6.1 Modular Programming

- Modular programming: breaking a program up into smaller, manageable components (modules)
- Function: a collection of statements to perform a task, grouped into a single named unit.

Why is modular programming important?
- Improves maintainability/readability of programs by giving structure and organization to the code
- Simplifies the process of writing programs: programmer can write one small function at a time

6.2 Defining and Calling Functions

- Function definition: statements that make up a function, along with its name, parameters and return type.

  ```c
  return-type function-name (parameters)
  {
    statements
  }
  ```

- Function call: statement (or expression) that causes a function to execute

  ```c
  function-name (arguments)
  ```
Function Definition

A Function definition includes:

• return type: data type of the value that the function returns to the part of the program that called it.
• function-name: name of the function. Function names follow same rules as variables.
• parameters: optional list of variable definitions. These will be assigned values each time the function is called.
• body: statements that perform the function’s task, enclosed in {}.

Function Return Type

• If a function computes and returns a value, the type of the value it returns must be indicated as the return type:

```cpp
int getRate()
{
    ...
}
```

• If a function does not return a value, its return type is void:

```cpp
void printHeading()
{
    cout << "Monthly Sales\n";
}
```

Calling a Function

• To execute the statements in a function, you must “call” it from within another function (like main).
• To call a function, use the function name followed by a list of expressions (arguments) in parens:

```cpp
printHeading();
```
• Whenever called, the program executes the body of the called function (it runs the statements).
• After the function terminates, execution resumes in the calling function after the function call.
Functions in a program

- Example:

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

void displayMessage()
{
    cout << "Hello from the function displayMessage.\n";
}

int main()
{
    cout << "Hello from Main.\n";
    displayMessage();
    cout << "Back in function Main again.\n";
    return 0;
}
```

Output:

- Flow of Control (order of statements):

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```

Calling Functions: rules

- A program is a collection of **functions**, one of which must be called “main”.
- Function definitions can contain **calls** to other functions.
- A function must be defined before it can be called
  - In the program text, the function definition must occur before all calls to the function
  - Unless you use a “prototype”

6.3 Function Prototypes

- Compiler must know the following about a function before it can process a function call:
  - name, return type and data type (and order) of each parameter
- Not necessary to have the **body** of the function before the call.
- Sufficient to put just the function header before all functions containing calls to that function
  - The complete function definition must occur later in the program.
  - The header alone is called a function prototype
#include <iostream>
using namespace std;

// function prototypes
void first();
void second();

int main() {
    cout << "I am starting in function main.\n";
    first(); // function call
    second(); // function call
    cout << "Back in function main again.\n";
    return 0;
}

// function definitions
void first() {
    cout << "I am now inside the function first.\n";
}
void second() {
    cout << "I am now inside the function second.\n";
}

6.4 Sending Data into a Function

• You can pass (or send) values to a function in the function call statement.
• This allows the function to work over different values each time it is called.

• Arguments: Expressions (or values) passed to a function in the function call.
• Parameters: Variables defined in the function definition header that are assigned the values passed as arguments.

Prototype Style Notes

• Place prototypes near the top of the program (before any other function definitions)—good style.

• Using prototypes, you can place function definitions in any order in the source file.

• Common style: all function prototypes at beginning, followed by definition of main, followed by other function definitions.

A Function with a Parameter

void displayValue(int num) {
    cout << "The value is " << num << endl;
}

displayValue(5);

• num is the parameter.
• Calls to this function must have an argument (expression) that has an integer value:

• 5 is the argument.
Parameter Passing Semantics

- Given this function call, with the argument of 5:
  
  ```cpp
displayValue(5);
  ```

- Before the function body executes, the parameter (\texttt{num}) is \textbf{initialized} to the argument (5), like this:
  
  ```cpp
int num = 5; //this stmt is executed implicitly
  ```

- Then the body of the function is executed, using \texttt{num} as a regular variable:
  
  ```cpp
cout << "The value is " << num << endl;
  ```

Passing Multiple Arguments

When calling a function that has multiple parameters:

- the following must all match:
  - the number of data types in the prototype
  - the number of parameters in the function definition
  - the number of arguments in the function call

- the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

- they are assigned in order.
Example: function calls function

```
void deeper() {
    cout << "I am now in function deeper.\n";
}

void deep() {
    cout << "Hello from the function deep.\n";
    deeper();
    cout << "Back in function deep.\n";
}

int main() {
    cout << "Hello from Main.\n";
    deep();
    cout << "Back in function deep.\n";
    return 0;
}
```

Output: Hello from Main.
Hello from the function deep.
I am now in function deeper.
Back in function deep.
Back in function Main again.

Q1.2

Example: call function more than once

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

void pluses(int count) {
    for (int i = 0; i < count; i++)
        cout << "+";
    cout << endl;
}

int main() {
    int x = 2;
    pluses(4);
    pluses(x);
    pluses(x+5);
    pluses(pow(x,3.0));
    return 0;
}
```

Output:
```
++++
++
+++++++ 
```

Q2.2

Example: multiple parameters

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

void pluses(char ch, int count) {
    for (int i=0; i < count; i++)
        cout << ch;
    cout << endl;
}

int main() {
    int x = 2;
    char cc = '!';
    pluses('#',4);
    pluses('*',x);
    pluses(cc,x+5);
    pluses('x',pow(x,3.0));
    return 0;
}
```

Output:
```
####
**
!!!!!!!
xxxxxxxxx 
```

Q3.4

Testing

- **Testing**: running the program with simulated data, checking the actual output against expected output, in order to find bugs
- **Bug**: coding mistake causing an error
- **Test Case**: a set of specific input data and the corresponding expected program output
- **Choose input data wisely**:
  - Values used in if/while conditions
  - Smallest and largest valid values of a dataset
  - Put data in multiple positions: for maximum, put max value in first position, then last position, then middle position
Sample Test Cases (for PA4)

- **Input:** A 400 **Output:** $39.99 (no savings)
- **Input:** A 480 **Output:** $53.49 (no savings)
tests computation of overage minutes
- **Input:** B 900 **Output:** $59.99 (no savings)
tests value in if condition: if (minutes>900)
- **Input:** C 1000 **Output:** $69.99 (no savings)
tests package C
- **Input:** A 500 **Output:** $62.49 Savings on B: $2.50
tests savings on B but not C
- **Input:** A 905 **Output:** $244.74
  Savings on B: $182.75, Savings on C: 174.75
tests savings on B and C

Debugging

- **Test failure:** actual output from running a test case does not match the expected output.
- **Debugging:** figure out why it failed, find the coding mistake and fix it.
- Try hand tracing the code (or function).
- Add output statements in strategic places
  - Using cout, output values of variables (use labels and endl) before and after variables are set, beginning and end of functions, before and after function calls.
  - trace execution path, see which statements are being reached. Add cout<<“here1”<<endl; statements after every three or four statements in your program.