Standard Version of Starting Out with C++, 4th Edition

Chapter 6

Functions

Topics

6.1 Modular Programming

6.2 Defining and Calling Functions

6.3 Function Prototypes

6.4 Sending Data into a Function

6.5 Passing Data by Value

6.6 Using Functions in a Menu-Driven Program

6.7 The `return` Statement

6.8 Returning a Value from a Function

Topics

6.10 Local and Global Variables

6.11 Static Local Variables

6.12 Default Arguments

6.13 Using Reference Variables as Parameters

6.14 Overloading Functions

6.15 The `exit()` Function

6.16 Stubs and Drivers

6.9 Returning a Boolean Value

6.1 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

- 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 << "\tMonthly 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.

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 Documentation

Function definition should be preceded by comments that indicate

- Purpose of the function
- How it works, what it does
- Input values that it expects, if any
- Output that it produces, if any
- Values that it returns, if any

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();`

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 = sqrt(a*a + b*b);
```

- Values passed to function are arguments
- Variables in function that hold values passed as arguments are parameters

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 Headings

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

```
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 > 1 parameter
- 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

Calling Functions with Multiple Arguments

When calling a function with 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.

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);`

- `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

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
- Without a `return` statement, the function ends at its last `}`

6.8 Returning a Value From a Function

- `return` statement can be used to return a value from function to the point of call
- Prototype and definition must indicate 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

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

Boolean `return` Example

```
bool validTest(int);        // prototype
bool validTest(int test)    // header
{
  int lScore = 0, hScore = 100;
  if (test >= lScore && test <= hScore)
     return true;
  else
     return false;
}
if (validTest(score)) {...} // call
```

6.10 Local and Global Variables

- local variable: defined within a function or block, accessible only within the function or block
- Other functions and blocks can define variables with the same name
- When a function is called, local variables in the calling function are not accessible from within the called function

Local and Global Variables

- global variable: defined outside all functions, accessible to all functions within its scope
- Easy way to share large amounts of data between functions
- Scope of a global variable: program from point of definition to the end
- Use sparingly

Initializing Local and Global Variables

- Local variables must be initialized by programmer
- Global variables are initialized to `0` (numeric) or `NULL` (character) when variable is defined

Local and Global Variable Names

- Local variables can have same names as global variables
- When a function contains a local variable that has the same name as a global variable, the global variable is unavailable from within the function. The local definition “hides” the global definition

6.11 Static Local Variables

- Local variables only exist while function is executing. When function terminates, 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 default initialization

6.12 Default Arguments

Default argument is passed automatically to a function if argument is missing on the function call

- Must be a constant declared in prototype:

```
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:

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

Default Arguments

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

```
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:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```

6.13 Using Reference Variables as Parameters

- 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’ > 1 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

Pass by Reference - Example

```
void sqareIt(int &); //prototype
void squareIt(int &num)
{
     num *= num;
}

int localVar = 5;
squareIt(localVar);  // now has 25
```

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

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

Function Overloading Examples

Using these overloaded functions,

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

the compiler will use them as follows:

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

6.15 The `exit()` Function

- Terminates 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

6.16 Stubs and Drivers

- Stub: dummy function in place of actual function
- Usually displays a message indicating it was called. May also display parameters
- Driver: function that tests a function by calling it
- Useful for testing and debugging program and function logic and design

Standard Version of Starting Out with C++, 4th Edition

Chapter 6

Functions