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 {}
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

6.3 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

```
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

```cpp
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

In the program:

```cpp
Program 6-6

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 }
```

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

**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 ()
    ```cpp
    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.

Passing Multiple Arguments in Program 6-8

```
int main() {
    int a, b, c;
    cin >> a >> b >> c;
    cout << a + b + c;
    return 0;
}
```

Passing Multiple Arguments in Program 6-8

```
void showSum(int num1, int num2, int num3)
{
    cout << num1 + num2 + num3 << endl;
}
```
Passing Multiple Arguments in Program 6-8

Function Call: `showSum(value1, value2, value3)`

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

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

6.5
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

---

Passing Information to Parameters by Value

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

  ```c
  evenOrOdd(val);
  ```

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

---

6.6
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 }

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
// Program 6-12
// This program uses a function that returns a value.
// Function name:
// sum(int num1, int num2)
// Parameters:
// int num1, int num2
// Return type:
// int

int value1 = 20; // The first value
int value2 = 40; // The second value
int total = sum(value1, value2); // To hold the total

// Display the sum of the values.
cout << "The sum of " << value1 << " and " << value2 << " is " << total << endl;
```

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

```cpp
area = PI * square(radius);

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

6.9

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
Program 6-15

// This program uses a function that returns true or false.
#include <iostream>

// Function prototype
bool is_even(int);

int main() {
    int n;
    std::cout << "Enter a number: " << std::endl;
    std::cin >> n;
    std::cout << (is_even(n) ? "Even" : "Odd") << std::endl;
    return 0;
}

bool is_even(int number) {
    return number % 2 == 0;
}

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

(continued)
6.10 Local and Global Variables

**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**

Program 6-16:

```c
#include <stdio.h>

int main()
{
    int num = 1; // local variable
    while (num <= 100)
    {
        printf("%d
", num);
        num = num + 1;
    }
    return 0;
}
```

When the program is executed, 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.

```c
// Get overtime pay if any.
if (hoursWorked > BASE_HOURS) {
  overtime = getOvertimePay(hoursWorked);
}
```

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

Program 6-21

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

(Program Continues)

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

Program 6-22

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

(Program Continues)

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

```
15 // Definition of function showStatic.
16 // statNum is a static local variable. Its value is displayed
17 // each time a call is made to function showStatic.
18 // The value of statNum is reset to 0 before the function returns.
19 // Definition of function showStatic.
20 void showStatic()
21 {
22    static int statNum = 0;
23    cout << "statNum is " << statNum << endl;
24    statNum++;
25 }
```

Program Output

```
showNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

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

```
14 // Definition of function showStatic.
15 // statNum is a static local variable. Its value is displayed
16 // each time a call is made to function showStatic.
17 // The value of statNum is reset to 0 before the function returns.
18 // Definition of function showStatic.
19 void showStatic()
20 {
21    static int statNum = 5;
22    cout << "statNum is " << statNum << endl;
23    statNum++;
24 }
```

Program Output

```
showNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```

Default Arguments

6.12
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:
  ```cpp
  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:
  ```cpp
  int getSum(int, int=0, int=0);
  ```

**Default Arguments in Program 6-24**

Default arguments specified in the prototype

Program 6-24

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

// Function prototype with default arguments
void displayEven(int = 10, int = 5);

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

Default Arguments in Program 6-24

```
// Use default values for cols and rows.
// An even number increases the column.
void displayEven(int cols, int rows)
{
  // The outer loop goes to rows - 1
  for (int row = 0; row < rows; row++)
  {
    // The inner loop goes to cols - 1
    for (int col = 0; col < cols; col++)
    {
      cout << "*";
      cout << " ";
    }
    cout << "\n";
  }
}
```

**Program Output**

```
*****
*****
*****
```

Default Arguments

- 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); // 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
  ```

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

6.13

Using Reference Variables as Parameters
Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (&)
- 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

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

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

int main()
{
    int value = 5;
    cout << "In main, value is: " << value << endl;
    cout << "Now calling doubleDim..." << endl;
    doubleDim(value);
    cout << "Now back in main, value is: " << value << endl;
    return 0;
}
```

Program Output

In main, value is 5
Now calling doubleDim...

6.14 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

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,

```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:

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

Function Overloading in Program 6-27

```c
// A program using overloaded functions.
#include <stdio.h>

void getDimensions(int);
void getDimensions(int, int);
void getDimensions(int, double);
void getDimensions(double, double);

int main()
{
    double area;
    double perimeter;

    // Call the functions:
    getDimensions(10); // Call with an int parameter
    getDimensions(10, 20); // Call with two int parameters
    getDimensions(10, 5.0); // Call with one int and one double parameter
    getDimensions(5.0, 10.0); // Call with two double parameters

    // Calculate the area and perimeter
    area = 10 * 20; // Calculate area as int x int
    perimeter = 2 * (10 + 20); // Calculate perimeter as int + int

    // Output the results
    printf("Area: %.2f, Perimeter: %.2f\n", area, perimeter);

    return 0;
}
```

6.15 The exit() Function

The `exit()` function is used to terminate the execution of a program. It can be called from any function and can pass an int value to the operating system to indicate status of program termination. It is usually used for abnormal termination of a program and requires the `cstdlib` header file.

Example:

```c
exit(0); // Normal termination
exit(EXIT_SUCCESS); // Success termination
exit(EXIT_FAILURE); // Failure termination
```
6.16

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