Unit 6

Thinking with Functions
Functions Overview

- Functions (aka procedures, subroutines, or methods) are the unit of code decomposition and abstraction
  - **Decomposition**: Breaking programs into smaller units of code
  - **Abstraction**: Generalizing an action or concept without specifying how the details are implemented
Recall: Walking a Square in Scratch

• We can define a function (i.e. block of code) once and then "call" it any time we want to execute that block of code.

• Can provide different input values (aka "arguments" / "parameters") and even get an output (aka "return" value).
Function Signatures/Prototypes

• We think of a function as a blackbox (don't know or care how it does the task internally) where **we can provide inputs and get back a value**

• A function has:
  – A **name**
  – Zero or more input parameters
  – 0 or 1 **return** (output) values
  • We only specify the type

• The signature (or **prototype**) of a function specifies these aspects so others know how to "call" the function
"Functional" Programming

• While we can write arithmetic expressions directly in C++, let's practice using functions to perform the same operations.

• Suppose you are given:
  – add(p, q) // returns p+q
  – sub(p, q) // returns p-q
  – mul(p, q) // returns p*q
  – div(p, q) // returns p/q

• Convert the following expressions to use functions and no operators (+, -, *, /)

• Key Ideas:
  – Execution works from inside to outside (i.e. f(g(x)) invokes g(x) first)
  – The return value of a function is substituted and used in the larger expression

```
// Add 3 numbers
a = x + y + z;
// which upholds the order of ops
a = add(add(x,y),z);
a = add(x,add(y,z));
```

```
// Exercise 1
a = x / y + y * z - x;
a = ________________________________
```

```
// Exercise 2
a = x * (y - z) / z;
a = ________________________________
```

Disclaimer: These functions (add, sub, etc.) are fictitious and in C++ we just use the +, -, etc. operators, but this is to practice using functions.
Function call statements

- Reminder that you can call a function anywhere
- Result is replaced into bigger expression
- Take care to "save" the result
  - If you don't save the return value into a variable or use it immediately, the result is lost

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

int main()
{
    // can call functions
    // in an assignment
    double res = cos(0); // res = 1.0
    // can call functions in an
    // expression
    res = sqrt(2) / 2; // forgot to save result
    // save 1.414/2 in res
    cout << max(34, 56) << endl; // outputs 56
    return 0;
}

http://www.cplusplus.com/reference/cmath/
```
User Defined Functions

- We can define our own functions.
- Good practice is to "declare" your function by placing the prototype (signature) at the top of your code.
- "Define" the function (actual code implementation) anywhere by placing the code in {}.
- As shown it is defined but never used (no one actually "calls" the max function).

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

int max(int a, int b); // prototype

int main()
{
    int x, y, mx;
    cin >> x >> y;
    /* Code for main */

    int max(int a, int b)
    {
        if(a > b)
            return a; // immediately stops max
        else
            return b; // immediately stops max
    }
```
Calling a Function (1)

• We "call" or "invoke" the function by:
  – Using its name and place variables or constants that the current function has declared in the order that we want them to map to the parameter/argument list
  – First variable listed (x) will map to the first parameter (a) in the function's argument list, the second variable (y) to the second parameter (b), etc.

• Don't
  – Relist the return type in the call
  – Relist the type of the arguments
  – Use variable names that don't exist in the current function
  – Forget to save the returned value

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

int max(int a, int b); // prototype

int main()
{
    int x, y, mx;
    cin >> x >> y;

    /* Call the function */
    mx = max(x, y);

    /* Bad */
    mx = int max(x, y);
    mx = max(int x, int y);
    mx = max(a, b);
    max(x, y);

} 

int max(int a, int b)
{
    if(a > b)
        return a; // immediately stops max
    else
        return b; // immediately stops max
} 
```
Calling a Function (2)

• The we can "call" (activate/invoke) our function from any other location in our code as many times as we like

• Semantics of a function call:
  – Pause the caller code (i.e. main)
  – Pass copies of the arguments to the function (i.e. pass a copy of x and y to a and b)
  – Let the function execute
  – When the function completes we return back to the caller (i.e. main) and resume execution
  – Any return value is substituted in place of the function call

• Notice 2 sets of parameter names:
  – "Formal names": Generic parameter names used as placeholders to refer to an argument inside the code (e.g. a title like "CEO" or "parent" that refer to a specific person"
  – "Actual" names: True parameters that the formals will reference (e.g. "Jeff Bezos", "Mark Zuckerberg", etc.)

#include <iostream>
using namespace std;

int max(int a, int b); // prototype

int main()
{
  int x, y;
  cin >> x >> y;  // User types: -5 7
  int mx = 1 + max(x, y); // call max
  cout << mx << endl;
  cout << max(0, x) << endl; // call max
}

int max(int a, int b)
{
  if(a > b)
    return a; // immediately stops max
  else
    return b; // immediately stops max
}

Program Output (if user types -5 7):

8
0
Execution Timeline

1. We always start at main() and execute sequentially until a function call or the end of main()
2. When we hit a function call, execution of main pauses and execution of max begins
3. Max will compare the arguments and return the bigger
4. Upon return we go back to where we left off in the previous function and replace the function call with the return value
5. We continue to evaluate the expression in which the function call was made (i.e. 1 + max)
6. We assign the result to mx
7. We continue sequentially until another function call or until the end of main()
8. Another function call again causes main to pause and max begins execution anew
9. Max compares arguments
10. It then returns to the caller (main) and substitutes its return value in the larger expression
Example Functions 1

Function Signature/Prototype

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

// prototype
double calcInterest(double amt, int yrs, double rate);

int main()
{
    double amount, r;
    cin >> amount >> r;
    double interest = calcInterest(amount, 30, r);
    cout << "Interest: " << interest << endl;
    return 0;
}

double calcInterest(double amt, int yrs, double rate)
{
    return amt * pow(rate/12, 12*yrs);
}
```
Example Functions 2

Function Signature/Prototype

```cpp
bool getAndCheckLogin(string exp_pwd);
```

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

// prototype
bool getAndCheckLogin(string exp_pwd);

int main()
{
    string pass = "Open123!"; // secret password
    bool valid;

    valid = getAndCheckLogin(pass);
    if(valid == true) { cout << "Success!" << endl; } else { cout << "Password mismatch!" << endl; } return 0;
}

bool getAndCheckLogin(string exp_pwd)
{
    string pwd;
    cout << "Enter your password: " << endl; cin >> pwd;
    return pwd == exp_pwd;
}
```
# Example Functions 3

## Function Signature/Prototype

```cpp
void printLoginResult(bool correct);
```

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

// prototype
void printLoginResult(bool correct);

int main()
{
    string pass = "Open123!"; // secret password
    bool valid;

    valid = getAndCheckLogin(pass);
    printLoginResult(valid);
    return 0;
}

void printLoginResult(bool correct)
{
    if(correct == true) {
        cout << "Success!" << endl;
    } else {
        cout << "Password mismatch!" << endl;
    }
}
```
```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

// prototype
bool genCoinFlip();

int main()
{
    bool heads;

    heads = genCoinFlip();
    if(heads == true) { cout << "Heads!" << endl; }
    else { cout << "Tails!" << endl; }
    return 0;
}

bool genCoinFlip()
{
    int r = rand(); // Generate random integer
    return (r%2 == 1);
}
```
Exercises

• Exercises
  – hypotenuse
  – wakeup
Reading Documentation

• Much of programming is calling other library functions which do small pieces of work in an effort to accomplish the overall application
  – Learn to read documentation

• Documentation at:
Predicate Functions

- We can write functions that return a bool (true or false) to assert or confirm something about the input.
- These functions can then be called in the condition of an if statement or a loop.

```cpp
bool isNegative(double x) {
    return x < 0;
}

bool isDigit(char c) {
    return (c >= '0' && c <= '9');
}

bool _________(int s1, int s2, int s3, int s4) {
    return (s1 == s2) && (s2 == s3) && (s3 == s4);
}

int main() {
    int a;  char b;  int f, g, y, z;
    cin >> a >> b >> c;
    if( isNegative(a) ) {
        cout << "Error...neg. #" << endl;
    }
    if( (f==g) && (g == y) && (y == z) ) {
        cout << "Yes...___________" << endl;
    }
    return 0;
}
```
SOLUTIONS
"Functional" Programming

- While we can write arithmetic expressions directly in C++, let's practice using functions to perform the same operations.

- Suppose you are given:
  - add(p, q) // returns p+q
  - sub(p, q) // returns p-q
  - mul(p, q) // returns p*q
  - div(p, q) // returns p/q

- Convert the following expressions to use functions and no operators (+, -, *, /)

- Key Ideas:
  - Execution works from inside to outside (i.e. f(g(x)) invokes g(x) first
  - The return value of a function is substituted and used in the larger expression

// Add 3 numbers
a = x + y + z;
// which upholds the order of ops
a = add(add(x,y),z);
a = add(x,add(y,z));

// Exercise 1
a = x / y + y * z - x;
a = sub(add(div(x,y),mul(y,z)),x);

// Exercise 2
a = x * (y - z) / z;
a = div(mul(x, sub(y,z)), z);

Disclaimer: These functions (add, sub, etc.) are fictitious and in C++ we just use the +, -, etc. operators, but this is to practice using functions.