Discussion 1B Notes (Week 4): Functions

TA: Zhou Ren

Acknowlegement: Brian Choi's material

Suppose I want to write a program that computes 3 things:
- the sum of integers from 1 to 100
- the sum of integers from 51 to 150
- the sum of integers from 103 to 276

You can create a loop for each one of the sums, but this is cumbersome. If we are allowed to define a method of "summing up" numbers just once, and then use it three times, it will reduce some burden. In other words, you can teach your computer "how to sum," as opposed to "what to sum."

In C++, **functions** (or equivalently known as **methods**, **subroutines**, or **procedures** in other contexts) are meant to serve this purpose.

An example of a C++ function is shown here **---->**

```
int sum(int begin, int end)
{
   int temp = 0;
   for (int i = begin; i <= end; i++)
   {
      temp += i;
   }
   return temp;
}
```

Hey, that resembles the `main()` thing that we used to write. In fact, `main()` is also a function, which we call "**main function**," and it happens to be the one that gets executed when the program begins.

Before we take it apart, let me bring up an analogy:

This is called a "function" for a reason. Where else do you see "functions"? In high school algebra, we've seen functions presented in the following way:

$$f(x) = 2x + 5$$

This function, called f, takes a value, denoted x, and maps this number in a predetermined way to another number. If the input x is 5, for instance, we do:

$$f(5) = 2(5) + 5 = 15$$

and it would result in a different number for a different output. Once we have defined this function f, we can simplify other expressions using f.

$$4x + 10 = 2f(x)$$
$$2x^2 + 7x + 5 = (x+1)(2x+5) = (x+1)f(x)$$

This is exactly what we do with our C++ functions. We define it, and then use it to simplify our program.

Now let's take the function definition apart.
- `int` indicates that the output is of type integer. We call this **return type**.
- `sum` is the name, or the **identifier**, of the function, just like f is for the function above.
- `begin` and `end` are **arguments** to the function `sum`, just like x is for the function f. The arguments must be within parentheses.
- The **function body** within `{ }` defines the "mapping rule."
- The function must `return` the value explicitly, depending on the input it receives.

In general, the syntax goes as follows:

```
return_type function_name(arg1, arg2, arg3, ...)
{
   function_body
}
```

and the function body must include return statements such that every execution of the function ends up in a return statement. A function can also have no arguments, in which case the argument list simply becomes empty (i.e., `()`).

So where in the program should this function go and how do we use it? Here is an example.

```
#include <iostream>
using namespace std;
int sum(int begin, int end)
{
   int temp = 0;
   for (int i = begin; i <= end; i++)
   {
       temp += i;
   }
   return temp;
}

int main()
{
   cout << sum(1, 20) << endl;       // prints the sum 1 + ... + 20
   cout << sum(20, 50) << endl;      // prints the sum 20 + ... + 50

   int sum1, sum2;
   sum1 = sum(4, 8);                 // stores the sum 4+5+6+7+8 into sum1
   sum2 = sum(1, 10) + sum(15, 20); // stores the sum 1+...+10+15+...+20 into sum2
   cout << sum1 << " " << sum2 << endl;
   return 0;
}
```

Note that the function must be defined before it can be used. If the definition of sum comes after the main function, the compiler will complain that `sum` is an undefined identifier.

If for a style reason you want to keep the main function on the top and define other functions later, you can "let the computer" know that there's going to be a function with a certain name first, and define later. For instance, you can do what's on the right.

The bolded line is called a **function prototype**.

```
#include <iostream>

// I will define "sum" later.
int sum(int begin, int end);

int main()
{
    ...
}

// Define it here.
int sum(int begin, int end)
{
    ...
}
```

Now it's time for a little practice:

Question: Define a function `f` that takes in a real number `x` and returns the result of `2x + 5`.

Question: Use the function `f` and `sum` to find the sum from f(3) to f(72), and store it in a variable called `z`.

Question: How many #'s will you see on the screen?

```
int mystery(int len)
{
   for (int i = 0; i < len; i++)
   {
      cout << "#";
   }

   if (len % 2 == 0)
      return len + 1;
   else
      return len + 3;
}

int main()
{
   int y = 5;
   while (y < 10)
   {
      y = mystery(y);
   }
   return 0;
}
```

Passing Arguments By Value and By Reference

So far, we have been passing in "values" into functions. We say we **pass arguments by value**. This method does not allow you to access variables outside. We will make a way to access an outside variable now.

```
double rush(double currentTotal, double yards, int &count)     // ampersand!
{
  count++;
  return currentTotal + yards;
}

int main()
{
  double rushTotal = 0;
  int rushCount = 0;

  rushTotal = rush(rushTotal, 10.5, rushCount);
  rushTotal = rush(rushTotal, 5.3, rushCount);
  rushTotal = rush(rushTotal, 7.2, rushCount);

  cout << "Our running back ran " << rushCount << " times and the total of "
       << rushTotal << " yards." << endl;

  return 0;
}
```

Output:

What is that ampersand in the function definition? As usual, when the function `rush` is called, it will start by creating the variables for the arguments – `currentTotal`, `yards`, and `count`. Then it sees the ampersand before `count`. This means it is a **reference** to a variable, not a real variable. The variable it references to is what is passed from `main()` – in this case, `rushCount`. `count` is like a window that leads to the box labeled `rushCount` in the main world! (In effect, we are returning two values here.)

In the function `rush`, whenever we access `count`, we are actually accessing `rushCount` in main. When the function terminates, we'll destroy the memory space for `rush`, but the changes made to `rushCount` will remain.

We say that `count` is **passed by reference**. When in doubt, draw the "box" diagram.

void functions

There are functions that do not have to return any return value. For example, it might be some function that displays a few lines of text such as "About this program: Copyright 2008 blah blah..." message. You can create such a function by making the return type `void`. You do not have to have a `return` statement, but if you need a return statement to terminate the function, you can just say "`return;`" without any value.

More on Strings

Here are things you can do with strings.

operation	what it does	example

`string s = "hello";`
`string s2 = "!!!";`
	declares strings `s` and `s2`	

`s.length() or s.size()`	returns the length of `s`	`cout << s.size(); // prints 5`

`s[i] or s.at(i)`	returns `i`-th character `i` must be an integer between 0 and size-1 (inclusive).	`cout << s[1];     // prints 'e'`
		`cout << s.at(0);  // prints 'h'`

`s.empty()`	returns true if `s` is empty	`if (!s.empty())`
		`  cout << "not empty";`

`s + s2`
`s + "?"`
	concatenates two strings	`cout << s + s2;   // prints "hello!!!"`
		`cout << s + "?";  // prints "hello?"`

`s.substr(i, n)`
`s.substr(i)`
	takes a substring of length n, starting from the `i`-th character	`cout << s.substr(2,2);  // prints "ll"`
		`cout << s.substr(2);   // prints "llo"`

`s.replace(i, n, s2)`	replaces a substring of length `n` starting at `i` with another string s2, and <u>sets `s` with a new string</u>	`s.replace(2, 2, s2); // sets s to`
		`                     // "he!!!o"`

There are a few more functions, but this set of functions should be enough for our purposes.

Now here are some functions you can call on characters, after including `<cctype>` library:

operation	what it does
`char c;`	declares a character `c`
`isspace(c)`	true if `c` is a whitespace character (space, tab, newline)
`isalpha(c)`	true if `c` is a letter
`isalnum(c)`	true if `c` is a letter or digit
`islower(c)`	true if `c` is a lowercase letter
`isupper(c)`	true if `c` is an uppercase letter
`tolower(c)`	returns the lowercase version of `c`, if `c` is a letter
`toupper(c)`	returns the uppercase version of `c`, if `c` is a letter

ASCII and Characters

Remember, everything is represented in 0s and 1s in computers. That means characters must be stored as some binary numbers as well, and ASCII defines the mapping between characters and integers. (It is just one mapping scheme -- there are others (e.g. EBCDIC, Unicode)).

For example, a character ‘0’ maps to 48 in integer, and ‘A’ maps to 65, etc. The mapping is not random: because ‘0’ maps to 48, ‘1’ maps to 49, ‘2’ maps to 50, and the list goes on. Similarly, ‘B’ maps to 66, ‘C’ maps to 67, following the mapping of ‘A’.

You don’t need to know the mapping by heart, though. C++ (as well as other programming languages) understands the mapping well, and it does the conversion for you when you try to store a character into an integer.

```
int x = ‘0’;
```

will store the number 48 into `x`.

Given this information can you convert a digit character to the “right” integer? That is, can you take ‘3’ and somehow convert it into an integer 3? Of course, if you know the ASCII map, you could do:

```
int x = ‘3’;
x -= 48;
```

But this program won’t work anymore if some other mapping scheme is used. We can do the following:

```
int x = ‘3’ - ‘0’;
```

I hope this is obvious -- ‘3’ converts to whatever ASCII number it corresponds to, and ‘0’ converts to whatever ASCII number it corresponds to: the important fact we must exploit is that the deficit is going to be 3 (and it holds for other mapping scheme), which is all we care about.

Now, can you write some functions?

1. Write a function that takes in a string and prints out every other character in the string.

```
void printEveryOther(string s)
{




}
```

2. Write a function that reverses a string and returns it.

```
string reverse(string s)
{




}
```

3. Write a function that takes in a string which only has digits and returns the corresponding positive integer. If it contains a non-digit character, return -1. Assume the number is not bigger than what an integer variable can store.

```
int strToInt(string s)
{















}
```

4. Write a function that returns true if the given string is a **palindrome** (a string whose reverse is the same -- e.g. "level", "racecar"), false otherwise. Assume there is no white space in the string. An empty string is a palindrome.

```
bool palindrome(string s)
{











}
```

Functions FAQ

Q: Where do we define functions?
A: There are two conventional ways, which are equivalent. The requirement is that the function must be defined before it can be used, just like variables.
So you either completely define it before the function is used, or add the **prototype** and define it later in the program. The prototype is a way of telling your compiler that there is such a function, but that we will define it later. Remember to add a semicolon after the prototype, but not after the function header. If you take the program on the left, and move the definition of `func1` below `main`, it won't work.

```
#include <iostream>
#include <string>

using namespace std;

int func1(int x, string str)
{
    // function definition omitted
}

int main()
{
    int y = func1(5, “hello”);
    ...
}
```

```
#include <iostream>
#include <string>

using namespace std;

int func1(int x, string str);

int main()
{
    int y = func1(5, “hello”);
    ...
}

int func1(int x, string str)
{
    // function definition omitted
}
```

Q: I defined the function, why doesn't it run?
A: Defining a function does not imply using it. You must explicitly call (or invoke) the function somewhere to see it running. When you call it, it will be run as you defined it. Where you call it and how you call it depend on you.

Q: Why does the return value not show up on the screen?
A: Because you did not display it, and it's not meant to be displayed. There are people who confusing “returning” with “outputting,” which is different. When you return a value from a function, you return it to whoever called the function.

For example, consider the following function:

```
int func1(int x, string str)
{
   int y = str.size() + x;
   return y;
}
```

Now in the `main` function, suppose I have the following statement,
`int z = func1(5, “hello”);`

Notice that I am **calling** the function `func1` here, and the return value of `func1`, `10` in this case, is returned to this `main` function, exactly where it was called. You can say that the value 10 replaces the function call you have up there:

```
int z = 10;
```

This works because the return type of `func1` is defined to be an integer, and in turn, of the same reason you can treat the whole function call as an integer.

```
int z = func1(5, “hello”);
cout << func1(3, “hi”);
z = 13 + func1(3, “people”);
```

would be the same as:

```
int z = 10;
cout << 5;
z = 9;
```

Again, returning a value does <u>not</u> print out the value.

Common Errors

Do not define a function within a function, since you can’t!

```
int func1(int x, string str)
{
   void func2(int x, string str)
   {
      ...
   }
   ...
}
```

When calling a function, you only need the values, not the types.

```
int main()
{
   int y;
   y = func1(int 5, string “hello”);    // WRONG!
   y = func1(x = 5, str = “hello”);     // WRONG!
   y = func1(5, “hello”);               // CORRECT

   return 0;
}
```

More Problems

5. One can "compare" strings like this:

```
string s1;
string s2;
cout << "Enter two strings: ";
getline(cin, s1);
getline(cin, s2);
if (s1 < s2)
   cout << s1 << " is smaller than " << s2 << endl;
else if (s1 == s2)
   cout << s1 << " is equal to " << s2 << endl;
else
   cout << s1 << " is greater than " << s2 << endl;
```

A string comparison is done based on the lexicographical order, i.e. the order in which they may appear in the dictionary. Exceptions are that uppercase letters are ALWAYS less than lowercase letters are ALWAYS greater than uppercase letters. For instance, a is greater than A, B, ..., Z, and Z is less than a, b, ..., z. Digits are less than alphabets, and ties are broken by letters followed by them. Some examples include:

```
"Zebra" < "alphabet"
"zebra" > "alphabet"
"uniform" < "university"
"alpha" < "alphabet"
"10apples" < "7apples"
```

Write a function that takes in 3 strings `s1`, `s2`, and `s3` by reference, and store them back into `s1`, `s2`, and `s3` in a non-decreasing order.

```
void order3Str(string &s1, string &s2, string &s3)
{









}
```

You might find it helpful to write a helper function `swap(x, y)` as follows:

```
void swap(string &x, string &y)
{
   string temp = x;
   x = y;
   y = temp;
}
```

Quick Question: What do you think you should do to order them in the true dictionary order (such that "Zebra" comes after "alphabet", assuming strings consist only of letters?

6. Write a function `revert()`, which takes in a string and returns a string with all lowercase letters made uppercase, and all uppercase leters made lowercase, while leaving all non-letters the same. For instance:

```
revert(“abc”) --> ABC
revert(“ABc”) --> abC
revert(“pac-12”) --> PAC-12
revert(“oRZ”) --> Orz

string revert(string s)
{









}
```

7. Evaluate the following code and predict the output.
(**Note:** `str.substr(str.size())` returns an empty string.)

```
void mystery(int pos, string &str)
{
  if (pos < 0 || pos >= str.size())
    return;
  cout << str[pos] << “ “;
  str = str.substr(0, pos) + str.substr(pos + 1);
}

int main()
{
  string s = “abcdefg”;
  mystery(3, s);
  mystery(3, s);
  mystery(4, s);
  mystery(1, s);
  cout << endl;
  cout << “s = “ << s << endl;
}
```

8. Write a function called **consonantGroups** that, given a (possibly long) string of text, prints all segments of "consecutive consonants" (let us call them *consonant groups*) in the string, each separated by a character. For example:

`consonantGroups("abracadabra")` prints `br c d br`
`consonantGroups("whatawonderfulworld")` prints `wh t w nd rf lw rld`
`consonantGroups("facebookandgoogle")` prints `f c b k ndg gl`

Note that the function is not simply printing the consonants in the string -- it must group characters that appear consecutively. Further, you should print only <u>one</u> space between two consonant groups. (Notice that there is only one space between **b** and **k** in the last example.) You may have trailing spaces after outputting all consonant groups.

You may assume that the string consists only of alphabets (uppercase or lowercase) with no space characters. Also assume the following helper function is defined:

```
// isVowel(c) returns true only if c is a vowel (a, e, i, o, or u)
bool isVowel(char c);
```

```
_______ consonantGroups(___________________)
{










}
```

Also try implementing `isVowel()`.