CSCI 123 Introduction to Programming Concepts in C++

Brad Rippe

Function Pass By Reference
What’s wrong?

for(int star = 0; star < 10; star++) {
    ...
}
for(star = 0; star < 10; star++) {
    ...
}

Style – Open and End Brackets
Indenting
Overview

5.1 \textit{void} Functions

5.2 Call-By-Reference Parameters

5.3 Using Procedural Abstraction

5.4 Testing and Debugging
void-Functions

• In top-down design, a subtask might produce
  – No value (just input or output for example)
  – One value
  – More than one value

• We have seen how to implement functions that return one value

• A void-function implements a subtask that returns no value or more than one value
void-Function Definition

• Two main differences between void-function definitions and the definitions of functions that return one value
  – Keyword void replaces the type of the value returned
    • void means that no value is returned by the function
  – The return statement does not include an expression or value

• Example:
  void promptForLowerChar() {
    cout << "Please type a lowercase character ";
    cout << "and I will make it uppercase.\n";
    return;
  }
Using a void-Function

• void-function calls are executable statements
  – They do not need to be part of another statement
  – They end with a semi-colon

• Example:

```
promptForLowerChar();
cin >> theChar;
outputUpperCase(theChar);
```

**NOT:** cout << promptForLowerChar();

• May accept parameters or not
void-Function Calls

• Mechanism is nearly the same as the function calls we have seen
  – Argument values are substituted for the formal parameters
    • It is fairly common to have no parameters in void-functions
      – In this case there will be no arguments in the function call
  – Statements in function body are executed
  – Optional return statement ends the function
    • Return statement does not include a value to return
    • Return statement is implicit if it is not included
void fixACar(int numberOfCarsToFix) {
    if (numberOfCarsToFix == 0)
        return;
    int carsCounter = 1;
    cin >> hours;
    cout << "Your current charges are: ";
    cout << calculateCharges(hours) << endl;
    return;
}
void-Functions
Why Use a Return?

• Is a return-statement ever needed in a void-function since no value is returned?
  – Yes!
    • What if a branch of an if-else statement requires that the function ends to avoid producing more output, or creating a mathematical error?
    • void-function in next example, avoids division by zero with a return statement
void outputDivide(int op1, int op2) {
    if(op2 == 0)
        return;
    else
        cout << (op1/op2) << endl;
}
The Main Function

- The main function in a program is used like a void function...do you have to end the program with a return-statement?
  - Because the main function is defined to return a value of type int, the return is needed
  - C++ standard says the return 0 can be omitted, but many compilers still require it
Section 5.1 Conclusion

• Can you
  – Describe the differences between void-functions and functions that return one value?
  – Tell what happens if you forget the return-statement in a void-function?
  – Distinguish between functions that are used as expressions and those used as statements?
  • Return value the other returns void or nothing
Call-By-Reference Parameters

5.2
Call-by-Reference Parameters

• Call-by-value is not adequate when we need a sub-task to obtain input values
  – Call-by-value means that the formal parameters receive the values of the arguments
  – To obtain input values, we need to change the variables that are arguments to the function
    • Recall that we have changed the values of formal parameters in a function body, but we have not changed the arguments found in the function call
• Call-by-reference parameters allow us to change the variable used in the function call
  – Arguments for call-by-reference parameters must be variables, not numbers
void square(int& aVar) {
    aVar *= aVar;
    cout << "\tinside " << aVar << endl;
}

• ‘&’ symbol (ampersand) identifies “a” as a call-by-reference parameter
  – Used in both declaration and definition!

• The book places the ‘&’ at the end of the formal parameters type, but it may be placed at the beginning of the variable name

• I like the book’s style
Call-By-Reference Details

- Compare what happens with pass-by-value to what happens with pass-by-reference
- Call-by-reference works almost as if the argument variable is substituted for the formal parameter, not the argument’s value
- In reality, the memory location of the argument variable is given to the formal parameter
  - Whatever is done to a formal parameter in the function body, is actually done to the value at the memory location of the argument variable
Call Comparisons
Call By Reference vs Value

• Call-by-reference
  – The function call:
    `getSubmarines(subs, torps);`
    `void getSubs(int& subs, int& torps);`

  - Memory table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>subs</td>
<td>1001</td>
<td>0</td>
</tr>
<tr>
<td>torps</td>
<td>1002</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1003</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1004</td>
<td></td>
</tr>
</tbody>
</table>

• Call-by-value
  – The function call:
    `getSubmarines(subs, torps);`
    `void getSubs(int subs, int torps);`
Example: swap

- void swap(int& variable1, int& variable2) {
  int temp = variable1;
  variable1 = variable2;
  variable2 = temp;
}

- If called with swap(num1, num2);
  - num1 is substituted for variable1 in the parameter list
  - num2 is substituted for variable2 in the parameter list
  - temp is assigned the value of variable1 (num1) since the next line will lose the value in num1
  - variable1 (num1) is assigned the value in variable2 (num2)
  - variable2 (num2) is assigned the original value of variable1 (num1) which was stored in temp
Mixed Parameter Lists

• Call-by-value and call-by-reference parameters can be mixed in the same function

• Example:
  void getSubmarines(int& submarines, int& torpedos, int depthCharges);
  
  – submarines and torpedoes are call-by-reference formal parameters
    • Changes in submarines and torpedos change the argument variable
  – depthCharges is a call-by-value formal parameter
    • Changes in depthCharges do not change the argument variable

• Side Note: One disadvantage of pass-by-value is that, if a large data item is being passed, copying that data can take a considerable amount of execution time and memory space.
Choosing Parameter Types

• How do you decide whether a call-by-reference or call-by-value formal parameter is needed?
  – Does the function need to change the value of the variable used as an argument?
    – Yes? Use a call-by-reference formal parameter
    – No? Use a call-by-value formal parameter
Inadvertent Local Variables

• If a function is to change the value of a variable the corresponding formal parameter must be a call-by-reference parameter with an ampersand (&) attached

• Forgetting the ampersand (&) creates a call-by-value parameter
  – The value of the variable will not be changed
  – The formal parameter is a local variable that has no effect outside the function
  – Hard error to find...it looks right!
Section 5.2 Conclusion

- Can you
  - Write a void-function definition for a function called zeroBoth that has two reference parameters, both of which are variables of type int, and sets the values of both variables to 0.
  - Write a function that returns a value and has a call-by-reference parameter?
  - Write a function with both call-by-value and call-by-reference parameters
Using Procedural Abstraction

5.3
Using Procedural Abstraction

• Functions should be designed so they can be used as black boxes
• To use a function, the declaration and documentation should be sufficient
• Programmer should not need to know the details of the function to use it
Functions Calling Functions

• A function body may contain a call to another function
  – The called function declaration must still appear before it is called
• Functions cannot be defined in the body of another function
  – Example:

```cpp
void setDepthCharges(int& submarines, int& torpedoes, int depthCharges) {
    getSubsAndTorpedoes(submarines, torpedoes);
    for(int i = 0; i < depthCharges; i++) {
        int hitSub = 1 + rand() % (submarines * 2);
        if(hitSub <= submarines) {
            cout << "Submarine " << hitSub << " has been hit by ";
            cout << "a depth charge\n";
        } else {
            cout << "Depth charge " << (i+1) << " missed.\n";
        }
    }
}
```
Pre and Postconditions - Documentation

• Precondition – uses the @pre annotation
  – States what is assumed to be true when the function is called
    • Function should not be used unless the precondition holds
• Postcondition – uses the @post annotation
  – Describes the effect of the function call
  – Tells what will be true after the function is executed (when the precondition holds)
  – If the function returns a value, that value is described
  – Changes to call-by-reference parameters are described
swapByRef Documentation

• Using preconditions and postconditions the declaration of `swapByRef` becomes:

```cpp
/**
 * Swaps the values of variable 1 with variable 2
 *
 * @param var1 the value of variable 1
 * @param var2 the value of variable 2
 * @pre var1 and var2 have been initialized to values
 * @post the value of variable 1 has been replaced with the value
 * of variable 2 and the value of variable 2 has been replaced
 * with the value of variable 1
 */
void swapByRef(int& var1, int& var2);
```
Function outputVars

- Preconditions and postconditions make the declaration for outputVars:

/**
 * This function outputs the values in var1 and var2
 * It is intended use is after one of the swap functions
 * have been called.
 * @param funcName the name of the swap function that was called before
 * @param outputVars
 * @param var1 the value of variable 1
 * @param var2 the value of variable 2
 * @pre variable 1 and 2 have been swapped by one of the swap functions
 * @post The name of the swap function and the values of variables 1 and 2
 * have been output to the user
 */

void outputVars(string& funcName, int var1, int var2);
Why use preconditions and postconditions?

• Preconditions and postconditions
  – should be the first step in designing a function
  – specify what a function should do
    • Always specify what a function should do before designing how the function will do it
  – Minimize design errors
  – Minimize time wasted writing code that doesn’t match the task at hand
Case Study
Garage Pricing

• Problem definition
  – Determine the parking charge for 3 cars parked
  – Flat fee of $2.00 for up to and including 3 hour of parking
  – Flat fee $10.00 for a car parked for 24 hours
    • No car will exceed 24 hours
  – $0.50 charge for each hour or part of an hour in excess of three hours
– Input
  • The hours that each car was parked in the garage
– Output
  • The hours for each car, the charge, the total hours and the total charges for the day
Garage Pricing: Problem Analysis

• Three main subtasks
  – Input the data
  – Compute the charges based on the hours
  – Output the results

• Each task can be implemented with a function
  – Notice the use of call-by-value and call-by-reference parameters in the following function declarations
Garage Pricing:
Function getHoursForCars

/**
 * Prompt the user to input hours for three cars and sets the hours for each car
 * @param hour1 the hours for car 1
 * @param hour2 the hours for car 2
 * @param hour3 the hours for car 3
 * @pre hours have been declared
 * @post hours 1 2 and 3 have been sent from user input
 */

void getHoursForCars(double& hours1, double& hours2, double& hours3);
Garage Pricing: Function setDecimalFormat

/**
 * Sets the cout object to display decimal digits with two decimal places
 * @pre none
 * @post cout will display decimal digits with two decimal places
 */

void setDecimalFormat();
Garage Pricing: Function outputCharges

/**
 * Outputs the charges for all three cars.
 * @param hour1 the hours for car 1
 * @param hour2 the hours for car 2
 * @param hour3 the hours for car 3
 * @pre hours have been initialized and the cout object has been manipulated to output decimal digits with two decimal places
 * @post hours and charges for each car has been output in addition to the total hours and charges for all three cars
 */

void outputCharges(double hours1, double hours2, double hours3);
Garage Pricing: The main function

• With the functions declared, we can write the main function:

```java
int main() {
    double hours1;
    double hours2;
    double hours3;
    
    getHoursForCars(hours1, hours2, hours3);
    setDecimalFormat();
    outputCharges(hours1, hours2, hours3);
}
```
Algorithm Design calculateCharges

- pseudocode for the price function
  if hours <= 3
    return $2.00
  if hours == 24
    return $10.00
  otherwise
    return 2.00 + roundup (hours-3)*0.50;
Coding The calculateCharges Function

- The body of the price function

```java
double calculateCharges(double hours) {
    if(hours <= 3) {
        return 2.0;
    } else if(hours == 24) {
        return 10.0;
    } else {
        return 2.0 + ceil(hours-3.0)*.50;
    }
}
```
Garage Pricing: Program Testing

- Testing strategies
  - Use data that tests both the high and low markup cases
  - Test boundary conditions, where the program is expected to change behavior or make a choice
    - 3 hours, 4 hours and 24 hours
    - Test for exactly 4.5 hours make sure your program is rounding up
      - 3 = 2.00
      - 4.5-3 = 1.5
      - 2*.5 = 1.00
      - Total 3.00
Section 5.3 Conclusion

• Can you
  – Define a function in the body of another function?
  – Call one function from the body of another function?
  – Give preconditions and postconditions for the predefined function sqrt?
5.4

Testing and Debugging
Testing and Debugging Functions

- Each function should be tested as a separate unit
- Testing individual functions facilitates finding mistakes
- Driver programs allow testing of individual functions
- Once a function is tested, it can be used in the driver program to test other functions
- Function `getSubsAndTorpedoes` is tested in the driver program
void getSubsAndTorpedoes(int& submarines, int& torpedoes) ;

int main() {
    int submarines = 0;
    int torpedoes = 0;
    int depthCharges = 10;
    char again = 'Y';

    while(toupper(again) != 'N') {
        cout << "Type integers for subs and torpedoes\n";
        getSubsAndTorpedoes(submarines, torpedoes);
        cout << "Subs set by the user " << submarines << endl;
        cout << "Torpedoes set by the user " << torpedoes << endl;
        cout << "Would you like to continue?\n";
        cout << "Y for yes and N for no\n";
        cin >> again;
    }
    return 0;
}
void getSubsAndTorpedoes(int& submarines, int& torpedoes) {
    cin >> submarines;
    cin >> torpedoes;
    cout << "You have typed ";
    cout << submarines << " submarines and ";
    cout << torpedoes << " torpedoes\n";
}
Stubs

• When a function being tested calls other functions that are not yet tested, use a stub
• A stub is a simplified version of a function
  – Stubs are usually provide values for testing rather than perform the intended calculation
  – Stubs should be so simple that you have confidence they will perform correctly
  – Function `getSubsAndTorpedoes` is used as a stub to test the rest of the submarine program
getSubsAndTorpedoes is now a stub
* suppose the requirement was to read subs and torpedoes from a file since this code hasn't been tested, we can use a stub
* to test setDepthCharges before we tackle the problem of reading submarines from a file

```c
void getSubsAndTorpedoes(int& submarines, int& torpedoes) {
    submarines = 5;
    torpedoes = 2;
}
```
void setDepthCharges(int& submarines, int& torpedoes, int depthCharges) {

    getSubsAndTorpedoes(submarines, torpedoes);

    for(int i = 0; i < depthCharges; i++) {
        int hitSub = 1 + rand() % (submarines * 2);
        if(hitSub <= submarines) {
            cout << "Submarine " << hitSub << " has been hit by ";
            cout << "a depth charge\n";
            submarines--;
        } else {
            cout << "Depth charge " << (i+1) << " missed.\n";
        }
    }
}
Rule for Testing Functions

• Fundamental Rule for Testing Functions
  – Test every function in a program in which every other function in that program has already been fully tested and debugged.