Modular Programming

- **Modular programming**: breaking a program up into smaller, manageable functions or modules
- **Function**: a collection of statements to perform a task
- **Motivation for modular programming**:
  - Improves maintainability of programs
  - Simplifies the process of writing programs
Defining and Calling Functions

- Function call: statement that causes a function to execute
- Function definition: statements that make up a function
Function Definition

- **Definition includes:**
  - *return type*: data type of the value that the function returns to the part of the program that called it
  - *name*: name of the function. Function names follow same rules as variables
  - *parameter list*: variables containing values passed to the function
  - *body*: statements that perform the function’s task, enclosed in `{ }`

```
return-type function-name (parameter declarations, if any)
{
    statements
}
```
Function Return Type

- If a function returns a value, the type of the value must be indicated:

  ```cpp
  int main()
  ```

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

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

Calling a Function

- To call a function, use the function name followed by `()`

  ```cpp
  printHeading();
  ```

- When called, the program executes the body of the called function

- 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;
}
```

Functions in a program

• Output
  Hello from Main.
  Hello from the function displayMessage.
  Back in function Main again.

• Flow of Control:
Calling Functions

• main can call any number of functions
• Functions can call other functions
• Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

Function Prototypes

• Two ways to notify the compiler about a function before it encounters a call to the function:
  - Place the function definition before all calls to that function.
  - Place a function prototype (function declaration) before all calls to that function
    * Prototype looks like the function header
    * Example: `void printHeading();`
Prototypes in a program

• Example:

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

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

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

• Example, cont.:

```cpp
void first()
{
    cout << "I am now inside the function first.\n";
}

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

Output:
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
Prototype Notes

• Place prototypes near the top of the program (before any other function definitions)

• Program must include either a prototype or full function definition before any call to the function – otherwise: compiler error

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

Sending Data into a Function

• You can pass values to a function through the function call:
  
  \[ c = \text{pow}(a, 2); \]

• Expressions (or values) passed to a function are called arguments

• Variables in a function that accept the values passed as arguments are parameters
A Function with a Parameter

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

• num is the parameter. It accepts int arguments.

• Calls to this function must have an argument of type int.

    displayValue(5);

Function with parameter in program

• Example:

    #include <iostream>
    using namespace std;

    // Function Prototype
    void displayValue(int);

    int main() {
        cout << "I am passing 5 to displayValue.\n";
        displayValue(5);
        cout << "Back in function main again.\n";
        return 0;
    }

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

    Output:
    I am passing 5 to displayValue.
The value is 5
    Back in function main again.
Parameter Passing behavior

- The function call, with the argument:
  
  ```
  displayValue(5);
  ```

- Before the function executes, the parameter is initialized to the argument expression:
  
  ```
  int num = 5;
  ```

- Then the body of the function is executed, using num as a variable:
  
  ```
  cout << "The value is " << num << endl;
  ```

Parameters, Prototypes, and Function Headers

- The prototype must include the data type of each parameter inside its parentheses
  
  ```
  void evenOrOdd(int);  //prototype
  ```

- The header must include a declaration for each parameter in its () (data type + param name)
  
  ```
  void evenOrOdd(int num) //header
  ```

- The call must include an expression for each parameter, inside its parentheses.
  
  ```
  evenOrOdd(val);    //call
  ```
Passing Multiple Arguments

- When calling a function and passing multiple arguments:
  - the number of arguments in the call must match the prototype and definition
  - the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

Function Call Notes

- The value of the argument expression is copied into the parameter (using initialization) when the function is called
- A function can have multiple parameters
- There must be a data type listed in the prototype and a parameter declaration in the function header for each parameter
- Arguments will be type converted as necessary to match parameters
- A parameter’s scope is the function which uses it