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
MODULAR PROGRAMMING

This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
    statement;
    statement;
    statement;
}
```

- **Function call:**
  - Statement causes a function to execute
- **Function definition:**
  - Statements that make up a function

Definition includes:
- **Return type:**
  - Data type of the value that 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 {}
FUNCTION
RETURN TYPE

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

  ```
  int main()
  ```

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

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

FUNCTION
CALLING

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

  ```
  printHeading();
  ```

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

- After the function terminates, execution resumes in the calling function at point of call.
FUNCTION
IN PROGRAM

Program 6-1

```c
1 // This program has two functions: main and displayMessage
2 #include <iostream>
3 using namespace std;
4
5 //*********************************************************
6 // Definition of function displayMessage *
7 // This function displays a greeting. *
8 //*********************************************************
9
10 void displayMessage()
11 {
12    cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*********************************************************
16 // Function main *
17 //*********************************************************
18
19 int main()
20 {
21    cout << "Hello from main.\n";
22    displayMessage();
23    cout << "Back in function main again.\n";
24    return 0;
25 }
```

Program Output
Hello from main.
Hello from the function displayMessage.
Back in function main again.

FUNCTION
FLOW OF CONTROL

```c
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;
}
```
FUNCTION CALLING

- `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

- Ways to notify the compiler about a function before a call to the function:
  - Place function definition before calling function's definition
  - Use a function prototype (function declaration) – like the function definition without the body
    - Header: `void printHeading();`
    - Prototype: `void printHeading();`
FUNCTION PROTOTYPES

IN PROGRAM

Program 6.5

1 // This program has three functions: main, First, and Second.
2 #include <iostream>
3 using namespace std;
4
5 // Function Prototypes
6 void first();
7 void second();
8
9 int main()
10 {
11   cout << "I am starting in function main.\n";
12   first(); // Call function first
13   second(); // Call function second
14   cout << "Back in function main again.\n";
15   return 0;
16 }
17

FUNCTION PROTOTYPES

IN PROGRAM – CONTINUED

18 //*****************************************************************************
19 // Definition of function first.  *
20 // This function displays a message.  *
21 //*****************************************************************************
22 void first()
23 {
24   cout << "I am now inside the function first.\n";
25 }
26
27 //*****************************************************************************
28 // Definition of function second.  *
29 // This function displays a message.  *
30 //*****************************************************************************
31 void second()
32 {
33   cout << "I am now inside the function second.\n";
34 }
FUNCTION PROTOTYPES

NOTES

• Place prototypes near top of program

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

• When using prototypes, can place function definitions in any order in source file

SENDING DATA INTO A FUNCTION

• Can pass values into a function at time of call:
  
  ```
  c = pow(a, b);
  ```

• Values passed to function are arguments

• Variables in a function that hold the values passed as arguments are parameters
A FUNCTION WITH A PARAMETER VARIABLE

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

- The integer variable `num` is a parameter.
- It accepts any integer value passed to the function.

A FUNCTION WITH A PARAMETER VARIABLE IN PROGRAM

```c++
Program 6-6
```

```c++
1 // This program demonstrates a function with a parameter.
2 #include <iostream>
3 using namespace std;
4
5 // Function Prototype
6 void displayValue(int);
7
8 int main()
9 {
10    cout << "I am passing 5 to displayValue.\n";
11    displayValue(5); // Call displayValue with argument 5
12    cout << "Now I am back in main.\n";
13    return 0;
14 }
15
A FUNCTION WITH A PARAMETER VARIABLE

Program 6.6 (continued)

16 // Definition of function displayValue. *
17 // It uses an integer parameter whose value is displayed. *
18 //*******************************************************************************
20 21 void displayValue(int num)
22  {  
23   cout << "The value is " << num << endl;
24  }

Program Output

I am passing 5 to displayValue.
The value is 5
Now I am back in main.

A FUNCTION WITH A PARAMETER VARIABLE

FLOW

- The function call in line 11 passes the value 5 as an argument to the function.

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

- A parameter can also be called a **formal parameter** or a **formal argument**
- An argument can also be called an **actual parameter** or an **actual argument**

PARAMETERS, PROTOTYPES, AND FUNCTION HEADERS

- For each function argument,
  - the prototype must include the data type of each parameter inside its parentheses
  - the header must include a declaration for each parameter in its ()

```c
void evenOrOdd(int); //prototype
evenOrOdd(val);     //call
```
FUNCTION CALL NOTES

- Value of argument is copied into parameter when the function is called

- A parameter’s scope is the function which uses it

- Function can have multiple parameters

- There must be a data type listed in the prototype and an argument declaration in the function header for each parameter

- Arguments will be promoted/demoted as necessary to match parameters

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.
PASSING MULTIPLE ARGUMENTS

Program 6-8

```cpp
1 // This program demonstrates a function with three parameters.
2 #include <iostream>
3 using namespace std;
4
5 // Function Prototype
6 void showSum(int, int, int);
7
8 int main()
9 {
10    int value1, value2, value3;
11    // Get three integers.
12    cout << "Enter three integers and I will display ";
13    cout << "their sum: ";
14    cin >> value1 >> value2 >> value3;
15    // Call showSum passing three arguments.
16    showSum(value1, value2, value3);
17    return 0;
18 }
```

PASSING MULTIPLE ARGUMENTS

Program Output with Example Input Shown in Bold

Enter three integers and I will display their sum: 4 8 7 [Enter]

Lecture 06 - Functions
PASSING MULTIPLE ARGUMENTS

- The function call in line 18 passes value1, value2, and value3 as arguments to the function.

```cpp
void showSum(int num1, int num2, int num3)
{
    cout << (num1 + num2 + num3) << endl;
}
```

PASSING DATA BY VALUE

- **Pass by value:**
  - when an argument is passed to a function, its value is copied into the parameter.
  - Changes to the parameter in the function do not affect the value of the argument

- **Example:**
  ```cpp
  int val=5;
  evenOrOdd(val);
  ```

- **evenOrOdd** can change variable `num`, but it will have no effect on variable `val`
THE RETURN STATEMENT

- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the return statement will not be executed
- Can be used to prevent abnormal termination of program
- In a void function without a return statement, the function ends at its last }

Program 6-11

```c++
// This program uses a function to perform division. If division
// by zero is detected, the function returns.
#include <iostream>
using namespace std;

// Function prototype.
void divide(double, double);

int main()
{
  double num1, num2;
  cout << "Enter two numbers and I will divide the first\n";
  cout << "number by the second number: ";
  cin >> num1 >> num2;
  divide(num1, num2);
  return 0;
}
```
THE RETURN STATEMENT

RETURNING A VALUE FROM A FUNCTION

- A function can return a value back to the statement that called the function.
- You’ve already seen the `pow` function, which returns a value:
  ```c
  double x;
  x = pow(2.0, 10.0);
  ```

- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:
  ```c
  int sum(int num1, int num2)
  {
    double result;
    result = num1 + num2;
    return result;
  }
A VALUE-RETURNING FUNCTION

Return Type

```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

Functions can return the values of expressions, such as `num1 + num2`
A VALUE-RETURNING FUNCTION

Program 6-12

```cpp
1 // This program uses a function that returns a value.
2 #include <iostream>
3 using namespace std;
4
5 // Function prototype
6 int sum(int, int);
7
8 int main()
9 {
10    int value1 = 20, // The first value
11        value2 = 40, // The second value
12        total;   // To hold the total
13
14    // Call the sum function, passing the contents of
15    // value1 and value2 as arguments. Assign the return
16    // value to the total variable.
17    total = sum(value1, value2);
18
19    // Display the sum of the values.
20    cout << "The sum of " << value1 << " and "
21    << value2 << " is " << total << endl;
22    return 0;
23 }
```

A VALUE-RETURNING FUNCTION

```cpp
26 //*******************************************************************************
27 // Definition of function sum. This function returns *
28 // the sum of its two parameters. *
29 //*******************************************************************************
30 int sum(int num1, int num2)
31 {
32    return num1 + num2;
33 }
```

Program Output

The sum of 20 and 40 is 60
A VALUE-RETURNING FUNCTION

• The statement in line 17 calls the sum function, passing `value1` and `value2` as arguments.
• The return value is assigned to the `total` variable.

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

A VALUE-RETURNING FUNCTION

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

area = PI * square(radius);
A VALUE-RETURNING FUNCTION

• The prototype and the definition must indicate the data type of return value (not void)

• Calling function should use return value:
  • assign it to a variable
  • send it to cout
  • use it in an expression

RETURNING A BOOLEAN VALUE

• Function can return true or false
• Declare return type in function prototype and heading as bool
• Function body must contain return statement(s) that return true or false
• Calling function can use return value in a relational expression
RETURNING A BOOLEAN VALUE

Program 6-15

```cpp
// This program uses a function that returns true or false.
#include <iostream>
using namespace std;

// Function prototype
bool isEven(int);

int main()
{
    int val;

    // Get a number from the user.
    cout << "Enter an integer and I will tell you ";
    cout << "if it is even or odd ";
    cin >> val;

    // Indicate whether it is even or odd.
    if (isEven(val))
        cout << val << " is even.";
    else
        cout << val << " is odd.";
    return 0;
}
```

RETURNING A BOOLEAN VALUE

```cpp
// Definition of function isEven. This function accepts an
// Integer argument and tests it to be even or odd. The function
// returns true if the argument is even or false if the argument
// is odd. The return value is a bool.

bool isEven(int number)
{
    bool status;
    if (number % 2 == 0)
        status = true; // The number is even if there is no remainder.
    else
        status = false; // Otherwise, the number is odd.
    return status;
}
```

Program Output with Example Input Shown in Bold
Enter an integer and I will tell you if it is even or odd: 5 [Enter]
5 is odd.
LOCAL AND GLOBAL VARIABLES

• Variables defined inside a function are *local* to that function. They are hidden from the statements in other functions, which normally cannot access them.

• Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

```cpp
// This program shows that variables defined in a function
// are hidden from other functions.
#include <iostream>
using namespace std;

void anotherFunction(); // Function prototype

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

// Definition of anotherFunction

void anotherFunction()
{
    int num = 20; // Local variable
    cout << "In anotherFunction, num is " << num << endl;
}
```
LOCAL AND GLOBAL VARIABLES

When the program is executing in main, the num variable defined in main is visible. When anotherFunction is called, however, only variables defined inside it are visible, so the num variable in main is hidden.

LOCAL VARIABLE LIFETIME

A function’s local variables exist only while the function is executing. This is known as the lifetime of a local variable.

When the function begins, its local variables and its parameter variables are created in memory, and when the function ends, the local variables and parameter 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 AND GLOBAL CONSTANTS

- 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 from the variable definition to the end.
- This means that a global variable can be accessed by all functions that are defined after the global variable is defined.
- You should avoid using global variables because they make programs difficult to debug.
- Any global that you create should be global constants.

LOCAL AND GLOBAL VARIABLES

```cpp
1 // This program calculates gross pay.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 // Global constants
7 const double PAY_RATE = 22.55;  // Hourly pay rate
8 const double BASE_HOURS = 40.0;  // Max non-overtime hours
9 const double OT_MULTIPLIER = 1.5;  // Overtime multiplier
10
11 // Function prototypes
12 double getBasePay(double);
13 double getOvertimePay(double);
14
15 int main()
16 {
17    double hours,  // Hours worked
18        basePay,  // Base pay
19        overtime = 0.0,  // Overtime pay
20        totalPay;  // Total pay
```
LOCAL AND GLOBAL VARIABLES

- The constants are then used for those values throughout the program.

```c
29    // Get overtime pay, if any.
30    if (hours > BASE_HOURS)
31        overtime = getOvertimePay(hours);
```

```c
56    // Determine base pay.
57    if (hoursWorked > BASE_HOURS)
58        basePay = BASE_HOURS * PAY_RATE;
59    else
60        basePay = hoursWorked * PAY_RATE;
```

```c
75    // Determine overtime pay.
76    if (hoursWorked > BASE_HOURS)
77        
78        overtimePay = (hoursWorked - BASE_HOURS) * 
79            PAY_RATE * OT_MULTIPLIER;
80    
```

INITIALIZING LOCAL AND GLOBAL VARIABLES

- Local variables are not automatically initialized. They must be initialized by programmer.

- Global variables (not constants) are automatically initialized to 0 (numeric) or NULL (character) when the variable is defined.
STATIC LOCAL VARIABLES

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
- static local variables retain their contents between function calls.
- static local variables are defined and initialized only the first time the function is executed. 0 is the default initialization value.

**Program 6-21**

```cpp
1    // This program shows that local variables do not retain
2    // their values between function calls.
3    #include <iostream>
4    using namespace std;
5
6    // Function prototype
7    void showLocal();
8
9    int main()
10    {
11        showLocal();
12        showLocal();
13        return 0;
14    }
15```

STATIC LOCAL VARIABLES
IN PROGRAM
STATIC LOCAL VARIABLES

IN PROGRAM

- In this program, each time showLocal is called, the localNum variable is re-created and initialized with the value 5.

```
Program 6-21  (continued)
16                         // Definition of function showLocal.
17                         // The initial value of localNum, which is 5, is displayed.
18                         // The value of localNum is then changed to 99 before the *
19                         // function returns. *
20                         // ************************************************************************
21
22 void showLocal()
23 {
24     int localNum = 5; // Local variable
25     cout << "localNum is " << localNum << endl;
26     localNum = 99;
27 }
```

Program Output
localNum is 5
localNum is 5

STATIC LOCAL VARIABLES

A DIFFERENT APPROACH

```
Program 6-22
1     // This program uses a static local variable.
2     #include <iostream>
3     using namespace std;
4     
5     void showStatic(); // Function prototype
6     
7     int main()
8     {
9         // Call the showStatic function five times.
10         for (int count = 0; count < 5; count++)
11             showStatic();
12         return 0;
13     }
14 
```
STATIC LOCAL VARIABLES

**Program 6-22**  
(continued)

```c
15 //********************************************************************************
16 // Definition of function showStatic. *
17 // statNum is a static local variable. Its value is displayed *
18 // and then incremented just before the function returns. *
19 //********************************************************************************
20 void showStatic()
21 {
22   static int statNum;
23   cout << "statNum is " << statNum << endl;
24   statNum++;
25 }
```

Program Output

- `statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

- statNum is 0
- statNum is 1
- statNum is 2
- statNum is 3
- statNum is 4

STATIC LOCAL VARIABLES

- If you do initialize a local static variable, the initialization only happens once.

```c
16 //********************************************************************************
17 // Definition of function showStatic. *
18 // statNum is a static local variable. Its value is displayed *
19 // and then incremented just before the function returns. *
20 //********************************************************************************
22 void showStatic()
23 {
24   static int statNum = 5;
25   cout << "statNum is " << statNum << endl;
26   statNum++;
27 }
```

Program Output

- statNum is 5
- statNum is 6
- statNum is 7
- statNum is 8
- statNum is 9
DEFAULT ARGUMENTS

- A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.

- Must be a constant declared in prototype:

```c
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype

- Multi-parameter functions may have default arguments for some or all of them:

```c
int getSum(int, int=0, int=0);
```

DEFAULT ARGUMENTS IN PROGRAM

Default arguments specified in the prototype

```c
Program 6-24

// This program demonstrates default function arguments.
#include <iostream>
using namespace std;

// Function prototype with default arguments
void displayStars(int = 10, int = 1);

int main()
{
    displayStars(); // Use default values for cols and rows.
    cout << endl;
    displayStars(5); // Use default value for rows.
    cout << endl;
    displayStars(7, 3); // Use 7 for cols and 3 for rows.
    return 0;
}
```
DEFAULT ARGUMENTS CONTINUED

• If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```cpp
int getSum(int, int=0, int=0); // OK
int getSum(int, int=0, int);  // NO
```

• When an argument is omitted from a function call, all arguments after it must also be omitted:

```cpp
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);  // NO
```
PASSING BY REFERENCE

- A mechanism that allows a function to work with the original argument from the function call, not a copy of the argument
- Allows the function to modify values stored in the calling environment
- Provides a way for the function to 'return' more than one value

- A reference variable is an alias for another variable
- Defined with an ampersand (&)
  ```c
  void getDimensions(int&, int&);
  ```
- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters by reference

PASSING BY REFERENCE
IN PROGRAM

```
// This program uses a reference variable as a function parameter.
#include <iostream>
using namespace std;

// Function prototype. The parameter is a reference variable.
void doubleNum(int &);

int main()
{
    int value = 4;

    cout << "In main. value is " << value << endl;  
    cout << "Now calling doubleNum..." << endl;
    doubleNum(value);
    cout << "Now back in main. value is " << value << endl;
    return 0;
}
```

The & here in the prototype indicates that the parameter is a reference variable.

Here we are passing value by reference.
PASSING BY REFERENCE
CONTINUED

The & also appears here in the function header.

```cpp
20 //******************************************************************************************
21 // Definition of doubleNum.
22 // The parameter refVar is a reference variable. The value
23 // in refVar is doubled.
24 //******************************************************************************************
25
26 void doubleNum (int &refVar)
27 {
28   refVar *= 2;
29 }
```

Program Output
In main, value is 4
Now calling doubleNum...
Now back in main, value is 8

PASSING BY REFERENCE

- Each reference parameter must contain &
- Space between type and & is unimportant
- Must use & in both prototype and header
- Argument passed to reference parameter must be a variable
  - cannot be an expression or constant
- Use when appropriate
  - don't use when argument should not be changed by function,
  - or if function needs to return only 1 value
OVERLOADING FUNCTIONS

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take different parameter types or different number of parameters
- Compiler will determine which version of function to call by argument and parameter lists

FUNCTION OVERLOADING

Using these overloaded functions,

```c
void getDimensions(int);       // 1
void getDimensions(int, int);  // 2
void getDimensions(int, double); // 3
void getDimensions(double, double);// 4
```

the compiler will use them as follows:

```c
int length, width;
double base, height;
getDimensions(length);       // 1
getDimensions(length, width); // 2
getDimensions(length, height); // 3
getDimensions(height, base); // 4
```
OVERLOADING FUNCTIONS

IN PROGRAM

Program 6-27

```c++
1 // This program uses overloaded functions.
2 #include <iostream>
3 #include <iomanip>
4 using namespace std;
5
6 // Function prototypes
7 int square(int);
8 double square(double);
9
10 int main()
11 {
12   int userInt;
13   double userFloat;
14
15   // Get an int and a double.
16   cin >> userInt >> userFloat;
17   cout << fixed << showpoint << setprecision(2);
18   cout << "Enter an integer and a floating-point value: ";
19   cout >> userInt >> userFloat;
20
21   // Display their squares.
22   cout << "Here are their squares: ";
23   cout << square(userInt) << " and " << square(userFloat) << endl;
24   return 0;
25 }
```

The overloaded functions have different parameter lists

Passing an int

Passing a double

OVERLOADING FUNCTIONS

CONTINUED

```c++
26 //*******************************************************************************
27 // Definition of overloaded function square.
28 // This function uses an int parameter, number. It returns the int square of number.
29 //*******************************************************************************
30 int square(int number)
31 {
32   return number * number;
33 }
34
35 //*******************************************************************************
36 // Definition of overloaded function square.
37 // This function uses a double parameter, number. It returns the square of number as a double.
38 //*******************************************************************************
39 double square(double number)
40 {
41   return number * number;
42 }
```

Program Output with Example Input Shown in Bold

Enter an integer and a floating-point value: 12 4.2 [Enter]

Here are their squares: 144 and 17.64
THE `EXIT()` FUNCTION

- Terminates the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for abnormal termination of program
- Requires `cstdlib` header file

Example:
```c
exit(0);
```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:
  ```c
  exit(EXIT_SUCCESS);
  exit(EXIT_FAILURE);
  ```

STUBS AND DRIVERS

- Useful for testing and debugging program and function logic and design

  - Stub:
    - A dummy function used in place of an actual function
    - Usually displays a message indicating it was called. May also display parameters

  - Driver:
    - A function that tests another function by calling it
    - Various arguments are passed and return values are tested