Functions and Prime Numbers

Topics: functions, parameters, pass by reference
Lab 5: https://maryash.github.io/135/labs/lab_05.html

- Sadab Hafiz

What are functions?

A function is a named sequence of instructions that performs a specific task and returns the result of its computation. Once defined, it can be then called in your program wherever that particular task should be performed.

A function can receive zero or more arguments. For example, consider a function sum, which receives three arguments named a, b, and c, and returns their sum:

```
int sum(int a, int b, int c) {
    return a + b + c;
}


int main(){
     cout << sum(2,3,4) << endl;                 // prints “9” which is the sum of 2, 3, and 4
}
```

Function Declaration

The function declaration, also known as the function prototype, contains the name of the function, what type of data the function returns and what type of data the function takes. Let’s look at the sum function

```
int sum(int a, int b, int c);
```

From the function declaration, we know the following:

- Function Name: sum()
- Return type: integer
- Parameters: three integer parameters passed by value

There are functions that don’t return anything. Such functions are called void functions. An example is provided on a later slide.

Function Parameters

The arguments given to a function are known formally as parameters. Each parameter contains a data type followed by a name. There can be functions that take no parameters. For example:

```
void printName() {
     cout << “Genady Maryash” << endl;
}
```

If a parameter is passed by value, it can be used like a variable:

```
int sum(int a, int b) {
     a = a + b;                      // add b to a (a+b)
     return a;                       // a is now the sum of a and b
}
```

Formal Parameters vs Actual Parameters

Formal parameters are the parameters in the function definition. Actual parameters are the parameters given to the function during the function call.

```
int sum(int a, int b, int c);
```

In this case, a and b are formal parameters.

```
int main(){

      cout << sum(2,3,4) << endl;                    // calling sum() function

}
```

The actual parameters in this case are 2, 3, and 4. The function would treat the integer 2 as a, 3 as b, and 4 as c.

Passing parameter by reference

So far, the examples showed the parameter being passed by value. There are cases where you want to modify a given parameter so that it changes the actual value of the given data. In such cases, parameters can be passed by reference(&):

```
void increment(int &a) {            // a is passed by reference using & operator
      a++;        // increment given parameter by 1
}
int main(){
      int x = 2;                                // declare variable x as 2
      cout << “Before: ” << x << endl;          // prints “Before: 2”
      increment(x);                             // increment variable x by 1
      cout << “After: ” << x << endl;           // prints “After: 3”
}
```

The value of variable x changed after calling the function!!!

pizza = ⊕

pizza = ⊕

eat(   )

eatByReference(&   )

Passing parameters as `const`

Passing large data by value can be costly. Therefore passing by reference is more space efficient. What if you want to pass something by reference and you don't want the function to modify the parameter? In such cases, use the const keyword to pass a parameter by reference while ensuring that the function won't modify the data that it gets. The parameter is passed as a read-only reference:

```
void increment(const int &a) {          // a is passed as read-only reference
     a++;                               // THIS WILL THROW AN ERROR BECAUSE `a` IS const
}
int main(){
     int x = 3;
     increment(x);                      // leads to an error from increment function
}
```

Scope of a variable inside a function

Whenever you declare a variable inside a function, the variable can only be used within that function after its declaration. Outside that function, the variable is useless. This also applies to the main() function. For example:

```
void func() {

      int y = 0;

}

int main(){

      cout << y << endl;                // WILL CAUSE AN ERROR BECAUSE y IS INSIDE func()

}
```

It is good to know scope variables in general whether or not functions are involved.

Learn more: https://www.geeksforgeeks.org/scope-of-variables-in-c/

Returning from a function

The data being returned must match with the data promised by the function declaration. If the function declaration says it returns an integer, it must return an integer! If it doesn’t, compiler will complain:

```
int func() {                    // func promises to return an integer but returns string instead
      return “1”;               // ERROR SINCE FUNCTION RETURNS STRING FROM INTEGER FUNCTION
}
int main() {
      func();                   // WILL CAUSE AN ERROR BECAUSE func() DOESN’T RETURN AN INTEGER
}
```

The main function is an exception. If you don’t return anything from main(), it returns 0 by default. Some compilers will give a warning for non-returning main() functions.

Returning to exit a void function

Void functions can also `return`. However, no value is returned since it is void. The `return` serves as a way to exit the function early. You will find uses for this at some point if you major in CS.

```
void func(){
      cout << “before returning” << endl;
      return;                                                 // exit out of the function
      cout << “You will never see this!” << endl;             // never executed
}
int main(){
      func();                     // prints “before returning”
}
```

Returning any value from void function will cause compilation error.

Using one function within another function

It is good practice to write functions to prevent code repetition. Writing reusable functions and using one function inside another will significantly improve code readability and debuggability. For example:

```
// singleRow() function would be useful to print many different shapes and would prevent retyping a for-loop each time

string singleRow(int size){                             // function will return  a string with `size` number of stars

       string row = “”;                                       // initialize row as empty

       for (int i=0; i<size; i++){                      // add `size` number of stars to variable row

              row += “*”;

       }

       return row;                                      // return row

}

void square(int size){                                        // creates a square

       for (int i=0; i<size; i++){

              cout << singleRow(size) << endl;                // use singleRow() function to print `size` number of stars

       }

}
```