THE UNIVERSITY COLLEGE
OF THE CARIBOO

Computing Science – **COMP213**

Lab Exercises #7 – C - *Structures*

Marks: 5 marks | Due: November 5th, 2004 (in lecture)

Introduction

Most high-level languages have a technique of encapsulating a *related set of data*. Depending on the given language, it can be called: aggregate type, abstract data type, structure, record, or class (in OOP).

The purpose of this construct is to allow various sub-types to be defined as a single entity; much like a single person containing sub-definitions such as name, age, etc.

Although they lack the flexibility and extensibility of what is available in higher-level, OOP languages, structures in C still have their merits. With the low-level access possibilities of C, the structures allow for complex data manipulation as well as encapsulation.

The exercises below provide you an opportunity to experiment with the syntax and concepts.

Compilers and Tools

Use the techniques and resources from the previous labs to complete the following exercises. Also, you are encouraged to use the on-line references for examples and details not expressed in lecture.

Exercises / Programming Problems

1. You have already seen how the GNU C-compiler establishes the memory order of variable declarations in a program. With the introduction of variables members within structures, do the same rules apply? Test it!

   Write a program,

   - create a structure that holds a double, an integer, and a char
   - in the **main()** declare a double variable, an integer variable, and a char variable, as well as a variable of the new structure type, followed by another integer variable
   - display the addresses of the above 5 variables, in the exact order they were declared; when it comes to the structure variable, show the addresses of the structure variable followed by it's members, in the proper order
   - showing the addresses of members is no different than with single variables,

     `&var.member` or `&(var.member)`

   What is your observation about the order of the members in memory?
   (*hint: the new structure is a single data type*)

2. Write a simple program that uses a structure containing 4 double values: *a*, *b*, *c*, and *average*.
   The program loops until the user selects 0, 0, 0 as the three input values.
   Each time through the loop, the program accepts three double values from the user, stores them into a variable of the type (declared outside the loop), as well as calculate and store the average.
   The 3 values and the average are displayed back to the user.

3. Modify the program in 2., such that the structure variable is a pointer using a dynamic memory location.

4. Write a program that uses a **union** structure to create a password verifier. The purpose of the program is either generate the **short** integer that corresponds to 2 characters provided by the user, or accept a short and 2 characters and verify that they correspond. Both options are available to the user.

   The union structure is,

```
typedef union
{
   short v;      // variable to hold numeric value
   char  c[2];   // character array at same memory location
} verifytype;
```

5. Write a program that asks the user for ***n*** value sets, with each set containing an integer and a character. From this data, the program displays a bar-graph showing bars of the lengths drawn with the characters.

   For example,

```
Please enter the number of sets: 4

Set 1: 3 @
Set 2: 5 *
Set 3: 1 #
Set 4: 2 ,

@@@
*****
#
,,
```

   In the program,

   - create a structure that holds an integer and a character—this is the set
   - write a **void** function, called **showset()**, that has a single parameter: a variable of the set type, and the function performs the actual showing of that particular set
   - in the **main()**, create an array of the set type, based in the value of ***n*** provided by the user (you may have to research the topic: *using arrays of structures*)
   - a call the **showset()** function is made for each element in the array

6. Use the following definitions of a byte-structure and a union that allows a character and byte-structure to exchange data,

```
typedef struct
{
   unsigned int b0 : 1;   // lsb
   unsigned int b1 : 1;
   unsigned int b2 : 1;
   unsigned int b3 : 1;
   unsigned int b4 : 1;
   unsigned int b5 : 1;
   unsigned int b6 : 1;
   unsigned int b7 : 1;   // msb
} bytetype;
```

```
typedef union
{
   char c;
   bytetype b;
} controltype;
```

and the method **displaybyte()** that displays an argument of **bytetype** as a nice sequence of bits.

```
// display_byte() - displays a bytetype as a string of
//                  bits: msb -> lsb
void display_byte (bytetype word)
{
    printf("%d%d%d%d%d%d%d%d",
          word.b7,
          word.b6,
          word.b5,
          word.b4,
          word.b3,
          word.b2,
          word.b1,
          word.b0);
}// end of display_byte()
```

In a program,

- declare a few variables of type **controltype**
- while treating the member c of the controltype variable as a character, store a user's input symbol to the member; use the function **display_byte()** to display the binary form of the character;
  and
  hard code (do not ask the user) a sequence of bits and verify that sequence represents the intended character
- treat the member c of the controltype variable as an unsigned integer, store a user's input value to the member; use the function **display_byte()** to display the binary form of the number;
  and
  hard code a sequence of bits and verify that sequence represents the intended number
- *note: displayable ASCII characters are 33 to 127, with 128-255 being extended character that may not be on the keyboard*

Conclusion

Submit the well-documented source code for the program: #4, #6. Include output captures.

Attach a cover page, with: *course number, exercises/assignment title, your name, date of submission*.