Chapter 5

Functions for All Subtasks

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electrical Engineering
National Chiao-Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nctu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

5.1 More about 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), one value (typical case), or more than one value
- A void-function implements a subtask that returns no value or more than 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 any expression

```
void showResults(double f_degrees, double c_degrees)
 {
     cout << f_degrees
          << " degrees Fahrenheit is euivalent to " << endl
          << c_degrees << " degrees Celsius." << endl;
     return;
 }
```


Syntax for void Function

void Function Declaration

```
void Function_Name(Parameter_List);
Function_Declaration_Comment
```

void Function Definition

```
void Function_Name(Parameter_List)
{
        Declaration_1
        Declaration_2
        ...
        Declaration_Last

        Executable_Statement_1
        Executable_Statement_2
        ...
        Executable_Statement_Last
}
```

function header

You may intermix the declarations with the executable statements.

body

May (or may not) include one or more return statements.

void-Function Calls

- Mechanism is nearly the same as the function calls we have seen
  - Argument values are passed to 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

    → automatically return when the function is finished

Example for void Functions

```
//DISPLAY 5.2 void Functions
#include <iostream>
void initializeScreen( );
//Separates current output from
//the output of the previously run program.
double celsius(double fahrenheit);
//Converts a Fahrenheit temperature
//to a Celsius temperature.
void showResults(double fDegrees, double cDegrees);
//Displays output. Assumes that cDegrees
//Celsius is equivalent to fDegrees Fahrenheit.

int main( )
{
    using namespace std;
    double fTemperature, cTemperature;

    initializeScreen( );
    cout << "I will convert a Fahrenheit temperature"
         << " to Celsius.\n"
         << "Enter a temperature in Fahrenheit: ";
    cin >> fTemperature;

    cTemperature = celsius(fTemperature);
    showResults(fTemperature, cTemperature);
    return 0;
}
```

```
void initializeScreen( )
{
    using namespace std;
    cout << endl;
    return;
}

double celsius(double fahrenheit)
{
    return ((5.0/9.0)*(fahrenheit - 32));
}

void showResults(double fDegrees, double cDegrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << fDegrees
         << " degrees Fahrenheit is equivalent to\n"
         << cDegrees << " degrees Celsius.\n";
    return;
}
```

optional

optional

Sample Dialogue

```
I will convert a Fahrenheit temperature to Celsius.
Enter a temperature in Fahrenheit: 32.5
32.5 degrees Fahrenheit is equivalent to
0.3 degrees Celsius.
```

Return Statement in void-Functions

- No return value can be expected while calling a void-function
  - Correct: showResults(32.5, 0.3);
    Wrong: cout << showResults(32.5, 0.3);
- 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 early?
    - Avoid producing more output or creating a mathematical error, as shown in next example

Ex: Using *return* in void Function

Function Declaration

```
1   void iceCreamDivision(int number, double totalWeight);
2   //outputs instructions for dividing totalWeight ounces of
3   //ice cream among number customers.
4   //If number is 0, nothing is done.
```

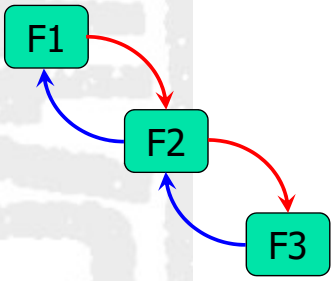
Function Declaration

```
1   //Definition uses iostream:
2   void iceCreamDivision(int number, double totalWeight)
3   {
4           using namespace std;
5           double protion;
6
7           if (number == 0)
8               return;
9           portion = totalWeight/Number;
10          cout.setf(ios::fixed);
11          cout.setf(ios::showpoint);
12          cout.precision(2);
13          cout << "Each one receives "
14                  << portion << " ounces of ice cream." << endl;
15  }
```

If number is 0, finish the function at here.

Functions Calling Functions

- A function body may contain a call to another function
  - Ex:
```
void order(int&  n1, int&  n2)
{
    if (n1 > n2)
        swapValues(n1, n2);
}
```
    - *swapValues* is another function to make n1 and n2 in ascending order
  - Return to the upper level only, not the top level



- The called function must be declared before it is used
  - Functions cannot be defined in the body of another function → often put all function declarations at top

Recursive Function Call

- A recursive function is a function that calls itself, either directly, or indirectly (through another function).
- The function only knows how to solve the simplest case(s), or so-called base case(s)
  - If the function is called with a base case, the function simply returns a result
- The recursive function divides a complex problem into
  - What it can do (base case) → return the result
  - What it cannot do → resemble the original problem, but be a slightly simpler or smaller version
  - The function calls a new copy of itself (recursion step) to solve the smaller problem
- Eventually base case gets solved
  - Return the result to solve the problem at upper level

Example: Factorial Function

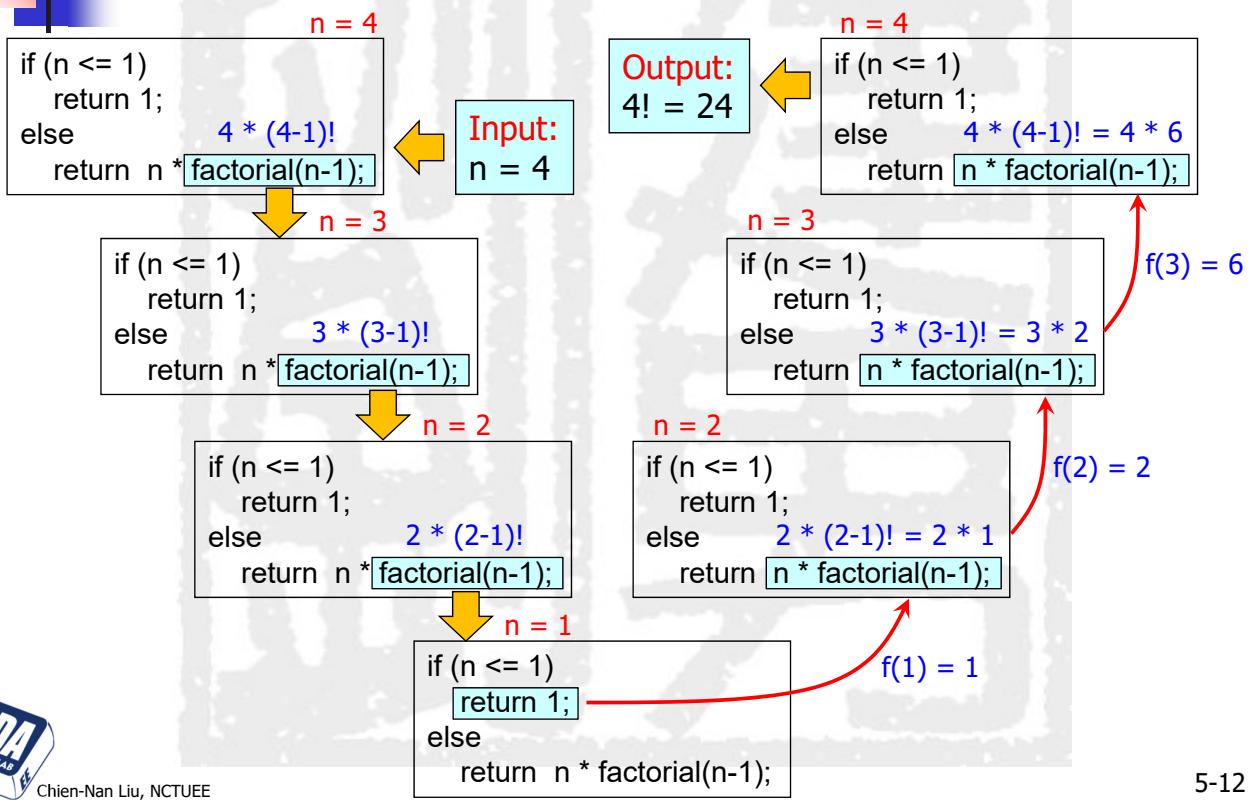
- n! = n x (n-1) x (n-2) x … x 2 x 1
  = n x (n-1)!
- 1! = 1, 0! = 1 → base case
- This function can be solved iteratively or recursively

```
// Iterative version
int factorial(int n)
{
    int product = 1;
    while (n > 0)
    {
        product = n * product;
        n--;
    }
    return product;
}
```

```
// Resursive version
int factorial(int n)
{
    if (n <= 1)   // base case
        return 1;
    else          // recursive step
        return  n * factorial(n-1);
}
```

n x (n-1)!

Recursive Execution of factorial(4)



Another Example: Power (X^y)

- $X^n = X * X * X * \ldots * X$
  $= X * X^{(n-1)}$
- $X^0 = 1 \rightarrow$ base case
- This function can be solved iteratively or recursively
  - For more details about recursion, please check Chapter 14

```
// Iterative version
int power(int x, int y)
{
    int product = 1;
    for (int i=1; i<=y; i++)
        product = x * product;

    return product;
}
```

```
// Resursive version
int power(int x, int y)
{
    if (y < 1)    // base case
        return 1;
    else          // recursive step
        return  x * power(x, y-1);
}
```

$X * X^{(n-1)}$

Recursion v.s. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
  - Make sure the termination condition eventually occurs
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)
  - Avoid using recursion in performance situations
    - Recursive calls take time and consume additional memory

Overview

5.1 More about Functions

5.2 Call-By-Reference Parameters

5.3 Using Procedural Abstraction

5.4 Testing and Debugging

Call-by-Reference Parameters

- Call-by-value (default mechanism) means that the formal parameters receive values only
  - Changing the values of internal variables will not change the original data
- Call-by-reference parameters allow us to change the variable used in the function call
  - Pass the "address" for exchanging data between functions
  - Arguments for call-by-reference parameters must be variables, not constant numbers

Call-by-Reference Example

```
void getInput(double& fVariable)
 {
       using namespace std;
       cout << " Convert a Fahrenheit temperature to Celsius.\n"
            << " Enter a temperature in Fahrenheit: ";
       cin  >> fVariable;
}
```

- '**&**' symbol (ampersand) identifies *fVariable* as a call-by-reference parameter
  - Used in both declaration and definition !!
- Whatever is done to a formal parameter in the function, is actually done to the value at the given memory addr.
  - Work almost as if the argument variable substitutes the formal parameter, not the argument's value

Code with Call-by-Reference

```
//DISPLAY 5.4 Call-by-Reference Parameters
#include <iostream>

void getNumbers(int& input1, int& input2);
//Reads two integers from the keyboard.

void swapValues(int& variable1, int& variable2);
//Interchanges the values of variable1 and variable2.

void showResults(int output1, int output2);
//Shows the values of variable1 and variable2 in order.

int main( )
{
   int firstNum, secondNum;

   getNumbers(firstNum, secondNum);
   swapValues(firstNum, secondNum);
   showResults(firstNum, secondNum);
   return 0;
}
```

Sample Dialogue

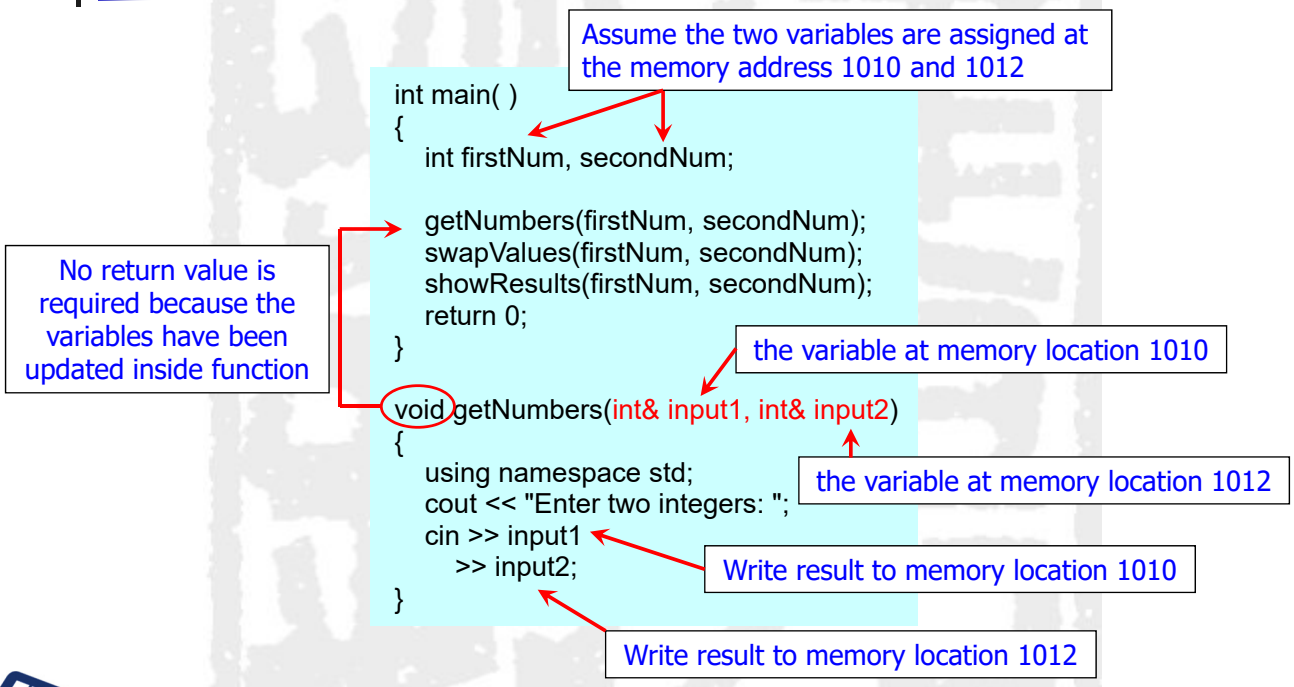
```
Enter two integers: 5 10
In reverse order the numbers are: 10 5
```

```
//Uses iostream:
void getNumbers(int& input1, int& input2)
{
   using namespace std;
   cout << "Enter two integers: ";
   cin >> input1
       >> input2;
}

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

//Uses iostream:
void showResults(int output1, int output2)
{
   using namespace std;
   cout << "In reverse order the numbers are: "
        << output1 << " " << output2 << endl;
}
```

Behavior of Call-by-Reference



Call by Reference v.s. Value

- Call-by-reference
  - The function definition:
    void f(int& ref_par);
  - The function call:
    f(age);
- Call-by-value
  - The function definition:
    void f(int var_par);
  - The function call:
    f(age);

The same address with two different names

Two different variables with the same value

Memory

Name	Location	Contents
age	**1001**	**34**
initial	1002	A
hours	1003	23.5
	1004	

Example: swapValues

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

- If called with swap(firstNum, secondNum);
  - firstNum is substituted for variable1 in the parameter list
  - secondNum is substituted for variable2 in the parameter list
  - temp is assigned the value of variable1 (firstNum) since the next line will loose the value in firstNum
  - variable1 (firstNum) is assigned the value in variable2 (secondNum)
  - variable2 (secondNum) is assigned the original value of variable1 (firstNum) which was stored in temp

Choosing Parameter Types

- Call-by-value and call-by-reference parameters can be mixed in the same function
- Ex: void goodStuff(int& par1, int par2, double& par3);
  - par1 and par3 are call-by-reference parameters
    - Changes in par1 and par3 change the argument variable
  - par2 is a call-by-value parameter
    - Changes in par2 do not change the argument variable
- 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

Comparing Argument Mechanism

call-by-value parameter: not changed after function

call-by-reference parameter: value updated after function

```
//DISPLAY 5.6 Comparing Argument Mechanisms
#include <iostream>

void doStuff(int par1Value, int& par2Ref);
//par1Value is a call-by-value formal parameter and
//par2Ref is a call-by-reference formal parameter.

int main( )
{
    using namespace std;
    int n1, n2;

    n1 = 1;
    n2 = 2;
    doStuff(n1, n2);
    cout << "n1 after function call = " << n1 << endl;
    cout << "n2 after function call = " << n2 << endl;
    return 0;
}
```

```
void doStuff(int par1Value, int& par2Ref)
{
    using namespace std;
    par1Value = 111;
    cout << "par1Value in function call = "
         << par1Value << endl;
    par2Ref = 222;
    cout << "par2Ref in function call = "
         << par2Ref << endl;
}
```

Sample Dialogue

```
par1Value in function call = 111
par2Ref in function call = 222
n1 after function call = 1
n2 after function call = 222
```

not affect original value

Inadvertent Local Variables

- If a function is to change the value of an argument
  → it must be a call-by-reference parameter with (&)
- Forgetting the ampersand (&) creates a call-by-value parameter
  - The formal parameter is a local variable that has no effect outside the function
  - Hard to find the error … it looks right!
- Using call-by-reference parameters is convenient but confusing → be careful about its side effects
  - This feature is only available in C++. In older C, it used pointer to pass the reference of a specific variable (introduced in Chapter 9)

Demo: Inadvertent Variables

```
int main( )
{
   int firstNum, secondNum;

   getNumbers(firstNum, secondNum);
   swapValues(firstNum, secondNum);
   showResults(firstNum, secondNum);
   return 0;
}

void swapValues(int variable1, int variable2)
{
   int temp;
   temp = variable1;
   variable1 = variable2;
   variable2 = temp;
}
```

After return, the changes on local variables are lost

Forget the & here: become local variables

Write result to local variable 1

Write result to local variable 2

Sample Dialogue

```
Enter two integers: 5 10
In reverse order the numbers are: 5 10
```


Overview

5.1 More about Functions

5.2 Call-By-Reference Parameters

5.3 Using Procedural Abstraction

5.4 Testing and Debugging

Pre- and Post-conditions

- Pre-conditions and post-conditions are the first step in designing a function
  - Specify what a function should do
  - Minimize design errors
  - Minimize time wasted on writing code that doesn't match
- Pre-condition
  - Function should not be used unless the pre-condition holds
- Post-condition
  - Describes the effect or the return value of the function call
  - Tells what will be true after the function is executed (when the pre-condition holds)

Case Study: Supermarket Pricing

- Problem definition
  - Determine retail price of an item given suitable input
  - 5% markup if item should sell in a week
  - 10% markup if item expected to take more than a week
    - 5% for up to 7 days, changes to 10% at 8 days
- Input
  - The wholesale price and the estimate of days until item sells
- Output
  - The retail price of the item

Pricing: Problem Analysis

- Three main subtasks
  - Input the data
  - Compute the retail price of the item
  - Output the results
- Each task can be implemented with a function
  - Pay special attention to the use of call-by-value and call-by-reference parameters in the following function declarations

Pricing: Input & Output

- void getInput(double& cost, int& turnover);

```
//Precondition: User is ready to enter values correctly
//Postcondition: The value of cost has been set to the
//                wholesale cost of one item. The value of
//                turnover has been set to the expected
//                number of days until the item is sold.
```

- void giveOutput(double cost, int turnover, double price);

```
//Precondition:  cost is the wholesale cost of one item;
//                turnover is the expected time until sale
//                of the item;  price is the retail price of
//                the item.
//Postcondition: The values of cost, turnover, and price
//                been written to the screen.
```

Pricing: Function price

- double price(double cost, int turnover);
  //Precondition: cost is the wholesale cost of one item.
  // turnover is the expected number of days
  // until the item is sold.
  //Postcondition: returns the retail price of the item

- pseudocode for the *price* function:
  - If turnover <= 7 days then
    return (cost + 5% of cost);
    else
    return (cost + 10% of cost);

Pricing: The main Function

- With function declarations, design the main function:

```
int main()
{
   double wholesaleCost, retailPrice;
   int shelftime;

   getInput(wholesaleCost, shelftime);
   retailPrice = price(wholesaleCost, shelftime);
   giveOutput(wholesaleCost, shelftime, retailPrice);
   return 0;
 }
```

- The numeric values in the pseudocode will be represented by constants
  - Const double LOW_MARKUP = 0.05; // 5%
  - Const double HIGH_MARKUP = 0.10; // 10%
  - Const int THRESHOLD = 7; // At 8 days use HIGH_MARKUP

Code for Supermarket Pricing (1/2)

```
//DISPLAY 5.9 Supermarket Pricing
#include <iostream>

const double LOW_MARKUP = 0.05; //5%
const double HIGH_MARKUP = 0.10; //10%
const int THRESHOLD = 7; //Use HIGH_MARKUP
                         //if not expected to sell in 7 days.

void introduction( );
//Postcondition: Show program description on screen.

void getInput(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: cost is the wholesale cost of one item.
//Turnover is the expected number of days until sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//Turnover is the expected number of days until sold.
//Postcondition: Returns the retail price of the item.

void giveOutput(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item;
//turnover is the expected time until sale of the item;
//price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price
//have been written to the screen.
```

```
void introduction( )
{
    using namespace std;
    cout << "This program determines the retail price for\n"
         << "an item at a Quick-Shop supermarket store.\n";
}
void getInput(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold:";
    cin >> turnover;
}
void giveOutput(double cost, int turnover, double price)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Wholesale cost = $" << cost << endl
         << "Expected time until sold = "
         << turnover << " days" << endl
         << "Retail price = $" << price << endl;
}
```

Code for Supermarket Pricing (2/2)

```
double price(double cost, int turnover)
{
    if (turnover <= THRESHOLD)
        return ( cost + (LOW_MARKUP * cost) );
    else
        return ( cost + (HIGH_MARKUP * cost) );
}
```

```
int main( )
{
    double wholesaleCost, retailPrice;
    int shelfTime;
    introduction( );
    getInput(wholesaleCost, shelfTime);
    retailPrice = price(wholesaleCost, shelfTime);
    giveOutput(wholesaleCost, shelfTime, retailPrice);
    return 0;
}
```

Sample Dialogue

```
This program determines the retail price for
an item at a Quick-Shop supermarket store.
Enter the wholesale cost of item: $1.21
Enter the expected number of days until sold: 5
Wholesale cost = $1.21
Expected time until sold = 5 days
Retail price = $1.27
```

- Testing strategies
  - Use data that tests both the high and low markup cases
  - Test boundary conditions: (Ex: 7 days in function price)
    - Test for exactly 7 days, one day more and one day less

Overview

5.1 More about Functions

5.2 Call-By-Reference Parameters

5.3 Using Procedural Abstraction

5.4 Testing and Debugging

Testing and Debugging Functions

- Each function should be tested as a separate unit
- Testing individual functions help finding mistakes
  - Isolate the error sources
- Prepare driver programs to test individual functions
  - Trigger each tested function and observe its outputs
  - You may have to test the function repeatedly with different test cases by using a loop
- Once a function is tested, it can be used in the driver program to test other functions
- Example: function getInput is tested in the driver program (Display 5.10)

Example for Driver Program

```
//DISPLAY 5.10 Driver Program
#include <iostream>

void getInput(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: cost is the wholesale cost of one item.
//Turnover is the expected number of days until sold.

int main( )
{
    using namespace std;
    double wholesaleCost;
    int shelfTime;
    char ans;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        getInput(wholesaleCost, shelfTime);

        cout << "Wholesale cost is now $"
             << wholesaleCost << endl;
        cout << "Days until sold is now "
             << shelfTime << endl;
        cout << "Test again?"
             << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}

void getInput(double& cost, int& turnover)
{
    using namespace std;
    cout << "Enter the wholesale cost of item: $";
    cin >> cost;
    cout << "Enter the expected number of days until sold: ";
    cin >> turnover;
}
```

The function under test

A loop to test the function repeatedly

Sample Dialogue

```
Enter the wholesale cost of item: $123.45
Enter the expected number of days until sold: 67
Wholesale cost is now $123.45
Days until sold is now 67
Test again? (Type y for yes or n for no): y

Enter the wholesale cost of item: $9.05
Enter the expected number of days until sold: 3
Wholesale cost is now $9.05
Days until sold is now 3
Test again? (Type y for yes or n for no): n
```


Stubs

- If the tested function calls other functions that are not yet tested, use a **stub**
- A stub is a simplified version of a function
  - Only provide values for testing rather than perform the intended calculation
  - Make it simple to make sure that they perform correctly
- As an example, function price is used as a stub to test the rest of the supermarket pricing program (Display 5.11)

Example for Program with Stub

```
//DISPLAY 5.11 Program with a Stub
#include <iostream>

void introduction( );
//Postcondition: Show program description on screen.

void getInput(double& cost, int& turnover);
//Precondition: User is ready to enter values correctly.
//Postcondition: cost is the wholesale cost of one item.
//Turnover is the expected number of days until sold.

double price(double cost, int turnover);
//Precondition: cost is the wholesale cost of one item.
//Turnover is the expected number of days until sold.
//Postcondition: Returns the retail price of the item.

void giveOutput(double cost, int turnover, double price);
//Precondition: cost is the wholesale cost of one item;
//turnover is the expected time until sale of the item;
//price is the retail price of the item.
//Postcondition: The values of cost, turnover, and price
//have been written to the screen.

double price(double cost, int turnover)  //This is a stub.
{
   return 9.99; //Not correct, but good enough for test.
}
```

Only replace this function with stub

```
void introduction( )
{
   using namespace std;
   cout << "This program determines the retail price for\n"
        << "an item at a Quick-Shop supermarket store.\n";
}

void getInput(double& cost, int& turnover)
{
   using namespace std;
   cout << "Enter the wholesale cost of item: $";
   cin >> cost;
   cout << "Enter the expected number of days until sold:";
   cin >> turnover;
}

void giveOutput(double cost, int turnover, double price)
{
   using namespace std;
   cout.setf(ios::fixed);
   cout.setf(ios::showpoint);
   cout.precision(2);
   cout << "Wholesale cost = $" << cost << endl
        << "Expected time until sold = "
        << turnover << " days" << endl
        << "Retail price = $" << price << endl;
}
```


General Debugging Guidelines

- Prepare proper inputs to the desired functions
  - Stubs, drivers, test case design
- Keep an open mind
  - Don't assume the bug is in a particular location
- Don't randomly change code without understanding what you are doing
  - You are not always so lucky!!
  - You may generate more errors on some other places
- Discuss the program with someone else
  - It's a good time to review your program flow through discussion

General Debugging Techniques

- Check for common errors, e.g.
  - Local vs. Reference Parameter
  - = instead of ==
- Localize the error
  - This temperature conversion program has a bug
  - Narrow down bug using cout statements
- Use a debugger tool
  - Typically integrated with a development environment
  - Allow you to stop and step through a program line-by-line while inspecting variables
- Use the **assert** macro to test pre or post conditions
  - #include <cassert>
    assert(boolean expression) → Abort the program if the boolean expression is false

The Original Code with Errors

```
1   #include <iostream>
2   using namespace std;
3
4   int main()
5   {
6         double fahrenheit;
7         double celsius;
8
9         cout << "Enter temperature in Fahrenheit." << endl;
10        cin >> fahrenheit;
11        celsius = (5 / 9) * (fahrenheit – 32);
12        cout << "Temperature in Celsius is " << celsius << endl;
13
14        return 0;
15  }
```

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
Temperature in Celsius is 0
```

Debugging with cout Statements

```
//DISPLAY 5.13 Debugging with cout statements
#include <iostream>
using namespace std;

int main()
{
    double fahrenheit;
    double celsius;

    cout << "Enter temperature in Fahrenheit." << endl;
    cin >> fahrenheit;

    // Comment out original line of code but leave it
    // in the program for our reference
    // celsius = (5 / 9) * (fahrenheit - 32);

    // Add cout statements to verify (5 / 9) and
    // (fahrenheit - 32) are computed correctly
    double conversionFactor = 5 / 9;
    double tempFahrenheit = (fahrenheit - 32);

    cout << "fahrenheit - 32 = " << tempFahrenheit << endl;
    cout << "conversionFactor = " << conversionFactor
         << endl;
    celsius = conversionFactor * tempFahrenheit;
    cout << "Temperature in Celsius is " << celsius << endl;

    return 0;
}
```

Use cout statements to export some info. for debugging

Keep the original code as comments for easily recovering after debug

Sample Dialogue

```
Enter temperature in Fahrenheit.
100
fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0
```

Why the factor becomes zero ??

Assert Example

- Denominator should not be zero in Newton's Method

```
// Approximates the square root of n using Newton's Iteration.
// Precondition: n is positive, num_iteration is positive
// Postcondition: returns the square root of n
double newton_sqroot(double n, int num_iterations)
{
        double answer = 1;
        int i = 0;

        assert((n > 0) && (num_iterations > 0));
        while (i < num_iterations)
        {
                answer = 0.5 * (answer + n / answer);
                i++;
        }
        return answer;
}
```