + b*b);
}
```

A brief preview of Separate Compilation

- We can separate the main logic from other logical groupings
- Problem: `main` needs to know about the existence of other functions
- Solution: put all the declarations in a **header file** (`.h`), then `#include` it
- Header files should **only contain**:
  - function prototypes
  - type definitions (e.g. `struct`s)
  - named constants
- No variables or function definitions should go in header files!

Separate Compilation

- New project structure:
  - `defs.h`
  - `defs.cpp` - `#include "defs.h"`
  - `main.cpp` - `#include "defs.h"`
- Prevents duplication of the code in `defs.h`, keeps main logic clear
- Compile in multiple steps:
  - `g++ -c defs.cpp` - compiles `defs.cpp` into `defs.o`
  - `g++ -c main.cpp` - compiles `main.cpp` into `main.o`
  - `g++ -o main main.o defs.o` - links the two object files

`make`

Compiling in multiple steps is a tedious process, so we automate it with a **makefile**

```
# This is "Makefile". Notice that comments begin with "#"
program: defs.o main.o
    g++ main.o defs.o –o program
main.o: main.cpp
    g++ -c main.cpp
defs.o: defs.cpp
    g++ -c defs.cpp
```

- Instead of running `g++` to compile, run `make` (with no arguments)
- This allows me to do autograded labs! For now, I'll provide makefiles and all you have to do is run `make`. You can ignore the contents of the makefile for now.

Tangent: Curly brace convention

- The curly braces `{}` are required to define blocks, but indentation is not
- The convention is to indent the contents of a block by 4 spaces
- Up to you whether the first `{` is on the same line or the next:

```
int main() {
    // ...
}
```

```
int main()
{
    // ...
}
```

- As usual, be consistent!

Refresher: Variable scope

As in Python, variables defined in a function (including parameters) are only accessible within that function:

```
int main() {
    int x = 5;
    int y = some_func();
    return 0;
}

int some_func() {
    return x * 2; // Error: x is not defined
}
```

> *We'll talk more about scope next lecture*

Coming up next

- Lab: Functions 🎉 - first lab with actual tests
- Lecture: Pass by reference, scope

> *Textbook 4.5, 5.1-5.2*