Hands-On Activity 5.2	
Structure	
Course Code: CPE007	**Program:** Computer Engineering
Course Title: Programming Logic and Design	**Date Performed: 09/30/2025**
Section: CPE11S1	**Date Submitted: 10/04/2025**
Name(s): James Daniel M. Verano	**Instructor: Engr. Jimlord M. Quejado**
6. Output	

⇒ **[Example 1 Analysis]:**

In the sample code provided, introducing the usage of the structure member and pointer operators, the code snippet introduces us to different ways on how to print out the values in various implementations/declarations. Firstly, let's start with the upper line codes. As we can see outside the main function, we have created a structure with the name of ***"Card"***. Which means, we are introducing the ***struct function*** procedure of printing out our desired output. Inside the struct function, we have declared 2 variable names with a return type of "***string***",

represented as:
struct Card {
string face;
string suit;
}

Since we now have a declared variable that has no declared values, now we will be proceeding inside the main block functions. As we can see, we have another declared variables, the "***Card a***" and "***Card* aPtr***". the special character "*****" Looks familiar right? Yes. That's the dereference pointer. That's the second procedure on how to take the variable's value in correlation with the structure function. Before we proceed on that part, let us first try the first method, which is the dot operator. One of the basic methods of taking assigned values inside the declared variables that are inside the structure functions. The declared variable "***Card a***" is the structure's variable name in terms of declaring the assigned values inside the "***Card***" Structure function.

For Visuals:
Card = String face; and string suit;
Card a = directing the string values directory to a.

which means:
a = string face; and string suit;

Using the dot operator:
a.face = Ace (Assigned value to .face)
" of "
a.suit = Spades (Assigned value to .suit)
Result: Ace of Spades

Now that we have introduced the first part of how to print out our desired output in the dot operator method, let's proceed to another way, where it's more focused on the exact declaration of "***aPtr***". The "***aPtr***" variable is directly declared "***&a***" which defines the address of a. And, back to the visuals above, we can see that, a is declared to the dot operator of the string face and suit, which means that we are taking the address of the string face and suit. Moving forward, we now know that "***aPtr***" is equal to the address of the strings, thus we just need to print it

out using the arrow operator. The ***arrow operator*** Requires a specific structure for it to work, it's used to access members of a struct and pointers struct most of the time. Which means, In the line code 26, we can see that, in the printing of the variable, we are pointing to the struct variable of face and suit, which defines that, we are pointing to the address of a within the specific variable name of the string.

For Visuals:
aPtr = &a ⇒ Address of a, a is declared to = .face and .suit with a declared string value.
[aPtr value = address of a]
aPtr->face ⇒ Directly pointing to the address of a with a variable named [face]
Address of ⇒[a].[face] ⇐named variable [face = Ace] ⇐ Declared string value.

For the last part, we are going to combine the dereference pointers and dot operators. Short recap, the dereference pointer means that you are pointing to the value of the address you declared inside the variable. In this example. in the line 29, we can see that we printed out the values with the structure of ***(*aPtr).face***, which interprets as, pointing to the value of the "***aPtr***" and using that value to locate the address of the value stored in the struct function.

For Visuals:
Pointing to the value of the address declared ⇒(*aPtr).face ⇐Struct variable with a declared value in the address of [a]
Value of the address declared ⇒[&a] = a.face ⇒ Ace

⇒ [Example 2 Analysis]:

With the code snippet provided, this is a basic representation on how to declare different data types variables in the struct function, and use them with ease in declaring them with values. Commonly, in the upper line codes, specifically, 6 - 11, are all only the struct function named books with declared specified data types variables. This is to prevent redundancy in rewriting data types and represents a cleaner visual design.

In the line code 15-16, this implies that, we are declaring 2 variables named **Book1 & Book2**, with a struct function as its data type, which defines that the variables declared in the struct function are needed to declare values in accordance with using the same variable name.

For Visuals:
Struct Function with declared specified data types variables ⇒ [Books] Book1; ⇐directory of values declared.
Implies to the declared value of the variable in book1 ⇒ Book1.title = "Value"
Implies to the declared value of the variable in book2 ⇒ Book2.title = "Value"

It is much easier and cleaner if you're using the struct function since you no longer need to repeat another variable type with the same data type over and over again. With this function all you have to do is to declare another variable with the struct function as its data type, and you will just need to declare values in each of the declared variables that are inside the struct function, WHICH ONLY IMPLIES to the variable you declared it to.

For Visuals:

Book1.title = "Value"
Book2.title = "Value2"
⇒ Has the same variable name yet, different values declared.

⇒ **[Example 3 Analysis]:**

In the code snippet in the example 3, we can see that we are introduced to "***Void***". "***Void***" is a return type, using void means that you are declaring a function that does not return a value. In the line code 6-11, We can see that we declared a structure that consists of our necessity for printing out our stored values in terms of void. in the line code 14, which is the "***printBook(Books book)***", is a function declaration wherein the structure is passed by value. It just passes the actual values to the variables.

In the main block area, line code 18-19, are structure variable declaration, which structurizes the variables with the struct declared variables,

For Visuals:
Books book1;
==
book1.title
book1. author
….

In the line code 22 - 31, Are just the declarations of values, it is just putting values into the variables.

in the line 34 - 36, is the printing of the void variable. Which is linked to the void after the main block function, the line code 42 - 47. The "***void printBook(Books book)***" defines the function of printing the actual variables of variable "***Books***" that was passed through references and linked to the variable "***book***" the variable "***book***" holds the variable that has the actual values of the variables "***Books book1***", and "***Books book2***" Which when we print the line code 34-36, it prints the actual values that are linked to variable "***book1***" and "***book2***".

7. Supplementary Activity

⇒ **[Supplementary 1]:**

[Code of the program]:

```
1   #include <iostream>
2   using namespace std;
3
4   struct rectangle {
5       float length;
6       float width;
7       float area;
8       float perimeter;
9       string design;
10  };
11
12  void compute (rectangle recV, float& area, float& perimeter) {
13      area = recV.length * recV.width;
14      perimeter = 2 * (recV.length + recV.width);
15  };
16
17  int main() {
18
19      rectangle recV;
20      rectangle struc;
21      struc.design = "----------------------------------------\n";
22
```

```
23      cout << "Enter the length of the rectangle ==> ";
24      cin >> recV.length;
25      cout << struc.design;
26      cout << "Enter the width of the rectangle ==>  ";
27      cin >> recV.width;
28      cout << struc.design;
29
30      float area, perimeter;
31
32      compute(recV, area, perimeter);
33
34      cout << " |\t" << "Area of the rectangle\t  : " << area << "\t| " << endl;
35      cout << " |\t" << "Perimeter of the rectangle: " << perimeter << " \t| " << endl;
36
37      return 0;
38  }
```

[Output/s of the program]:

```
Enter the length of the rectangle ==> 25
----------------------------------------
Enter the width of the rectangle ==>  45
----------------------------------------
 |      Area of the rectangle     : 1125        |
 |      Perimeter of the rectangle: 140         |

--------------------------------
Process exited after 11.01 seconds with return value 0
Press any key to continue . . .
```

figure 1.1: Integer Value Input

```
Enter the length of the rectangle ==> 25.5
----------------------------------------
Enter the width of the rectangle ==>  47.8
----------------------------------------
 |      Area of the rectangle     : 1218.9      |
 |      Perimeter of the rectangle: 146.6       |

--------------------------------
Process exited after 23.63 seconds with return value 0
Press any key to continue . . .
```

figure: 2.1: Float Value Input

[Analysis of the code]

⇒ In this supplementary activity 1, we are tasked to create a program that stores the rectangle's length and width values, and create a function that accepts the structure as an argument for the computation of the area and perimeter of the rectangle. With this instruction, we are visualized to have 4 objectives.

1. Create a structure to store the length and width values of a rectangle.
2. Create a function that accepts the structure as an argument.
3. Create a computation for area and perimeter using the function.
4. Print out the area and perimeter of the rectangle.

Let's start with the first objective, declaration of variables to store the length and width values. As represented in the code of the program above, line 4-10 is the structure for our storing values variable. Inside the structure are the declaration of all needed variables. We use float data type in all number related variables such as the length, width, area, and perimeter since they can possibly hold a decimal numerical value. The string is only for the structural design for an organized format of the outputs.

Since now we have our structure that stores our needed values for our rectangle, we will now have to create a function that accepts the structure as an argument. Let us introduce to you "***Void***". "***Void***" is a return type, using void means that you are declaring a function that does not return a value. Which means, what only happens in the void function is about modifying the functions and parameters that are passed by reference. With the line code above, line 12-15 to be exact, we can see that we are going to initialize functions that will compute the value of area and perimeter, using passed reference. What is a "***passed reference***"? Writing data types with the character "**&**" implies that you are giving the actual value of the variable, not just a copy, but exactly the exact value of the variable.

For visuals:
declared ⇒ float& area
Void processes: area = recV.length * recV.width ⇒ [Processes an actual value for area]

in the main block function**
compute(area);
cout << area ⇒ Prints out the actual value for area.

Now that we have created a function that accepts the structure as an argument, the next objective is to create a computation for area and perimeter. I have already introduced the computation of the area, which was the function notated as "***area = recV.length * recV.width***" which is expressed as rectangle value length * rectangle value width = area. Now, for the perimeter, we will be following this formula of: 2 * (length + width). Which in terms of the code of the program above, line 14, we have declared it as "***perimeter = 2* (recV.length + recV.width)***".

The last objective is the processing of getting the values to store for the rectangle's length and width values, and the printing of the area and perimeter of a rectangle. In the line code 19 - 21, is the declaration of the structure to access the declared variables to be used for the storing of values and also structure design. line code 23 - 28, is the prompt of requesting a value to the recipient, line 30 - 32, is the declaration of the data type for the variables that have actual values, and on line 32, is the function that will compute the inputted values and relay them to area and perimeter. Lastly, line 34 - 35, is the printing out of the computed value of area and perimeter of the rectangle of the inputted value by the recipient.

⇒[Supplementary 2]:

[Code of the program]:

```cpp
#include <iostream>
using namespace std;

struct structure{
    string design;
};

bool multiple(int numInteger, int integerX) {
    if (integerX == 0) {
        cout << "Cannot divide by zero." << endl;
        return false;
    }
    return (numInteger % integerX == 0);
}

int main() {

    int numInteger, integerX;
    structure struc;
    struc.design = "--------------------------------\n";

    cout << "     Enter an Integer : ";
    cin >> numInteger;
    cout << struc.design;
    cout << " Enter the Value of X : ";
    cin >> integerX;
    cout << struc.design;

    if(multiple(numInteger, integerX)) {
        cout << "  | " << numInteger << " is a multiple of " << integerX << ". | "
        << endl;
    }
    else {
        cout << "  | " << numInteger << " is NOT a multiple of " << integerX
        << ". | " << endl;
    }

    return 0;
}
```

[Output/s of the program]:

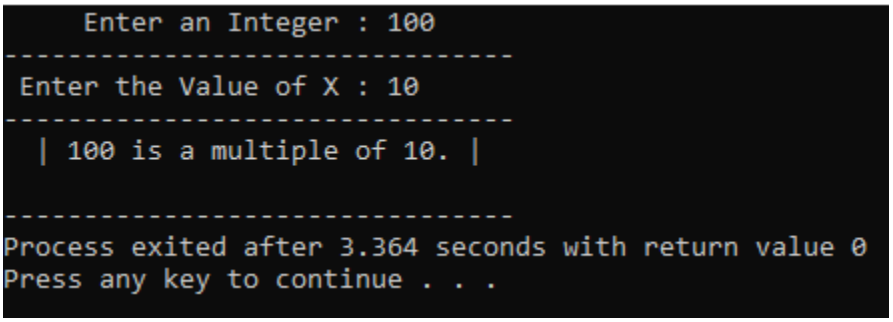

figure 2.1: Integer is a multiple of X

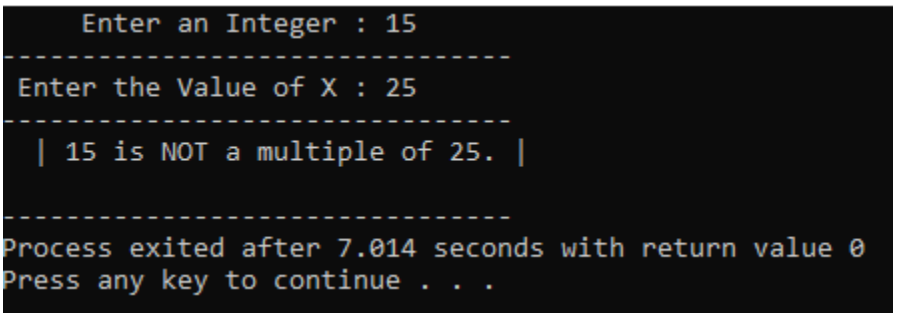

figure 2.2: Integer is NOT a multiple of X

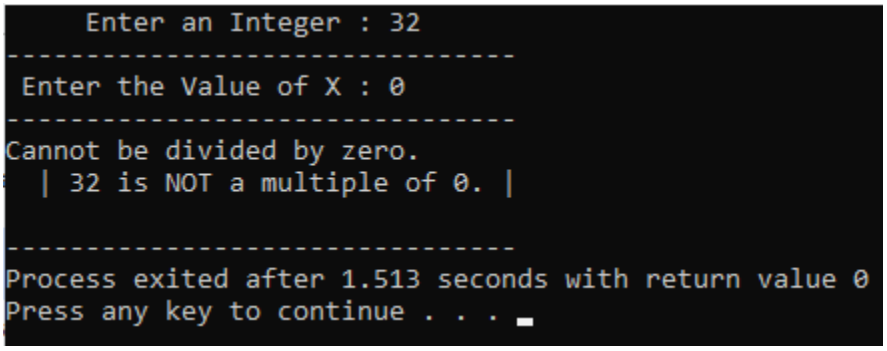

figure 2.3: if X is zero

[Analysis of the code]

Supplementary Activity 2, In this we are tasked to create a program that has a function multiple that determines if the integer entered from a keyboard is a multiple of some integer X. To summarize it into objectives, we will have exactly 3 objectives.

1. Create a function multiple that processes and identifies if integer is a multiple of the integer X.
2. Create a code that takes the recipient integer X and integer value to be identified with regards whether it is a multiple.
3. Create an if-else statement for IF function multiple has met its standard expression to identify whether the inputted integer value is a multiple of the integer X.

Let's start with the first objective. In the provided code in the program above, we can see that in the line 8 - 14 is our function multiple. Here, we are introduced to bool. Bool is a data type that represents a boolean value. A boolean value can only be either true or false. We used bool because it fits to our necessity in identifying whether the integer is a multiple to X. Which can only be answered by true or false. Below our function multiple, is a function that if X, or in this program named as, "***IntegerX***" is equal to "**0**" it will return false. If "***return (numInteger % integerX ==0)***" has been met, it will return true.

For Visuals:
X = 0 ⇒ False
Integer = 15 X = 10 ⇒ False (It has a remainder of 5) 15/10 = 1%5
Integer = 100 X = 10 ⇒ True (It does not have a remainder) 100/10 = 10

Now that we have achieved our first objective, let us proceed to the second objective, which is to create a code that takes the recipient values for X and integer for the identification whether integer is a multiple of X. In the main block code, lines 18 - 20 are our declaration of variables for the storing of values, and some variable declarations are just for the design structure. in the line 22-27, is our prompt for our recipients inputting values. Nothing new, All are just from the basic coding of c++.

Now for the last objective, which is to create an if-else statement for whether the function multiple has met a certain standard expression to identify whether the inputted integer value is a multiple of integer X. the function, "***if (multiple(numInteger, integerX))***" Implies to the process if it is true or false with regards to what values was inputted by our recipient. IF the function multiple has not met the if statement of X == 0, and returns true, then the function will process the if function. But if the function multiple has not been met, returned false, OR the X value is equal to 0, the else statement function will be implemented.

8. Conclusion

Guide in creating a conclusion:

⇒ With the given activities,I was able to learn quite a lot in critical analysis and deepen my connection with the structure and different types of new codes, such as the void, booleans, and passing reference. I was able to fully express and interpret the procedures of each code snippet and also supplementary objectives, that I was able to fully execute and create. With the analysis of all activities, specially the supplementary activity, I have fully recognized my ability to analyze deeply to the code I have compiled and functionalized to achieve the desired outcomes. The analysis was quite a hassle in terms of trying to adapt in the new way of explaining for a more precise and clear introduction and analysis of each line code functions. I think I did very well, explained it very well, and got the topic in the average level. I was able to adapt to some parts of the processing and creation of programs. If ever I need to improvise something, I think it would be probably in the creation of the coding program, where maybe I could find a more organized and easier alternatives in terms of declarations using structures.