3 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.

3.1 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:

```
void order(int& n1, int& n2)
{
        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

3.2 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

3.2.1 swap_values

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

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

3.2.2 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
```

3.2.3 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

3.3 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

3.3.1 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

3.3.2 Supermarket Pricing:Function get_input

```
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.
```

3.3.3 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
```

3.3.4 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.
```

3.3.5 Supermarket Pricing:The main function

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

```
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;
}
```

3.3.6 Supermarket Pricing:Algorithm Design – price

- 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);

3.3.7 Supermarket Pricing:Constants for The price Function

- 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
```

3.3.8 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

3.3.9 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

4 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
- Use IDE's debug tools.