Chapter 6: Functions

6.1 Modular Programming

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

6.2 Defining and Calling Functions

Defining and Calling Functions

- Function call: statement 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 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";
  }
  ```

Calling a Function

- 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.

Functions in Program 6-1

Flow of Control in Program 6-1
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

- 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 6-5

```cpp
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 int main()
9 {
10   cout << "I am starting in function main\n\n";
11   first(); // Call function first
12   second(); // Call function second
13   cout << "Back in function main again.\n\n";
14   return 0;
15 }
16
17 (Program Continues)
```

Function Prototypes in Program 6-5

```cpp
18 // ********************************************************************************
20 // This function displays a message. *
21 // ********************************************************************************
22 void first()
23 {
24   cout << "I am now inside the function first\n\n";
25 }
26
27 // ********************************************************************************
29 // Definition of function second. *
30 // This function displays a message. *
31 // ********************************************************************************
32 void second()
33 {
34   cout << "I am now inside the function second\n\n";
35 }
```

Prototype 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
6.4

Sending Data into a Function

- Can pass values into a function at time of call:
  \[ c = \text{pow}(a, b); \]
- 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.

Function with a Parameter in Program 6-6

```c
int main()
{
    // This program demonstrates a function with a parameter.
    include <iostream>
    using namespace std;

    // Function Prototype
    void displayValue(int);
    int main()
    {
        cout << "I am passing 5 to displayValue\n";
        displayValue(5); // Call displayValue with argument 5
        cout << "Now I am back in main\n";
        return 0;
    }
}
```

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

Function with a Parameter in Program 6-6 (continued)

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

Program Output

```
I am passing 5 to displayValue.
The value is 5
Now I am back in main.
```
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
void evenOrOdd(int num) //header
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.

```c
int main() {
    int val1, val2, val3;
    // Get three integers:
    cin >> val1 >> val2 >> val3;
    // Call showSum passing three arguments.
    showSum(val1, val2, val3);
    return 0;
}
```

Passing Multiple Arguments in Program 6-8

- This program demonstrates a function with three parameters.
- The function is defined:

```c
int showSum(int, int, int)
```

```c
void evenOrOdd(int);
void evenOrOdd(int num);
evenOrOdd(val);
```

```c
int main() {
    int val1, val2, val3;
    // Get three integers:
    cin >> val1 >> val2 >> val3;
    // Call showSum passing three arguments.
    showSum(val1, val2, val3);
    return 0;
}
```
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`.

Using Functions in Menu-Driven Programs

- Functions can be used
  - to implement user choices from menu
  - to implement general-purpose tasks:
    - Higher-level functions can call general-purpose functions, minimizing the total number of functions and speeding program development time
- **See Program 6-10 in the book**
6.7

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 `}`

Performing Division in Program 6-11

Program 6-11

```c++
// This program uses a function to perform division. If division
// by zero is detected, the function returns.
1 include <iostream>
2 using namespace std;
3 // Function prototype.
4 void divide(double, double);
5 int main()
6 {
7 double num1, num2;
8 cout << "Enter two numbers and I will divide the first by the second:");
9 cin >> num1 >> num2;
10 divide(num1, num2);
11 return 0;
12 }
```

(Continued)

Performing Division in Program 6-11

Program 6-11

```c++
// This program uses a function to perform division. If division
// by zero is detected, the function returns.
1 include <iostream>
2 using namespace std;
3 // Function prototype.
4 void divide(double num1, double num2);
5 int main()
6 {
7 double num1, num2;
8 int count = 0;
9 if (num2 == 0)
10 count == "Sorry, I cannot divide by zero.";
11 return;
12 
13 count == "The quotient is " << num1 / num2;
14 return 0;
15 }
```

(Note: The code above is incomplete and does not perform the division.)

6.8

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);
```
Returning a Value From a Function

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

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

Functions can return the values of expressions, such as `num1 + num2`

Function Returning a Value in Program 6-12

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

The statement in line 17 calls the `sum` function, passing `value1` and `value2` as arguments. The return value is assigned to the `total` variable.
Another Example from Program 6-13

area = PI * square(radius);

```cpp
100
   10

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

Returning a Value From a 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 in Program 6-15

```cpp
1 // This program uses a function that returns true or false.
2 include <iostream>
3 using namespace std;
4 // Function prototype
5 bool isEven(int number);
6 int main() {
7   int num;
8   // Get a number from the user.
9   cout << "Enter an integer and I will tell you ";
10  cin >> num;
11  // Indentify whether it is even or odd.
12  if (num % 2 == 0) {
13    // Even.
14    // "The number is even if there is no remainder.
15    // Even numbers are divisible by 2."
16    if (num % 2 == 0) {
17      // The number is even if there is no remainder.
18      // Even numbers are divisible by 2.
19      // Example: 2, 4, 6, 8, 10 are even numbers.
20      // Odd numbers are not divisible by 2.
21      // Example: 1, 3, 5, 7, 9 are odd numbers.
22      return 0;
23    }
24  }
25 } (Program Continues)
```

Returning a Boolean Value in Program 6-15

```cpp
25 //******************************************************************************
26 // This program uses a function that returns true or false.
27 // Define function isEven. This function accepts an integer and returns a boolean.
28 // The function must return true if the argument is even or false if the argument is odd.
29 //******************************************************************************
30 int main() {
31  int number;
32  // Get a number from the user.
33  cout << "Enter an integer and I will tell you if it is even or odd: ">
34  cin >> number;
35  // Call the function and print the result.
36  if (isEven(number)) {
37    cout << "The number is even.
38  }
39  else {
40    cout << "The number is odd.
41  }
42  // Program Output: with Example Input Shown in Bold
43  // Enter an integer and I will tell you if it is even or odd: 8 [Error]
44  // The number is odd.
```

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
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 normally cannot access them. Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

Local Variables in Program 6-16

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.
Global Variables and Global Constants

- You should avoid using global variables because they make programs difficult to debug.
- Any global that you create should be global constants.

Global Constants in Program 6-19

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

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.
- \textbf{static} local variables retain their contents between function calls.
- \textbf{static} local variables are defined and initialized only the first time the function is executed. 0 is the default initialization value.
Local Variables Do Not Retain Values Between Function calls in Program 6-21

```c
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 }
```

(Program Continues)

Local Variables Do Not Retain Values Between Function calls in Program 6-21

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

A Different Approach, Using a Static Variable in Program 6-22

```c
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 }
```

(program Continues)

A Different Approach, Using a Static Variable in Program 6-22

In this program, `statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

If you do initialize a local static variable, the initialization only happens once. See Program 6-23.

```c
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 // This is the same as a pre-increment.
20 //******************************************************************************
21 void showStatic()
22 {
23    static int statNum = 5; // statNum is automatically initialized to 0
24    cout << 'statNum is ' << statNum << endl;
25    statNum++;}
```

Program Output

statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9

```

6.12 Default Arguments
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 6-24

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

// Function prototype with default arguments
void displayArray(int = 10, int = 3);

int main()
{
  displayArray(); // Use default values for cols and rows.
  cout << endl;
  displayArray(5); // Use default value for rows.
  cout << endl;
  displayArray(7, 3); // Use 7 for cols and 1 for row.
  return 0;
}
```

Default Arguments in Program 6-24 (Program Continues)

```c
*********
*********
*********
```

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:
  ```c
  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:
  ```c
  sum = getSum(num1, num2);    // OK
  sum = getSum(num1, , num3);  // NO
  ```

Using Reference Variables as Parameters

- 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
Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (&)
  ```
  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 a Variable By Reference in Program 6-25

```
1 // This program uses a reference variable as a function
2 // parameters.
3 #include <iostream>
4 using namespace std;
5
6 // Function prototype. The parameter is a reference variable.
7 void doubleNum(int &);
8
9 int main()
10 {
11   int num1 = 4;
12   int num2 = 6;
13   cout << "In main, value is " << num1 << endl;
14   cout << "Now calling doubleNum..." << endl;
15   doubleNum(num1);
16   cout << "Now back in main, value is " << num1 << endl;
17   return 0;
18 }
```

(Program Continues)

Reference Variable Notes

- 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 Examples

Using these overloaded functions,

```cpp
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:

- `int length, width;`
- `getDimensions(length);` // 1
- `getDimensions(length, width);` // 2
- `getDimensions(length, height);` // 3
- `getDimensions(height, base);` // 4

### Function Overloading in Program 6-27

#### Program 6-27

- `// This program uses overloaded functions.
- #include <iostream>
- #include <cstdlib>
- using namespace std;
- // Function prototypes
- int square(int);
- double square(double);
- // Function definitions
- int main()
- {
- int area();
- double area(double);
- // Pass an int and a double...
- cout << "Enter two numbers: ";
- cin >> num1 >> num2; // cin >> num1 >> num2 is equivalent to cin >> num1; cin >> num2;
- // Search for integer values:
- if (cin.fail())
- {
- cout << "Error: invalid value.
- area = square(int(num1)) = " << square(num1) << " or square(double(num1)) = " << square(num1);
- exit(EXIT_FAILURE);
- }
- // Pass a double
- cout << "Enter two numbers: ";
- cin >> num1 >> num2; // cin >> num1 >> num2 is equivalent to cin >> num1; cin >> num2;
- // Search for integer values:
- if (cin.fail())
- {
- cout << "Error: invalid value.
- area = square(int(num1)) = " << square(num1) << " or square(double(num1)) = " << square(num1);
- exit(EXIT_FAILURE);
- }
- // Print results.
- cout << "Area is: " << area(num1) << endl;
- cout << "Area is: " << area(num2) << endl;
- return 0;
- }
```

Program Output with Examples Input: Shown in Bold

Enter an integer and a floating-point value: 12.42 [enter]

Area is: 149.16
Area is: 149.16

#### 6.15

### 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

### The `exit()` Function

- Example:
  ```cpp
  exit(0);
  ```

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

Stubs and Drivers

6.16

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

Stubs and Drivers

Stubs and Drivers