Chapter 5

Functions for All Subtasks
Overview

5.1 void Functions
5.2 Call-By-Reference Parameters
5.3 Using Procedural Abstraction
5.4 Testing and Debugging
5.5 General Debugging Techniques
5.1

void Functions
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
- Example:
  ```cpp
  void show_results(double f_degrees, double c_degrees)
  {
      using namespace std;
      cout << f_degrees
          << " degrees Fahrenheit is equivalent to " << endl
          << c_degrees << " degrees Celsius." << endl;
      return;
  }
  ```

Display 5.1
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:
  
  ```
  show_results(32.5, 0.3);
  
  NOT:   cout « show_results(32.5, 0.3);
  ```
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
Example: Converting Temperatures

- The functions just developed can be used in a program to convert Fahrenheit temperatures to Celsius using the formula

\[ C = \frac{5}{9} (F - 32) \]

- Do you see the integer division problem?
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 Display 5.3, avoids division by zero with a return statement

Display 5.3
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?
5.2

Call-By-Reference Parameters
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
Call-by-Reference Example

- void get_input(double& f_variable)
  
  ```
  using namespace std;
  cout << " Convert a Fahrenheit temperature"
  << " to Celsius.\n"
  << " Enter a temperature in Fahrenheit: ";
  cin >> f_variable;
  ```

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

Display 5.4 (1)
Display 5.4 (2)
Call-By-Reference Details

- 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.

Display 5.5 (1)

Display 5.5 (2)
Call Comparisons
Call By Reference vs Value

- Call-by-reference
  - The function call:
    ```
    void f(int & ref_par);
    ```

- Call-by-value
  - The function call:
    ```
    void f(int var_par);
    ```

---
Memory

<table>
<thead>
<tr>
<th>Name</th>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>1001</td>
<td>34</td>
</tr>
<tr>
<td>initial</td>
<td>1002</td>
<td>A</td>
</tr>
<tr>
<td>hours</td>
<td>1003</td>
<td>23.5</td>
</tr>
<tr>
<td></td>
<td>1004</td>
<td></td>
</tr>
</tbody>
</table>
Example: swap_values

- void swap(int& variable1, int& variable2)
  {
    int temp = variable1;
    variable1 = variable2;
    variable2 = temp;
  }
- If called with swap(first_num, second_num);
  - first_num is substituted for variable1 in the parameter list
  - second_num is substituted for variable2 in the parameter list
  - temp is assigned the value of variable1 (first_num) since the next line will lose the value in first_num
  - variable1 (first_num) is assigned the value in variable2 (second_num)
  - variable2 (second_num) is assigned the original value of variable1 (first_num) which was stored in temp
Mixed Parameter Lists

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

- Example:
  ```c
  void good_stuff(int& par1, int par2, double& par3);
  ```
  - par1 and par3 are call-by-reference formal parameters
    - Changes in par1 and par3 change the argument variable
  - par2 is a call-by-value formal parameter
    - Changes in par2 do not change the argument variable
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

Display 5.6
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!

Display 5.7
Section 5.2 Conclusion

- Can you
  - Write a void-function definition for a function called zero_both 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
5.3

Using Procedural Abstraction
Using Procedural Abstraction

- Functions should be designed so they can be used as black boxes
- To use a function, the declaration and comment should be sufficient
- Programmer should not need to know the details of the function to use it
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: void order(int& n1, int& n2)

```c
{
    if (n1 > n2)
        swap_values(n1, n2);
}
```

- swap_values called if n1 and n2 are not in ascending order
- After the call to order, n1 and n2 are in ascending order
Pre and Postconditions

- **Precondition**
  - States what is assumed to be true when the function is called
    - Function should not be used unless the precondition holds

- **Postcondition**
  - 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
Using preconditions and postconditions the declaration of `swap_values` becomes:

```cpp
void swap_values(int& n1, int& n2);
//Precondition: variable1 and variable 2 have
//been given values
// Postcondition: The values of variable1 and
//variable2 have been interchanged
```
Function celsius

- Preconditions and postconditions make the declaration for celsius:

  double celsius(double farenheit);
  //Precondition:  fahrenheit is a temperature
  // expressed in degrees Fahrenheit
  //Postcondition: Returns the equivalent temperature
  // expressed in degrees Celsius
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
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
Supermarket 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
  - Notice the use of call-by-value and call-by-reference parameters in the following function declarations
void get_input(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.
Supermarket 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
Supermarket Pricing: Function give_output

```
void give_output(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.
```
Supermarket Pricing: The main function

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

```c
int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}
```
Implementations of get_input and give_output are straightforward, so we concentrate on the price function.

- Pseudocode for the price function:
  - If turnover <= 7 days then
    - return (cost + 5% of cost);
  - else
    - return (cost + 10% of cost);
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
Supermarket Pricing: Coding The price Function

- The body of the price function
  ```
  { 
      if (turnover <= THRESHOLD) 
          return ( cost + (LOW_MARKUP * cost) ) ; 
      else 
          return ( cost + (HIGH_MARKUP * cost) ) ; 
  } 
  ```
- See the complete program in

See the complete program in Display 5.9 (1), Display 5.9 (2), Display 5.9 (3)
Supermarket 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
    - In function price, 7 days is a boundary condition
    - Test for exactly 7 days as well as one day more and one day less
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 get_input is tested in the driver program of Display 5.10 (1) and Display 5.10 (2).
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 price is used as a stub to test the rest of the supermarket pricing program in

Display 5.11 (1) and Display 5.11 (2)
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.
Section 5.4 Conclusion

- Can you
  - Describe the fundamental rule for testing functions?
- Describe a driver program?
- Write a driver program to test a function?
- Describe and use a stub?
- Write a stub?
5.5

General Debugging Techniques
General Debugging Techniques

- Stubs, drivers, test cases as described in the previous section
- 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 until the program works
  - This strategy may work for the first few small programs you write but is doomed to failure for any programs of moderate complexity
- Show the program to someone else
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
  - Display 5.12
- Narrow down bug using cout statements
  - Display 5.13
General Debugging Techniques

- Use a debugger
  - Tool typically integrated with a development environment that allows you to stop and step through a program line-by-line while inspecting variables

- The assert macro
  - Can be used to test pre or post conditions
    ```cpp
    #include <cassert>
    assert(boolean expression)
    ```
  - If the boolean is false then the program will abort
Assert Example

- Denominator should not be zero in Newton’s Method

```c
// Approximates the square root of n using Newton's
// Iteration.
// Precondition: n is positive, num_iterations 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;
}
```
Section 5.5 Conclusion

- Can you
  - Recognize common errors?
  - Use the assert macro?
  - Debug a program using cout statements to localize the error?
  - Debug a program using a debugger?
Chapter 5 -- End
Syntax for a \texttt{void} Function Definition

\begin{verbatim}
\texttt{void} Function Declaration
\texttt{void Function\_Name(Parameter\_List);}
Function\_Declaration\_Comment
\end{verbatim}

\begin{verbatim}
\texttt{void} Function Definition
\texttt{void Function\_Name(Parameter\_List)}  \rightarrow \textit{function header}
\{
    \begin{verbatim}
    Declaration\_1
    Declaration\_2 \rightarrow You may intermix the declarations with the executable statements.
    \ldots
    Declaration\_Last
    Executable\_Statement\_1
    Executable\_Statement\_2 \rightarrow May (or may not) include one or more return statements.
    \ldots
    Executable\_Statement\_Last
    \end{verbatim}
\}
\end{verbatim}
void Functions (part 1 of 2)

//Program to convert a Fahrenheit temperature to a Celsius temperature.
#include <iostream>

void initialize_screen();
//Separates current output from
//the output of the previously run program.

double celsius(double fahrenheit);
//Converts a Fahrenheit temperature
//to a Celsius temperature.

void show_results(double f_degrees, double c_degrees);
//Displays output. Assumes that c_degrees
//Celsius is equivalent to f_degrees Fahrenheit.

int main()
{
    using namespace std;
    double f_temperature, c_temperature;

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

    c_temperature = celsius(f_temperature);

    show_results(f_temperature, c_temperature);
    return 0;
}

//Definition uses iostream:
void initialize_screen()
{
    using namespace std;
    cout << endl;
    return; //This return is optional.
}
**void Functions (part 2 of 2)**

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

//Definition uses iostream:
void show_results(double f_degrees, double c_degrees)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);
    cout << f_degrees
        << " degrees Fahrenheit is equivalent to\n"
        << c_degrees << " degrees Celsius.\n";
    return;  // This return is 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.
Display 5.3

**Use of return in a void Function**

**Function Declaration**

```cpp
void ice_cream_division(int number, double total_weight);
//Outputs instructions for dividing total_weight ounces of
//ice cream among number customers.
//If number is 0, nothing is done.
```

**Function Definition**

```cpp
//Definition uses iostream:
void ice_cream_division(int number, double total_weight) {
    using namespace std;
    double portion;
    if (number == 0) {
        return; //If number is 0, then the function execution ends here.
    }
    portion = total_weight/number;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Each one receives "
    << portion << " ounces of ice cream." << endl;
}
```
//Program to demonstrate call-by-reference parameters.
#include <iostream>

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

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

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

int main()
{
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

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

void swap_values(int& variable1, int& variable2)
{
    int temp;
    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
Call-by-Reference Parameters (part 2 of 2)

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

Sample Dialogue

Enter two integers: 5 10
In reverse order the numbers are: 10 5
**DISPLAY 5.5  Behavior of Call-by-Reference Arguments (part 1 of 2)**

**Anatomy of a Function Call from Display 5.4 Using Call-by-Reference Arguments**

0 Assume the variables `first_num` and `second_num` have been assigned the following memory address by the compiler:

- `first_num` → 1010
- `second_num` → 1012

(We do not know what addresses are assigned and the results will not depend on the actual addresses, but this will make the process very concrete and thus perhaps easier to follow.)

1 In the program in Display 5.4, the following function call begins executing:

```cpp
get_numbers(first_num, second_num);
```

2 The function is told to use the memory location of the variable `first_num` in place of the formal parameter `input1` and the memory location of the `second_num` in place of the formal parameter `input2`. The effect is the same as if the function definition were rewritten to the following (which is not legal C++ code, but does have a clear meaning to us):

```cpp
void get_numbers(int& first_num, int& second_num)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num >> second_num;
}
```
DISPLAY 5.5  Behavior of Call-by-Reference Arguments (part 2 of 2)

Anatomy of the Function Call in Display 5.4 (concluded)

Since the variables in locations 1010 and 1012 are first_num and second_num, the effect is thus the same as if the function definition were rewritten to the following:

```cpp
void get_numbers(int& first_num, int& second_num)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

3 The body of the function is executed. The effect is the same as if the following were executed:

```cpp
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> first_num
        >> second_num;
}
```

4 When the cin statement is executed, the values of the variables first_num and second_num are set to the values typed in at the keyboard. (If the dialogue is as shown in Display 5.4, then the value of first_num is set to 5 and the value of second_num is set to 10.)

5 When the function call ends, the variables first_num and second_num retain the values that they were given by the cin statement in the function body. (If the dialogue is as shown in Display 5.4, then the value of first_num is 5 and the value of second_num is 10 at the end of the function call.)
Comparing Argument Mechanisms

//Illustrates the difference between a call-by-value //parameter and a call-by-reference parameter. #include <iostream>

void do_stuff(int par1_value, int& par2_ref);
//par1_value is a call-by-value formal parameter and //par2_ref is a call-by-reference formal parameter.

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

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

void do_stuff(int par1_value, int& par2_ref)
{
    using namespace std;
    par1_value = 111;
    cout << "par1_value in function call = "
         << par1_value << endl;
    par2_ref = 222;
    cout << "par2_ref in function call = "
         << par2_ref << endl;
}

Sample Dialogue

par1_value in function call = 111
par2_ref in function call = 222
n1 after function call = 1
n2 after function call = 222
Inadvertent Local Variable

// Program to demonstrate call-by-reference parameters.
#include <iostream>

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

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

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

int main()
{
    using namespace std;
    int first_num, second_num;

    get_numbers(first_num, second_num);
    swap_values(first_num, second_num);
    show_results(first_num, second_num);
    return 0;
}

void swap_values(int variable1, int variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

<The definitions of get_numbers and show_results are the same as in Display 4.4.>

Sample Dialogue

Enter two integers: 5 10
In reverse order the numbers are: 5 10
//Program to demonstrate a function calling another function.
#include <iostream>

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

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

void order(int& n1, int& n2);
//Orders the numbers in the variables n1 and n2
//so that after the function call n1 <= n2.

void give_results(int output1, int output2);
//Outputs the values in output1 and output2.
//Assumes that output1 <= output2

int main()
{
    int first_num, second_num;

    get_input(first_num, second_num);
    order(first_num, second_num);
    give_results(first_num, second_num);
    return 0;
}

//Uses iostream:
void get_input(int& input1, int& input2)
{
    using namespace std;
    cout << "Enter two integers: ";
    cin >> input1 >> input2;
}
Function Calling Another Function (part 2 of 2)

```c
void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}

void order(int& n1, int& n2)
{
    if (n1 > n2)
        swap_values(n1, n2);
}

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

Sample Dialogue

Enter two integers: 10 5
In increasing order the numbers are: 5 10
Supermarket Pricing  (part 1 of 3)

// Determines the retail price of an item according to
// the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

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

void introduction();
// Postcondition: Description of program is written on the screen.

void get_input(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.

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

void give_output(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.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}
//Uses iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n" << "an item at a Quick-Shop supermarket store.\n";
}

//Uses iostream:
void get_input(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;
}

//Uses iostream:
void give_output(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;
}

//Uses defined constants LOW_MARKUP, HIGH_MARKUP, and THRESHOLD:
double price(double cost, int turnover)
{
    if (turnover <= THRESHOLD)
    {
        return (cost + (LOW_MARKUP * cost));
    } else
    {
        return (cost + (HIGH_MARKUP * cost));
    }
}
Supermarket Pricing (part 3 of 3)

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
Driver Program (part 1 of 2)

//Driver program for the function get_input.
#include <iostream>

void get_input(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.

int main()
{
    using namespace std;
    double wholesale_cost;
    int shelf_time;
    char ans;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        get_input(wholesale_cost, shelf_time);

        cout << "Wholesale cost is now $"
        << wholesale_cost << endl;
        cout << "Days until sold is now "
        << shelf_time << endl;

        cout << "Test again?"
        << " (Type y for yes or n for no): ";
        cin >> ans;
        cout << endl;
    } while (ans == 'y' || ans == 'Y');

    return 0;
}
Driver Program (part 2 of 2)

```cpp
//Uses iostream:
void get_input(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;
}
```

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
Program with a Stub (part 1 of 2)

// Determines the retail price of an item according to
// the pricing policies of the Quick-Shop supermarket chain.
#include <iostream>

void introduction();
// Postcondition: Description of program is written on the screen.

void get_input(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.

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

void give_output(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.

int main()
{
    double wholesale_cost, retail_price;
    int shelf_time;

    introduction();
    get_input(wholesale_cost, shelf_time);
    retail_price = price(wholesale_cost, shelf_time);
    give_output(wholesale_cost, shelf_time, retail_price);
    return 0;
}

// Uses iostream:
void introduction()
{
    using namespace std;
    cout << "This program determines the retail price for\n" << "an item at a Quick-Shop supermarket store.\n";
Program with a Stub (part 2 of 2)

//Uses iostream:
void get_input(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;
}

//Uses iostream:
void give_output(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;
}

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

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 = $9.99
```c++
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   return 0;
14 }
```

**Sample Dialogue**

Enter temperature in Fahrenheit.

100

Temperature in Celsius is 0
```cpp
#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;
}

---

**Sample Dialogue**

Enter temperature in Fahrenheit.

100

fahrenheit - 32 = 68
conversionFactor = 0
Temperature in Celsius is 0