Lab 2

Objectives: This lab is simply to help you to recall some basic program techniques that you have learnt in C-Program.

Review on C-Program

The very first program, "Hello World".

```
#include <stdio.h>

int main()
{
    printf("\nHello World\n\n");

}
```

Declaration of Variables

Variable must be declared before it could be used in a program and has a specific type. Usually, the declaration appears at the beginning of a block.

Common Data Type:

Type	Size	Value Range
`char`	1 byte	[-128 to 127] or [0 to 255]
`int`	2 or 4 bytes	[-32,768 to 32767] or [-2,147,483,648 to 2,147,483,647]
`unsigned int`	2 or 4 bytes	[0 to 65,535] or [0 to 4,294,967,295]
`long`	4 bytes	[-2,147,483,648 to 2,147,483,647]
`float`	4 bytes	[1.2E-38 to 3.4E+38] (6 decimal places)
`double`	8 bytes	[2.3E-308 to 1.7E+308] (15 decimal places)

Examples:

```
int a = 1, b = 5;
float f;
```

However, the following declaration method may not work properly with C in some machine.

```
for (int i=0; i<10; i++)
...
```

Control Flow Statements

C provides two styles of flow control:

- Branching
- Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:

Branching is so called because the program chooses to follow one branch or another.

`if` *statement*

It takes an expression in parenthesis and a statement or block of statements. If the expression is true then the statement or block of statements get executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaluated values is non-zero.

"`if`" statements take the following forms:

```
if (expression)
  statement;
```

or

```
if (expression)
  {
    Block of statements;
  }
```

or

```
if (expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }
```

or

```
if (expression)
  {
    Block of statements;
  }
else if(expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }
```

`? :` *Operator*

The `? :` operator is just like an `if ... else` statement except that because it is an operator you can use it within expressions.

`? :` is a ternary operator in that it takes three values, this is the only ternary operator C has.

`? :` takes the following form:

```
#include <stdio.h>

int main(void)
{
   int a , b;

   a = 10;
```

```
   printf( "Value of b is %d\n", (a == 1)  ? 20: 30 );
   printf( "Value of b is %d\n", (a == 10) ? 20: 30 );
}
```

This will produce following result:

```
Value of b is 30
Value of b is 20
```

`switch` *statement:*

The `switch` statement is much like a nested `if .. else` statement. It's mostly a matter of preference which you use, `switch` statement can be slightly more efficient and easier to read.

```
switch( expression )
     {
        case constant-expression1:   statements1;
        [case constant-expression2:  statements2;]
        [case constant-expression3:  statements3;]
        [default : statements4;]
     }
```

Example:

```
#include <stdio.h>

int main(void)
{
     int  Grade = 'A';

     switch( Grade )
     {
        case 'A' : printf( "Excellent\n" );
        case 'B' : printf( "Good\n" );
        case 'C' : printf( "OK\n" );
        case 'D' : printf( "Mmmmm....\n" );
        case 'F' : printf( "You must do better than this\n" );
        default  : printf( "What is your grade anyway?\n" );
     }
}
```

Question: What is wrong with above example? How to fix it?

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

`while` *loop*

The most basic loop in C is the `while` loop. A `while` statement is like a repeating `if` statement. Like an `if` statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

```
while ( expression )
```

```
{
   Single statement
   or
   Block of statements;
}
```

Example:

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    while ( i > 0 )
    {
       printf("Hello %d\n", i );
       i--;
    }
}
```

Question: How many times does the program print the word "`Hello`"?

If we want to produce the following result, how should we modify the program?

```
Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
```

`for` *loop*

`for` loop is similar to `while`, it's just written differently. `for` statements are often used to process lists such a range of numbers:

Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
   Single statement
   or
   Block of statements;
}
```

In the above syntax:

- expression1 - Initialisese variables.
- expression2 - Condtional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.

Example:

```
#include <stdio.h>

int main(void)
{
```

```
    int i;
    int j = 10;

    for( i = 0; i <= j; i ++ )
    {
       printf("Hello %d\n", i );
    }
}
```

Question: How many "`Hello`" are printed from the program?

`do...while` *loop*

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

```
do
{
   Single statement
   or
   Block of statements;
}while(expression);
```

Example:

```
#include <stdio.h>

int main(void)
{
    int i = 10;

    do{
       printf("Hello %d\n", i );
       i = i -1;
    } while ( i > 0 );
}
```

Question: How many "`Hello`" are printed from the program? And, what would be the result if we change the statement `i = 10` to `i = 0`?

`break` *and* `continue` *statements*

C provides two commands to control how we loop:

`break` -- exit form loop or switch.
`continue` -- skip 1 iteration of loop.

Here is an example showing usage of `continue` statement.

```
#include <stdio.h>

int main(void)
{
    int i;
    int j = 10;
```

```
    for( i = 0; i <= j; i ++ )
    {
       if( i == 5 )
       {
          continue;
       }
       printf("Hello %d\n", i );
    }
}
```

This will produce following output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
```

Question: Modify the program to skip printing the line, "`Hello 8`".

Function

Similar to variable, function should be declared before it is used. In addition, when declare a function, the return type of data should be declared and as well as a parameter list. See example below.

```
int max(int num1, int num2)
{
    int result;
    if (num1 > num2)
          result = num1;
    else
          result = num2;
    return result;
}
```

Above example simply returns a bigger number when the function is called.

Question: The following function may not work as what we except. Why?

```
void swap(int num1, int num2)
{
    int temp;
    temp = num1;
    num1 = num2;
    num2 = temp;
    return;
}
```

Note: Will discuss this in "pointer".

Array

In general, an array in C is a variable that holds multiple elements which shares the same data type. In addition, array is a kind of data structure that can store a fixed-size sequential collection of elements of the same type.

Declaring an Array

Syntax: `type arrayName [ arraySize ]`

Example:
```
int num[5]
int num[5] = {91, 8, 13, 6, 1}
int num[]  = {91, 8, 13, 6, 1}
```

Above examples show how an array can be declared and initialized. That declares an array named `num` and holds five elements.

Accessing array elements

Notice that the array elements start from 0, i.e. the first element of the `num` array is `num[0]`. And, the last element is `num[size-1]` where `size` is the number of element in the `num` array, i.e. `num[4]`.

The following program shows how to access elements of an array.

```
#include <stdio.h>

int main()
{
   int num[5] = {91, 8, 13, 6, 1};
   int i;

   for(i = 0; i < 5; i++)
   {
       printf("num[%d] = %d\n",i,num[i]);
   }

   return 0;
}
```

Note: Will discuss more about array after "pointer". For example, how to pass an array to functions, how to return an array from a function and pointer to array.

Pointer

A pointer is a special kind of variable. Pointers are designed for storing memory address i.e. the address of another variable. Declaring a pointer is the same as declaring a normal variable except you stick an asterisk `'*'` in front of the variables identifier.

- There are two new operators you will need to know to work with pointers. The "address of" operator `'&'` and the "dereferencing" operator `'*'`. Both are prefix unary operators.

- When you place an ampersand in front of a variable you will get it's address, this can be stored in a pointer vairable.

- When you place an asterisk in front of a pointer, you will get the value at the memory address pointed to.

Here is an example to understand what we have stated above.

```
#include <stdio.h>

int main(void)
{
  int my_variable = 6, other_variable = 10;
  int *my_pointer;

  printf("the address of my_variable is    : %p\n", &my_variable);
  printf("the address of other_variable is : %p\n",
          &other_variable);

  my_pointer = &my_variable;

  printf("\nafter \"my_pointer = &my_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  my_pointer = &other_variable;

  printf("\nafter \"my_pointer = &other_variable\":\n");
  printf("\tthe value of my_pointer is %p\n", my_pointer);
  printf("\tthe value at that address is %d\n", *my_pointer);

  return 0;
}
```

This will produce following result.

```
the address of my_variable is    : 0xbfffdac4
the address of other_variable is : 0xbfffdac0

after "my_pointer = &my_variable":
        the value of my_pointer is 0xbfffdac4
        the value at that address is 6

after "my_pointer = &other_variable":
        the value of my_pointer is 0xbfffdac0
        the value at that address is 10
```

More explanation

Syntax:

`int *x` – Tells compiler that variable **`x`** is address of an **`int`**
`x = &y` – Tells compiler to assign address of **`y`** to **`x`** (**`&`** is the "address operator")
`z = *x` – Tells compiler to assign value at address in **`x`** to **`z`** (**`*`** is the "dereference operator")

Read the following fragment and try to figure out the value of `p` points to finally and what is `p` holding?

```
int *p, x;
x = 3;
p = &x;
```

What would happen if the following statement was added to the fragment?

```
*p = 5;
```

Discussion 1: About the function `swap(int num1, int num2)`

As you may notice that the function does not *swap* the values of the two variables as expected. We have just passed arguments to a function by value. The arguments have been copied into copied variables and the function works on those copies variables, not the original ones. If we have to swap the two variables' values, we should use pointers (pass arguments to a function by pointers). See the example below.

```
#include <stdio.h>

/* function declaration */
void swap(int *num1, int *num2);

int main ()
{
   /* local variable definition */
   int x = 100;
   int y = 200;

   printf("Before swap, value of x : %d\n", x );
   printf("Before swap, value of y : %d\n", y );

   /* calling x function to swap the values.
      * &x indicates pointer to x ie. address of variable x and
      * &y indicates pointer to y ie. address of variable y.
   */
   swap(&x, &y);

   printf("After swap, value of x : %d\n", x );
   printf("After swap, value of y : %d\n", y );

   return 0;
}

void swap(int *num1, int *num2)
{
   int temp;
   temp  = *num1;    /* save the value at address num1 */
   *num1 = *num2;    /* put num2 into num1 */
   *num2 = temp;     /* put temp into num2 */

   return;
}
```

Discussion 2: Pass an array to a function

One of the following methods to declare the parameter of an array:

```
void myFunction(int *arr)
{
   .
   .
   .
}
```

or

```
void myFunction(int arr[5])
{
   .
   .
   .
}
```

or

```
void myFunction(int arr[])
{
   .
   .
   .
}
```

Try the following example.

```
#include <stdio.h>

/* function declaration */
double getAverage(int arr[], int size);

int main ()
{
   /* an int array with 5 elements */
   int num[5] = {91, 18, 25, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( num, 5 ) ;

   /* output the returned value */
   printf( "Average value is: %f ", avg );

   return 0;
}

double getAverage(int arr[], int size)
{
   int i;
   double avg;
   double sum = 0;

   for (i = 0; i < size; ++i) {
      sum += arr[i];
   }
   avg = sum / size;

   return avg;
}
```

Discussion 3: Return array from a function

To return an array from a function, the function should have declared a returning pointer as shown in the following example.

```
#include <stdio.h>

/* function to generate and return random numbers */
```

```
int * getRandom( )
{
   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );

   for ( i = 0; i < 10; ++i)
   {
      r[i] = rand();
      printf( "r[%d] = %d\n", i, r[i]);
   }

   return r;
}

/* main function to call above defined function */
int main ()
{
   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();

   for ( i = 0; i < 10; i++ )
   {
      printf( "*(p + %d) : %d\n", i, *(p + i));
   }

   return 0;
}
```

Note: C does not advocate returning an address of a local variable to outside of the function, so we have to define the local variable as **`static`** variable.

Discussion 4: Pointer to an array

Indeed, an array name is a constant pointer to the first element of the array. For example, `int arr[5];` here `arr` is a pointer to `&arr[0]`, which is the address of the first element of the array `arr`. Below, we have a program fragment that assigns `ptr` as the address of the first element of `arr`.

```
int *ptr;
int arr[5];

p = arr;
```

The following may help to illustrate a bit more. Read this carefully.

```
#include <stdio.h>

int main ()
{
   /* an array with 5 elements */
   int arr[5] = {89, 73, 105, 6, 47};
   int *ptr;
```

```
    int i;

    ptr = arr;

    /* output each array element's value */
    printf( "\nArray values using pointer\n");

    for ( i = 0; i < 5; i++ )
    {
       printf("*(ptr + %d) : %d\n",  i, *(ptr + i) );
    }

    printf( "\nArray values using arr as address\n");

    for ( i = 0; i < 5; i++ )
    {
       printf("*(arr + %d) : %d\n",  i, *(arr + i) );
    }

    return 0;
}
```

We assign the address of arr to the pointer `ptr` (`ptr = arr;`). Then you can see the similar operations of these two variables. We try to access to different elements of the array by adjusting the addresses, e.g. `*(arr + 2)` is to access the data at `arr[2]`.

Exercise

Given an array `num[5]` and consists of values `{11, 55, 22, 44, 33}`. Write a C-program with functions to manipulate the values of the array. There are two functions to be implemented. The first function is to double each value in the array. The second one is to sort the values in the array in ascending order. After a function is called and executed, print out the result immediately in the [`main`].

Please use the following program fragment to fill in your codes and submit the source code file to Blackboard.

```
/*                                      */
/* Student Name:                        */
/* Student Number:                      */
/*                                      */

#include <stdio.h>

   /* the first function goes here */
   void doubleArr(        )
   {


   }

   /* the second function goes here */
   void sortArr(        )
   {


   }

/* main program */

int main ()
{
   /* given array is listed below */
   int num[5] = {11, 55, 22, 44, 33};



   /* call the doubleArr function */


   /* print the five elements of array num[] */


      /* call the sortArr function */


   /* print the five elements of array num[] */



   return 0;
}
```

- End -