6.1 Modular Programming

- Modular programming: breaking a program up into smaller, manageable components (modules)
- Function: a collection of statements that 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()
  {
    return 8;
  }
  ```

- 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: Hello from main.
  Hello from the function displayMessage.
  Back in function main again.

• Flow of Control (order of statements):

```
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;
}
```

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

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

- num is the parameter.
- Calls to this function must provide an argument (expression) that has an integer value:
  ```cpp
displayValue(5);
  ```
- 5 is the **argument**.
# 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";
    displayValue(8);  //call again with diff. argument
    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.
- The value is 8

---

## Parameter Passing Semantics

- **Given this function call, with the argument of 5:**
  ```cpp
displayValue(5);
  ```
- **Before the function body executes, the parameter (num) is 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 num as a regular variable:**
  ```cpp
cout << "The value is " << num << endl;
  ```

---

## Parameters in Prototypes and Function Definitions

- **The prototype** must include the **data type** of each parameter inside its parentheses:
  ```cpp
  void evenOrOdd(int);   //prototype
  ```
- **The definition** must include a **definition** for each parameter in its parens
  ```cpp
  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
  ```cpp
  evenOrOdd(x+10);   //call
  ```

---

## Passing Multiple Arguments

When calling a function that has multiple parameters:

```cpp
void power(int, int);   //prototype
```

- **the following must all match:**
  - the number and order of data types in the prototype
  - the number and order of parameters in the function definition
  - the number and order 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

```cpp
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.

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() {
    cout << "Hello from Main.\n";
    deep();
    cout << "Back in function deep.\n";
    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');
    pluses(x,5);
    pluses('x',pow(x,3.0));
    return 0;
}
```

Output: ####
**
!!!!!
xxxxxxx

6.7 The `return` statement

- Used to stop the execution of a void function
- Can be placed anywhere in the function body
  - the function immediately transfers control back to the statement that called it.
- Statements that follow the return statement will not be executed
- In a void function with no return statement, the compiler adds a return statement before the last }

```cpp
return;
```
The return statement: example

```cpp
void someFunc (int x) {
    if (x < 0) {
        cout << "x must not be negative." << endl;
    } else {
        // Continue with lots of statements, indented
        // ...
        // so many it’s hard to keep track of matching {}
    }
}
```

```cpp
void someFunc (int x) {
    if (x < 0) {
        cout << "x must not be negative." << endl;
        return;
    }
    // Continue with lots of statements, less indentation,
    // no brackets to try to match ...
}
```

This is equivalent, easier to read

6.8 Returning a value from a function

• You can use the return statement in a non-void function to send a value back to the function call:
  ```cpp
  return expr;
  ```

• The value of the expr will be sent back.

• The data type of expr must be placed in the function header:
  ```cpp
  int doubleIt(int x) {
      return x*2;
  }
  ```

Calling a function that returns a value

• If the function returns void, the function call is a statement:
  ```cpp
  pluses(4);
  ```

• If the function returns a value, the function call is an expression:
  ```cpp
  int y = doubleIt(4);
  ```

• The value of the function call (underlined) is the value of the expr returned from the function, and you should do something with it.

Returning the sum of two ints

```cpp
#include <iostream>
using namespace std;
int sum(int,int);
int main() {
    int value1;
    int value2;
    int total;
    cout << "Enter 2 numbers: " << endl;
    cin >> value1 >> value2;
    total = sum(value1, value2);
    cout << "The sum is " << total << endl;
}
```

```
int sum(int x, int y) {
    return x + y;
}
```

Enter 2 numbers: 20  40
The sum is 60
Data transfer

- The function call from main: `sum(value1, value2)` passes the values stored in `value1` and `value2` (20 and 40) to the function, assigning them to `x` and `y`.
- The result, `x+y` (60), is returned to the call and stored in `total`.

Function call expression

- When a function call calls a function that returns a value, it is an expression.
- The function call can occur in any context where an expression is allowed:
  - assign to variable (or array element)  `total = sum(x,y);`
  - output via cout  `cout << sum(x,y);`
  - use in a more complicated expression  `cout << sum(x,y)*.1;`
  - pass as an argument to another function  `z = pow(sum(x,y),2);`
- The value of the function call is determined by the value of the expression returned from the function.

6.9 Returning a boolean value

```cpp
bool isValid(int number) {
    bool status;
    if (number >=1 && number <= 100)
        status = true;
    else
        status = false;
    return status;
}
```

- the above function is equivalent to this one:

```cpp
bool isValid (int number) {
    return (number >=1 && number <= 100);
}
```

Returning a boolean value

- You can call the function in an if or while:

```cpp
int main() {

    int val;
    cout << "Enter a value between 1 and 100: ";
    cin >> val;
    while (!isValid(val)) {
        cout << "That value was not in range.\n";
        cout << "Enter a value between 1 and 100: ";
        cin >> val;
    }
    // . . .
}
```
6.5 Passing Data by Value

(review)

- **Pass by value**: when an argument is passed to a function, its value is copied into the parameter.
- Parameter passing is implemented using variable initialization (behind the scenes):
  
  ```cpp
  int param = argument;
  ```
- Changes to the parameter in the function definition cannot affect the value of the argument in the call

---

Pass by Value notes

When the argument is a variable (as in f(x)):

- The parameter is initialized to a *copy* of the argument's value.
- Even if the body of the function changes the parameter, the argument in the function call is unchanged.
- The parameter and the argument are stored in separate variables, separate locations in memory.

---

Example: Pass by Value

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

void changeMe(int);

int main() {
  int number = 12;
  cout << "number is " << number << endl;
  changeMe(number);
  cout << "Back in main, number is " << number << endl;
  return 0;
}

void changeMe(int myValue) {
  myValue = 200;
  cout << "myValue is " << myValue << endl;
}
```

Output:
- number is 12
- myValue is 200
- Back in main, number is 12

---

6.13 Passing Data by Reference

- **Pass by reference**: when an argument is passed to a function, the function has direct access to the original argument.
- Pass by reference in C++ is implemented using a reference parameter, which has an ampersand (&) in front of it:
  ```cpp
  void changeMe (int &myValue);
  ```
- A reference parameter acts as an *alias* to its argument.
- Changes to the parameter in the function **DO** affect the value of the argument
# Example: Pass by Reference

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

void changeMe(int &);

int main() {
    int number = 12;
    cout << "number is " << number << endl;
    changeMe(number);
    cout << "Back in main, number is " << number << endl;
    return 0;
}

void changeMe(int &myValue) {
    myValue = 200;
    cout << "myValue is " << myValue << endl;
}
```

Output: number is 12 myValue is 200 Back in main, number is 200

## Using Pass by Reference for input

```cpp
double square(double number) {
    return number * number;
}

void getRadius(double &rad) {
    cout << "Enter the radius of the circle: ";
    cin >> rad;
}

int main() {
    const double PI = 3.14159;
    double radius;
    double area;
    cout << fixed << setprecision(2);
    getRadius(radius);
    area = PI * square(radius);
    cout << "The area is " << area << endl;
    return 0;
}
```

Output: myValue is an alias for number, only one shared variable this statement changes number During the function execution, rad is an alias to radius in the main program.

## 6.10 Local and Global Variables

- Variables defined inside a function are local to that function.
  - They are hidden from the statements in other functions, which cannot access them.
  - Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.
    - This is not bad style. These are easy to keep straight

- Parameters are also local to the function in which they are defined.

## Pass by Reference notes

- Changes made to a reference parameter are actually made to its argument
- The & must be in the function header AND the function prototype.
- The argument passed to a reference parameter must be a variable – it cannot be a constant or contain an operator (like +)
- Use when appropriate – don’t use when:
  - the argument should not be changed by function (!)
  - the function returns only 1 value: use return stmt!
Local variables are hidden from other functions

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

void anotherFunction();

int main() {
    int num = 1;
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}

void anotherFunction() {
    int num = 20;
    cout << "In anotherFunction, num is " << num << endl;
}
```

Output:

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

Global Variables

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program starting from the variable definition to the end of the file.
- This means that a global variable can be accessed by all functions that are defined after the global variable is defined.
- A local variable may have the same name as a global variable. The global variable is hidden in that variable’s block.

Local Variable Lifetime

- A function’s local variables and parameters exist only while the function is executing.
- When the function begins, its parameters and local variables (as their definitions are encountered) are created in memory, and when the function ends, the parameters and local variables are destroyed.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables: example

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

void anotherFunction();

int main() {
    int num = 2;
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}

void anotherFunction() {
    int num = 50;
    cout << "In anotherFunction, num is " << num << endl;
    num = 50;
    cout << "But now it is changed to " << num << endl;
}
```

Output:

```
In main, num is 2
In anotherFunction, num is 50
But now it is changed to 50
Back in main, num is 50
```
Global Variables/Constants

Do not use global variables!!! Because:

- They make programs difficult to debug.
  - If the wrong value is stored in a global var, you must scan the entire program to see where the variable is changed.
- Functions that access globals are not self-contained
  - cannot easily reuse the function in another program.
  - cannot understand the function without understanding how the global is used everywhere.

It is ok (and good) to use global constants because their values do not change.

Global Constants: example

```cpp
const double PI = 3.14159;

double getArea(double number) {
    return PI * number * number;
}

double getPerimeter(double number) {
    return PI * 2 * number;
}

int main() {
    double radius;
    cout << fixed << setprecision(2);
    cout << "Enter the radius of the circle: ";
    cin >> radius;
    cout << "The area is " << getArea(radius) << "\n";
    cout << "The perimeter is " << getPerimeter(radius) << "\n";
}
```

Output:

```
Enter the radius of the circle: 2.2
The area is 15.21
The perimeter is 13.82
```

Functions and Array Elements

- An array element can be passed to any parameter of the same (or compatible) type:

```cpp
double square (double);

int main() {
    double numbers[5] = {2.2, 3.3, 5.11, 7.0, 3.2};
    for (int i=0; i<5; i++)
        cout << square(numbers[i]) << " ";
    cout << "\n";
    return 0;
}

double square (double x) {
    return x * x;
}
```

Output:

```
4.84 10.89 26.1121 49 10.24
```

- An array element can be passed by reference.

What is output by this program?

```cpp
void changeMe(int &myValue) {
    myValue = 200;
}

int main() {
    int numbers[5] = {2, 3, 5, 7, 3};
    for (int i=0; i<5; i++)
        changeMe(numbers[i]);
    for (int i=0; i<5; i++)
        cout << numbers[i] << " ";
    cout << "\n";
}
```

Output:

```
4.84 10.89 26.1121 49 10.24
```

functions and array elements
7.8 Arrays as Function Arguments

- An entire array can(!) be passed to a function that has an array parameter.

```cpp
void showArray(int[], int);

int main() {
    int numbers[5] = {2, 3, 5, 7, 3};
    showArray(numbers, 5);
    return 0;
}
```

```cpp
void showArray(int values[], int size) {
    for (int i = 0; i < size; i++)
        cout << values[i] << " ";
    cout << endl;
}
```

Output: 

2 3 5 7 3

---

Passing arrays to functions

- An array is always passed by reference.

- The parameter name is an alias to the array being passed in, even though it has no &.

- Changes made to the array (elements) inside the function DO affect the array in the function call.

```cpp
void incrArray(int[], int);
void showArray(int[], int);

int main() {
    int numbers[5] = {2, 3, 5, 7, 3};
    incrArray(numbers, 5);
    showArray(numbers, 5);
    return 0;
}
```

```cpp
void incrArray(int values[], int size) {
    for (int i = 0; i < size; i++)
        (values[i])++;
    //values[i]=values[i]+1;
}
```

Output:

3 4 6 8 4

---

Passing arrays to functions

- In the function definition, the parameter type is a variable name with an empty set of brackets: [ ]
  - Do NOT give a size for the parameter

```cpp
void showArray(int values[], int size) {...}
```

- In the prototype, empty brackets go after the element datatype.

```cpp
void showArray(int[], int);
```

- In the function call, use the variable name for the array (no brackets!).

```cpp
showArray(numbers, 5);```

---

Passing arrays to functions

- Changing an array inside a function:
Passing arrays to functions

• Usually functions that have an array parameter also have an int parameter for the count of the number of elements in the array.
  ‣ so the function knows how many elements to process.

• The count parameter is just a regular int parameter and must be included in the parameter list and a corresponding argument value must appear in the function call.