Multiple Assignment and Combined Assignment

- The `=` can be used to assign a value to multiple variables:
  \[ x = y = z = 5; \]
- Value of `=` is the value that is assigned
- Associates right to left:
  \[ x = (y = (z = 5)); \]

value is 5  value is 5  value is 5
Working with Characters and string Objects

- Using `cin` with the `>>` operator to input strings can cause problems:
- It passes over and ignores any leading whitespace characters (spaces, tabs, or line breaks)
- To work around this problem, you can use a C++ function named `getline`. 
Using `getline` in Program 3-19

```
// This program demonstrates using the getline function
// to read character data into a string object.
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string name;
    string city;
    cout << "Please enter your name: ";
    getline(cin, name);
    cout << "Enter the city you live in: ";
    getline(cin, city);
    cout << "Hello, " << name << endl;
    cout << "You live in " << city << endl;
    return 0;
}
```

Problem Output with Example Input Shown in Bold
Please enter your name: Kate Smith [Enter]
Enter the city you live in: Raleigh [Enter]
Hello, Kate Smith
You live in Raleigh

Working with Characters and `string` Objects

- To read a single character:
  - Use `cin`:
    ```
    char ch;
    cout << "Strike any key to continue";  
    cin >> ch;
    ```
    Problem: will skip over blanks, tabs, `<CR>`
  - Use `cin.get()`:
    ```
    cin.get(ch);
    ```
    Will read the next character entered, even whitespace
Using `cin.get()` in Program 3-21

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    char ch;
    cout << "This program has paused. Press Enter to continue.\n"; 
    cin.get(ch);
    cout << ch << "It has paused a second time. Please press Enter again.\n"; 
    ch = cin.get();
    cout << ch << "It has paused a third time. Please press Enter again.\n";
    cin.get(ch);
    cout << ch << "Thank you!\n";
    return 0;
}
```

Program Output with Example Input Shown in Bold
This program has paused. Press Enter to continue. [Enter]
The user presses enter.
It has paused a second time. Please press Enter again. [Enter]
The user presses enter.
It has paused a third time. Please press Enter again. [Enter]
The user presses enter.
Thank you!

Working with Characters and `string` Objects

- Mixing `cin >>` and `cin.get()` in the same program can cause input errors that are hard to detect.
- To skip over unneeded characters that are still in the keyboard buffer, use `cin.ignore()`:
  ```
  cin.ignore(); // skip next char
  cin.ignore(10, '\n'); // skip the next
  // 10 char. or until a '\n'
  ```
**string** Member Functions and Operators

- To find the length of a string:
  
  ```
  string state = "Texas";
  int size = state.length();
  ```

- To concatenate (join) multiple strings:
  
  ```
  greeting2 = greeting1 + name1;
  greeting1 = greeting1 + name2;
  ```

  Or using the `+=` combined assignment operator:
  
  ```
  greeting1 += name2;
  ```

---

**More Mathematical Library Functions**

- These require `cstdlib` header file
- `rand()`: returns a random number (int) between 0 and the largest int the compute holds. Yields same sequence of numbers each time program is run.
- `srand(x)`: initializes random number generator with unsigned int `x`
Comparing Characters and Strings

Comparing Characters

• Characters are compared using their ASCII values
• 'A' < 'B'
  – The ASCII value of 'A' (65) is less than the ASCII value of 'B' (66)
• '1' < '2'
  – The ASCII value of '1' (49) is less than the ASCII value of '2' (50)
• Lowercase letters have higher ASCII codes than uppercase letters, so 'a' > 'Z'
Relational Operators Compare Characters in Program 4-20

```c++
10   // Get a character from the user.
11   cout << "Enter a digit or a letter: ";
12   ch = cin.get();
13
14   // Determine what the user entered.
15   if (ch >= '0' && ch <= '9')
16     cout << "You entered a digit.\n";
17   else if (ch >= 'A' && ch <= 'Z')
18     cout << "You entered an uppercase letter.\n";
19   else if (ch >= 'a' && ch <= 'z')
20     cout << "You entered a lowercase letter.\n";
21   else
22     cout << "That is not a digit or a letter.\n";
```

Comparing **string** Objects

• Like characters, strings are compared using their ASCII values

```c++
string name1 = "Mary";
string name2 = "Mark";
name1 > name2    // true
name1 <= name2    // false
name1 != name2    // true
name1 < "Mary Jane" // true
```

The characters in each string must match before they are equal
Relational Operators Compare Strings in Program 4-21

```cpp
26   // Determine and display the correct price
27   if (partNum == "S-29A")
28       cout << "The price is $" << PRICE_A << endl;
29   else if (partNum == "S-29B")
30       cout << "The price is $" << PRICE_B << endl;
31   else
32       cout << partNum << " is not a valid part number.\n";
```

The Conditional Operator

- Can use to create short if/else statements
- Format: `expr ? expr : expr;`

```
x<0 ? y=10 : z=20;
```

First Expression: Expression to be tested

2nd Expression: Executes if first expression is true

3rd Expression: Executes if the first expression is false
5.12

Breaking and Continuing a Loop

Breaking Out of a Loop

• Can use `break` to terminate execution of a loop

• Use sparingly if at all – makes code harder to understand and debug

• When used in an inner loop, terminates that loop only and goes back to outer loop
The **continue** Statement

- Can use `continue` to go to end of loop and prepare for next repetition
  - `while`, `do-while` loops: go to test, repeat loop if test passes
  - `for` loop: perform update step, then test, then repeat loop if test passes
- Use sparingly – like `break`, can make program logic hard to follow
5.1 Introduction to Functions

• **Function**: group of statements within a program that perform as specific task
  – Usually one task of a large program
    • Functions can be executed in order to perform overall program task
  – Known as *divide and conquer* approach

• **Modularized program**: program wherein each task within the program is in its own function
5.1 Introduction to Functions

• The benefits of using functions include:
  – Simpler code
  – Code reuse
    • write the code once and call it multiple times
  – Better testing and debugging
    • Can test and debug each function individually
  – Faster development
  – Easier facilitation of teamwork
    • Different team members can write different functions

• You will learn about two types of functions:
  – `void` functions
    • A `void` function simply performs a task, and then terminates.
  – Value-returning functions
    • A value-returning function performs a task and then returns a value back to the part of the program that executed the function.
5.2 void Functions

• A function definition includes:
  – return type: The data type of the value that function returns to the part of the program that called it. If the word void is used, the function does not return a value.
  – 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 {}
5.2 void Functions

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

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

• After the function terminates, execution resumes in the calling function at point of call.

```
#include <iostream>

using namespace std;

// Definition of the displayMessage function
void displayMessage()
{
    cout << "This is the displayMessage function." << endl;
}

// Definition of the main function
int main()
{
    cout << "This is the main function." << endl;
    displayMessage();
    cout << "Back in the main function." << endl;
    return 0;
}
```

Program Output

This is the main function.
This is the displayMessage function.
Figure 5.3 Calling the `displayMessage` function

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

// Definition of the `displayMessage` function
void displayMessage()
{
    cout << "This is the `displayMessage` function." << endl;
}

// Definition of the `main` function
int main()
{
    cout << "This is the main function." << endl;
    displayMessage();
    cout << "Back in the main function." << endl;
    return 0;
}
```

The program jumps to the `displayMessage` function and executes the statements in its body.

Figure 5.4 The `displayMessage` function returns

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

// Definition of the `displayMessage` function
void displayMessage()
{
    cout << "This is the `displayMessage` function." << endl;
}

// Definition of the `main` function
int main()
{
    cout << "This is the main function." << endl;
    displayMessage();
    cout << "Back in the main function." << endl;
    return 0;
}
```

When the `displayMessage` function ends, the program returns to the part of the program that called it, and resumes execution at that point.
5.2 void 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

5.2 void Functions

• 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 – like the function definition without the body
    • Header: void displayMessage()
    • Prototype: void displayMessage();
5.2 **void Functions**

```
#include <iostream>
2 using namespace std;
3
4 // Function prototype
5 void displayMessage();
6
7 // Definition of the main function
8 int main()
9 {
10    cout << "This is the main function." << endl;
11    displayMessage();
12    cout << "Back in the main function." << endl;
13    return 0;
14 }
15
16 // Definition of the displayMessage function
17 void displayMessage()
18 {
19    cout << "This is the displayMessage function." << endl;
20 }
```

**Program Output**

This is the main function.
This is the displayMessage function.
Back in the main function.

Copyright © 2014 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
5.2 void Functions

• 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

5.2 void Functions

• A design technique known as top-down design (also known as stepwise refinement) can be used to break down an algorithm into functions by the following steps:
  – The overall task is broken down into a series of subtasks.
  – Each of the subtasks is repeatedly examined to determine if it can be further broken down.
  – Each subtask is coded.
5.3 Local Variables

- A **local variable** is declared inside a function and cannot be accessed by statements that are outside the function.

- More specifically:
  - A local variable's scope begins at the variable’s declaration and ends at the end of the function in which the variable is declared.
  - The variable cannot be accessed by statements that are outside this region.

```cpp
// This program has an error!
#include <iostream>
#include <string>
using namespace std;

// Function prototype
void getName();

// Definition of main function
int main()
{
    getName();
    cout << "Hello " << name << endl;
}

// Definition of getName function
void getName()
{
    string name;
    cout << "Enter your first name: ";
    cin >> name;
}
```
5.3 Local Variables

- You cannot have two variables with the same name in the same scope.

```c
void getTwoAges()
{
    int age;
    cout << "Enter your age: ";
    cin >> age;
    int age; // This will cause an error because
    cout << "Enter your pet's age: ";
    cin >> age; // the age variable has already
} // been declared.
```

5.4 Passing Arguments to Functions

- **Argument**: piece of data that is sent into a function
  - Function can use argument in calculations
  - When calling the function, the argument is placed in parentheses following the function name
  - A special variable known as a *parameter* is declared in the function header to receive the argument
Program 5-3  (PassArgument.cpp)

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

// Function prototype
void doubleNumber(int);

// Definition of the main function
int main()
{
    doubleNumber(4);
    return 0;
}

// Definition of the doubleNumber function
void doubleNumber(int number)
{
    int result = number * 2;
    cout << result << endl;
}
```

Program Output

8

Copyright © 2014 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
```cpp
int main()
{
    int value;

    cout << "Enter a value and I will double it: ";
    cin >> value;
    doubleNumber(value);
    return 0;
}

void doubleNumber(int number)
{
    int result = number * 2;
    cout << result << endl;
}
```

The contents of the value variable are assigned to the number parameter variable.

Copyright © 2014 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
5.4 Passing Arguments to Functions

• 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
```

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.
# Program 5.5 (TwoArgs.cpp)

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

// Function prototype
void showSum(int, int);

// Definition of the main function
int main()
{
    cout << "The sum of 12 and 45 is:" << endl;
    showSum(12, 45);
    return 0;
}

// Definition of the showSum function
void showSum(int num1, int num2)
{
    int result = num1 + num2;
    cout << result << endl;
}
```

**Program Output**
The sum of 12 and 45 is: 57
5.4 Passing Arguments to Functions

- **Pass by value**: when an argument is passed to a function, only its value is copied into the parameter.

- Changes to the parameter in the function do not affect the value of the argument.

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

// Function prototype
void changeVal(int);

int main()
{
    int number = 99;
    int
```
5.4 Passing Arguments to Functions

- Pass by **Reference** means that the argument is passed into a reference variable.
  - Two-way communication: Calling function can communicate with called function; and called function can modify the value of the argument.
  - To pass by reference, the parameter must be a reference variable.
  - Reference variables are declared like regular variables, except you place an ampersand ( & ) in front of the name.

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

// Function prototype
void setToZero(int &);

int main()
{
    int value = 99;
    cout << "The value is " << value << endl;
    setToZero(value);
    cout << "Now the value is " << value << endl;
    return 0;
}

// Definition of the setToZero function
void setToZero(int &num)
{
    num = 0;
}
```

**Program Output**
The value is 99.
Now the value is 0
5.5 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.

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

// Function prototype
void anotherFunction();

// Global variable declaration
int num = 2;

// Definition of the main function
int main()
{
    cout << "In main, num is " << num << endl;
    anotherFunction();
    cout << "Back in main, num is " << num << endl;
    return 0;
}

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

Program Output:
In main, num is 2
In anotherFunction, num is 2
But, it is now changed to 50
Back in main, num is 50
5.5 Global Variables and Global Constants

- A global constant is a named constant that is available to every function in the program.
- Global constants are typically used to represent unchanging values that are needed throughout a program.

5.6 Value-Returning Functions

- 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);
```
5.6 Value-Returning Functions

• When writing your own value-returning functions:
  – You must specify a data type for the function. The value that is returned from the function must be of the specified data type.
  – A value-returning function must have a return statement. The return statement causes a value to be returned from the function.
Program 5-10  (ValueReturn.cpp)

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

// Function prototypes
int sum(int, int);

// Definition of the main function
int main()
{
    // Variables
    int userAge,     // The user's age
        friendAge,  // A friend's age
        combinedAge; // The combined age

    // Get the user's age.
    cout << "What is your age? ";
    cin >> userAge;

    // Get a friend's age.
    cout << "What is your friend's age? ";
    cin >> friendAge;

    // Get the combined age.
    combinedAge = sum(userAge, friendAge);

    // Display the combined age.
    cout << "Your combined age is "
         << combinedAge << " years."
         << endl;
    return 0;
}
```

Program Output with Input Shown in Bold

What is your age? 23 [Enter]
What is your friend's age? 25 [Enter]
Your combined age is 48 years.
5.6 Value-Returning Functions

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

```plaintext
combinedAge = sum(userAge, friendAge);
```

Copyright © 2014 Pearson Education, Inc. Publishing as Pearson Addison-Wesley
5.6 Value-Returning Functions

- A 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

Example:

```cpp
bool isEven(int number)
{
    // Local variable to hold true or false
    bool numberIsEven;

    // Determine whether the number is even.
    if (number % 2 == 0)
    {
        numberIsEven = true;
    }
    else
    {
        numberIsEven = false;
    }
    // Return the result.
    return numberIsEven;
}
```
Calling the `isEven` function shown on the previous slide:

```cpp
if (isEven(number))
{
    cout << "The number is even." << endl;
}
else
{
    cout << "The number is odd." << endl;
}
```

5.6 Value-Returning Functions

- A function can return any type of data, including strings.

```cpp
string fullName(string first, string middle, string last)
{
    return first + " " + middle + " " + last;
}
```

*From Program 5-12*
5.7 Calling `string` Member Functions

- Objects created from the `string` class have built-in functions that are known as member functions.
- A *member function* is a function that operates on a specific object’s data.

You call a member function in a slightly different manner than a regular function. Here is the general format:

```
objectName.functionName(arguments)
```
5.7 Calling string Member Functions

• For example, a string object's length member function returns the length of the string that is stored in the object:

```cpp
string state = "Texas";
int size;
size = state.length();
```

• The c_str() member function:
  – You will use this function quite a bit in Part 2 of the book.
  – It returns the value of a string object formatted as a null terminated string.
  – This is a particular format that is required by the App Game Kit library.
5.7 Calling string Member Functions

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.
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 defined for values that do not change throughout the program’s execution.

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

```c++
// This program calculates gross pay.
#include <iostream>
#include <iomanip>
using namespace std;

// Global constants
const double PAY_RATE = 22.55; // Hourly pay rate
const double BASE_HOURS = 40.0; // Max non-overtime hours
const double OT_MULTIPLIER = 1.5; // Overtime multiplier

// Function prototypes
double getBasePay(double);
double getOvertimePay(double);

int main()
{
    double hours, // Hours worked
    basePay, // Base pay
    overtime = 0.0, // Overtime pay
    totalPay; // Total pay

    // Get overtime pay, if any.
    if (hours > BASE_HOURS)
    {
        overtime = getOvertimePay(hours);
    }

    // Determine base pay.
    if (hoursWorked > BASE_HOURS)
    {
        basePay = BASE_HOURS * PAY_RATE;
    }
    else
    {
        basePay = hoursWorked * PAY_RATE;
    }

    // Determine overtime pay.
    if (hoursWorked > BASE_HOURS)
    {
        overtimePay = (hoursWorked - BASE_HOURS) * PAY_RATE * OT_MULTIPLIER;
    }
```
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.

6.11

Static Local Variables
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

```c
// This program shows that local variables do not retain
// their values between function calls.
#include <iostream>
using namespace std;

// Function prototype
void showLocal();

int main()
{
    showLocal();
    showLocal();
    return 0;
}
```

(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

```cpp
// This program uses a static local variable.
#include <iostream>
using namespace std;

void showStatic(); // Function prototype

int main()
{
    // Call the showStatic function five times.
    for (int count = 0; count < 5; count++)
        showStatic();
    return 0;
}
```

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

```cpp
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 = 5;
23    cout << "statNum is " << statNum << endl;
24    statNum++;
25 }
```

**Program Output**
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9

---

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

(Program Continues)

---

Program 6-23 (Continued)

```cpp
// The default argument for cols is 10 and for rows is 1.
// This function displays a square made of asterisks.
void displayStars(int cols, int rows)
{
    // Nested loop. The outer loop controls the rows
    // and the inner loop controls the columns.
    for (int down = 0; down < rows; down++)
    {
        for (int across = 0; across < cols; across++)
            cout << "*";
        cout << endl;
    }
}
```

---

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, 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
```
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
Program 6-25

// 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;
}

(Program Continues)

---

Program 6-25 (Continued)

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

Here we are passing value by reference.

The & also appears here in the function header.

---

Program Output
In main, value is 4
Now calling doubleNum...
Now back in main, value is 8
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
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, 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
The overloaded functions have different parameter lists.

Passing an int
Passing a double

(Program Continues)

Program 6-27 (Continued)
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:
  ```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
Using Files for Data Storage

• Can use files instead of keyboard, monitor screen for program input, output
• Allows data to be retained between program runs
• Steps:
  – *Open* the file
  – *Use* the file (read from, write to, or both)
  – *Close* the file

Files: What is Needed

• *Use* `fstream` header file for file access
• File stream types:
  - `ifstream` for input from a file
  - `ofstream` for output to a file
  - `fstream` for input from or output to a file
• Define file stream objects:
  - `ifstream infile;`
  - `ofstream outfile;`
Opening Files

- Create a link between file name (outside the program) and file stream object (inside the program)
- Use the `open` member function:
  ```
  infile.open("inventory.dat");
  outfile.open("report.txt");
  ```
- Filename may include drive, path info.
- Output file will be created if necessary; existing file will be erased first
- Input file must exist for `open` to work

Testing for File Open Errors

- Can test a file stream object to detect if an open operation failed:
  ```
  infile.open("test.txt");
  if (!infile)
  {
    cout << "File open failure!";
  }
  ```
- Can also use the `fail` member function
Using Files

• Can use output file object and `<<` to send data to a file:
  
  ```
  outfile << "Inventory report";
  ```

• Can use input file object and `>>` to copy data from file to variables:
  
  ```
  infile >> partNum;
  infile >> qtyInStock >> qtyOnOrder;
  ```

Using Loops to Process Files

• The stream extraction operator `>>` returns `true` when a value was successfully read, `false` otherwise

• Can be tested in a `while` loop to continue execution as long as values are read from the file:
  
  ```
  while (inputFile >> number) ...
  ```
Closing Files

• Use the close member function:
  
  ```cpp
  infile.close();
  outfile.close();
  ```

• Don’t wait for operating system to close files at
  program end:
  – may be limit on number of open files
  – may be buffered output data waiting to send to file

Letting the User Specify a Filename

• The open member function requires that you
  pass the name of the file as a null-terminated
  string, which is also known as a C-string.

• String literals are stored in memory as null-
  terminated C-strings, but string objects are not.
Letting the User Specify a Filename

• string objects have a member function named c_str
  – It returns the contents of the object formatted as a null-terminated C-string.
  – Here is the general format of how you call the c_str function:

        stringObject.c_str()
Letting the User Specify a Filename in Program 5-24

```cpp
22 { // Read the numbers from the file and
23 // display them.
24 while (inputFile >> number)
25 { cout << number << endl;
26 }
27 // Close the file.
28 inputFile.close();
29 }
30 else
31 { // Display an error message.
32 cout << "Error opening the file.\n";
33 return 0;
34 }
35 // Display an error message.
36 cout << "Error opening the file.\n";
37 return 0;
38 }
39 }
```

Program Output with Example Input Shown in Bold

Enter the filename: ListOfNumbers.txt [Enter]
100
200
300
400
500
600
700