3 Programmer-Defined Functions

- Two components of a function definition
  - Function **declaration** (or **function prototype**)
    - Shows how the function is called
    - Must appear in the code before the function can be called
    - Syntax:
      Type_returned Function_Name(Parameter_List);
      //Comment describing what function does
- Function **definition**
  - Describes how the function does its task
  - Can appear before or after the function is called
  - Syntax:
    Type_returned Function_Name(Parameter_List)
    {
    //code to make the function work
    }

3.1 Function Declaration

- Tells the return type
- Tells the name of the function
- Tells how many arguments are needed
- Tells the types of the arguments
- Tells the formal parameter names
  - Formal parameters are like placeholders for the actual arguments used when the function is called
  - Formal parameter names can be any valid identifier
- Example:

```
double total_cost(int number_par, double price_par);
// computes total cost including 5% sales tax on
// number_par items at cost of price_par each
```

3.2 Function Definition

- Provides the same information as the declaration
- Describes how the function does its task
- Example:

```
double total_cost(int number_par, double price_par) // this is header
{
        const double TAX_RATE = 0.05;
        //5% tax double subtotal;
        double subtotal = price_par * number_par;
        return (subtotal + subtotal * TAX_RATE);
}
```

3.3 The Return Statement

- Ends the function call
- Returns the value calculated by the function
- Syntax:
```
return expression;
```
  - expression performs the calculation or
  - expression is a variable containing the calculated value
- Example:
```
return subtotal + subtotal * TAX_RATE;
```

3.4 The Function Call

- Tells the name of the function to use
- Lists the arguments
- Is used in a statement where the returned value makes sense
- Example:
```
double bill = total_cost(number, price);
```

3.5 Function Call Details

- The **values** of the arguments are plugged into the formal parameters (**Call-by-value** mechanism with call-by-value parameters)
  - The first argument is used for the first formal parameter, the second argument for the second formal parameter, and so forth.
  - The value plugged into the formal parameter is used in all instances of the formal parameter in the function body

3.6 Alternate Declarations

- Two forms for function declarations

1. List formal parameter names
2. List types of formal parameters, but not names

- First aids description of the function in comments
- Examples:
```
double total_cost(int number_par, double price_par);
```
or
```
double total_cost(int, double);
```
- Function **headers** must always list formal parameter names!

3.7 Order of Arguments

- Compiler checks that the types of the arguments are correct and in the correct sequence.
- Compiler cannot check that arguments are in the correct **logical** order
- Example: Given the function declaration:

```
char grade(int received_par, int min_score_par);
```

Calling from

```
int received = 95, min_score = 60;
cout << grade( min_score, received);
```

  - Produces a faulty result because the arguments are not in the correct logical order. **The compiler will not catch this!**

3.8 Function Definition Syntax

- Within a function definition
  - Variables must be declared before they are used
  - Variables are typically declared before the executable statements begin
  - At least one return statement must end the function
    - Each branch of an if-else statement might have its own return statement

3.8.1 Placing Definitions

- A function call must be preceded by either
  - The function's declaration or
  - The function's definition
    - If the function's definition precedes the call, a declaration is not needed
- Placing the function declaration prior to the main function and the function definition after the main function leads naturally to building your own libraries in the future.