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

- Function call: statement (or expression) that causes a function to execute
  
  `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 from each function call.
- **body**: statements that perform the function’s task, enclosed in {}

Function Header

\[
\text{return-type function-name (parameters)} \\
\{ \\
\text{statements} \\
\}
\]

Function Return Type

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

  ```
  int getRate() 
  { 
  …
  }
  ```

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

  ```
  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):

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

Flow of Control:
- Control always starts at `main`

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 sees a function call:
  - name, return type, number of parameters, and data type 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
  - Then include the complete definition later in the program.
  - The header alone is called a function prototype
#include <iostream>
using namespace std;

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";
}

Prototype Style Notes

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

6.4 Sending Data into a Function

- You can pass values to a function in the function call.
- 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.

A Function with a Parameter

```cpp
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:
Function with parameter in program

```cpp
#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;
}

// Function definition
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.

Parameters in Prototypes and Function Definitions

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

- The **definition** must include a *declaration* for each parameter in its parens
  
  ```
  void evenOrOdd(int num)   //header
  { if (num%2==0) cout << "even";
    else cout << "odd"; }
  ```

- The **call** must include an *argument* (expression) for each parameter, inside its parentheses
  
  ```
  evenOrOdd(x+10);   //call
  ```

Parameter Passing Semantics

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

- Before the function body executes, the parameter (num) is **initialized** to the argument (5), like this:
  ```
  int num = 5;
  ```

- Then the body of the function is executed, using num as a regular variable:
  ```
  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 arguments in the call
  - the number of data types in the prototype
  - the number of parameters in the definition

- 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

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

test 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 Main again.\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.

---

Example: call function more than once

```cpp
#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: ~++++~
**~++~
~+++++++~

---

Example: multiple parameters

```cpp
#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: ####
**
!!!!!!
xxxxxxx