Functions for All Subtasks

Chapter 5
In this chapter, you will learn about:

- Void Functions
- Call-By-Reference Parameters
- Using Procedural Abstraction
- Testing and Debugging
- General Debugging Techniques
void Functions

- Recall a Function implements subtask
- In General a Subtask can return:
  - Only one value --- See Chapter 4
  - No value (e.g. display GUI on screen)
    or
  - Multiple values

- How do you implement a subtask that returns no value?
  - Use void function
**void Function**

**Definition**

```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;
}
```

Example

No value is returned by `show_results`

No expression in return statement
void Function Syntax

Declaration

Insert Function Declaration Comments here

Function Definition

May (or may not) include one or more return statements. return is optional
void Function

calls

Consider:

```cpp
void show_results(double f_degrees, double c_degrees);
……
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;
}
```

Example call

```
show_result(32.5, 0.3);
```

What is wrong with this statement?

```
cout << show_result(325, 0.3)
```
void Function – Program Analysis-I

```c
void initialize_screen( );
// Separates current output from the output of previous run
double celsius (double fahrenheit);
// Converts Fahrenheit to Celsius
void_show_results (double f_degrees, double c_degrees);
// Displays output, Assume c_degrees is equivalent to f_degrees

int main( )  
{
    using namespace std;
    double f_temp, c_temp;
    initialize_screen( );
    cout << "Converts Fahrenheit to Celsius\n"
        << "Enter Fahrenheit temp"
        << f_temp;
    c_temp = celsius(f_temp);
    show_results (f_temp, c_temp);
    return;
}

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

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

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 equals"
        << c_degrees<< " Celsius\n";
    return;
}
```
```cpp
int main( )
{
    using namespace std;
    double f_temp, c_temp;
    initialize_screen( );
    cout << "Converts Fahrenheit to Celsius\n"
         << "Enter Fahrenheit temp"
         << endl;
    cin >> f_temp;
    c_temp = celsius(f_temp);
    show_results(f_temp, c_temp);
    return;
}

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

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

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 equals"
         << c_degrees<< " Celsius\n";
    return;
}
```
void Function

Use of return Statement

- The return statement is optional in a void function definition.
- Other times (when embedded in a conditional statement), it could prevent errors such as division by zero.
Call-by-Reference Parameters
Changing Multiple Values

- Allows a Function to change Variables used in a function call

  - Variables to be changed are uniquely identified in the function declaration and definition statements

  - Pertinent Arguments in the calling function must be variables not values (e.g., numbers)
Function Declaration
Call-by-Reference Parameter

Example

```c
void get_input(double & f_variable);
```

Identifies f_variable as a call-by-reference parameter
Function Definition

Call-by-Reference Parameter

Example

```cpp
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;
}
```

How do you implement a subtask that changes one or more values?
Function Calls

Call-by-Reference Parameter

Consider

```cpp
void get_input(double& f_variable);

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;
}
```

**Example call**

`get_input (temp)`

**Memory location of temp is plugged into f_variable. Not its contents**
Call-by-Reference – Program Analysis-I

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

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

void show_results (int output1, int output2);
// Displays output1 and output2 sequentially

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;
}

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

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

void show_results (int output1, int output2)
{
    using namespace std;
    cout << "In reverse order the numbers are: " << output1  << "  " << output2 << endl;
    return;
}
```
Call-by-Reference vs. Call-by-Value Comparison

Assume Memory Locations:

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>second_num</td>
<td>1001</td>
<td>10</td>
</tr>
<tr>
<td>last_num</td>
<td>1002</td>
<td>100</td>
</tr>
<tr>
<td>first_num</td>
<td>1006</td>
<td>20</td>
</tr>
</tbody>
</table>

Call-by-Reference Parameter:

```c
void get_number (int& input1);
get_numbers(first_num);
```

Call-by-Value Parameter:

```c
void get_number (int input1);
get_numbers(first_num);
```
Mixed Parameters
Reference & Value Parameter

- You can mix Call-by-Reference and Call-by-Value in the same function

Example:

```c
void cash (double& principal, double& time, double rate)
```

- `principal` and `time` are call-by-reference formal parameter
  - Changes in `principal` and `time` change the argument variables
- `rate` is a call-by-value formal parameter
  - Changes in `rate` do not change the argument variable
How do you decide when to use a call-by-reference or call-by-value formal parameter?

- Does the function need to change the value of the variable used as an argument?
  - If yes then
    - use a call-by-reference formal parameter
  - Otherwise
    - use a call-by-value formal parameter
Using Procedural Abstraction

Functions Calling Functions

- A Function **body** may contain a call to another function provided
  - The called function is already declared

- Do not pass a function call as an argument to another function *(Follow the KIS principle)*

- You cannot define a function within the body of another function – **Why?**
Functions Calling Functions

```c
void order(int& input1, int& input2);
// Orders the numbers in variables input1 and input2

void swap_values(int& variable1, int& variable2);
// interchange the values of variables1 and varaibles2

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

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

This function order( ) calls another function
How to write Function Declaration Comment?

Method 1: Refer to Lectures notes in Chap 4

Alternative Method:
- Break Comments into two parts
  - **Preconditions** – What is assumed to be true when function is called
  - **Postconditions** — The effect of calling the function

Let’s examine both concept for practical subtasks - real systems
double celsius(double farenheit);
    //Precondition:  fahrenheit is a temperature expressed
    //degrees Fahrenheit
    //Postcondition:  Returns the equivalent temperature
    //expressed in degrees Celsius

Refer to slide 29 in Chapter 4 lectures notes
Preconditions and Postconditions

Square root function

- Give a precondition and a postcondition for the predefined function `sqrt`, which returns the square root of its argument:
  - Assume function declaration:
    ```
    double sqrt (double n);
    ```
Preconditions and Postconditions

Square root function

- Assuming function declaration:
  ```
  double sqrt (double n);
  //Precondition: n >= 0
  //Postcondition: Returns the square root of n
  ```
Testing and Debugging Functions

Unit Testing

- Unit Testing
  - Test each function as a separate unit from the rest of the program
    - It’s a lot easier to debug your program suite
- How do you test individual functions?
  - Create a **driver** program to test each function
    - Provide the means to check, inspect value(s) returned/changed by the function for different scenarios
  - After a function is unit tested, you can use it as a driver program for untested functions

A driver program is written for the sole purpose of testing a function
Testing and Debugging Functions

Stubs

- How do you test a function that makes a call to other untested function(s)?
- Create a **Stub** program for the other untested function
- The Stub program should simply return values that you know are correct

**A Stub** is a simplified version of a function, which is used in place of the untested function so that the called functions can be tested
Write a stub for the function with the following declaration:

double rain_prob (double pressure, double humidity, double temp);
// Precondition: pressure is the baramotic pressure in
// inches of mercury,
// humidity is the relative humidity as a percent, and
// temp is the temperature in degrees Fahrenheit
// Returns: the probability of rain, which is a number
// between 0 and 1. 0 means no chance of rain, 1 means rain is 100% certain
Testing and Debugging Functions

Example: Stub

double abc(double p, double q, double r);
int main ( )
{
    declarations_1
    double result;
    declarations_last

    result = abc( x, y, z);
    - - - - - - - - - - - - -
}
double abc(double x, double y, double z)
{
    Local variable Declarations ..... 
    double cahnce;
    computations ..... 
    chance = rain_prob (double pressure, double humidity, double temp);
    computations ..... 
}

We need a stub for rain_prob( ) in order to unit test abc ( )
Testing and Debugging Functions
Example: Stubs

```c
double rain_prob (double pressure, double humidity, double temp)
{
    return 0;  // Insert comment here
}
```

```c
double rain_prob (double pressure, double humidity, double temp)
{
    return 0.5;  // insert comment here
}
```

```c
double rain_prob (double pressure, double humidity, double temp)
{
    return -1;  //Insert comment here
}
```

```c
double rain_prob (double pressure, double humidity, double temp)
{
    return 100;  //Insert comment here
}
```
Let’s summarize

- All subtasks can be implemented as functions
- In call-by-value substitution, the value of an argument is plugged into formal parameter
- In a call-by-reference substitution, the argument should be a variable, and the variable (i.e., its address) is plugged into the formal parameter
- Function declaration comments can be stated as precondition and postcondition
- Developer performs unit testing, while Verification Engineer performs Systems/Inter-Systems Testing
- A driver program has one task - test a function
- A stub is used in place of a function that is untested