Tutorial 4 : 23 August

1 Find the output for all the following problems

1.1 Problem 1

```
1  #include <stdio.h>
2
3  int main() {
4      int i = -3, j = 3;
5      if(!i + !j*1) {
6          printf("No");
7      } else
8          printf("Yes");
9          printf(", I am a fool");
10     return 0;
11 }
```

Output :
Yes, I am a fool

Explanation :

- Note that it is allowed to declare multiple variables in a single line as done here in line 4.
- Another important point is that it is not necessary to specify { and } with the *if-else* statements. If the braces are not specified, the statement following immediately after the if or *else* clause gets associated with it. For example, here line 9 is not a part of *else* but only line 8 is. Similar is the behaviour of braces with *else if*, *for*, and *while* statements too.
- The logical operators(like !, && and ||) treat every non zero number as being *TRUE* and zero as *FALSE*. Also the value returned by these operators is of type *int* and can only be 0(for *FALSE*) or 1(for *TRUE*). To understand the concept of return value, think of these operators as being mathematical functions themselves say $f(x) = !x$ of the following form.

$$f(x) = !x = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases}$$

Therefore, in line 5,

$$if\ (!i + !j * 1) \equiv if\ (f(-3) + f(3) * 1) \equiv if(0)$$

Similar to above, *if (expression)* will be interpreted as *TRUE* only when the value of *expression* is non-zero (e.g. -1, 10, etc.). This also applies for *else if(expression)*, *while(expression)*, and so on. Thus we get the output corresponding to the else statement in this example.

- As can be seen in line 9, improper indentation can be very misleading even for experts. Thus always try to indent your code consistently to improve its readability.

1.2 Problem 2

```
1 #include <stdio.h>
2
3 int main() {
4     int i, j;
5     i = j = 20;
6     printf("i = %d, j = %d", i, j);
7     return 0;
8 }
```

Output :
i = 20, j = 20

Explanation :

- Since = is right associative, instruction 5 is equivalent to

  i = (j = 20);

- C treats = as a binary operator which returns the value of the left hand side of the expression, say like * operator which takes 2 operands and return their product. Thus when executing this instruction 5, the following steps are performed :

  1. The value 20 is stored in j and the value of j is returned(which is now 20)
  2. The value of i is stored in j and the value of j is returned(which is now 20 too)

  Hence we get the above output. This example shows that there is always a return value of assignment operator, but is mostly ignored.

1.3 Problem 3

```
1 #include <stdio.h>
2
3 int main() {
4     int i = 10;
5     do {
6       printf("i = %d, \n", i);
7     } while (i = 20);
8     return 0;
9 }
```

Output :
i = 10,
i = 20,
i = 20,
i = 20,
i = 20,
..
..

Explanation :
As explained in above problems, in instruction 7 the value 20 will be stored in i and the value of i will returned by the assignment operator. Thus the instruction will **then** become equivalent to

```
} while (20);
```

Therefore, it gets stuck in an infinite loop.
To try: What would be the output if instruction 7 was

```
} while (i = 0);
```

2 Practice problems for finding output

2.1 Problem 1

```
1  #include <stdio.h>
2
3  int main( )
4  {
5    int x, y;
6    for ( x = 4, y = 0; x >= 0; x--, y++)
7    {
8      if ( x == y )
9        break;
10     else
11       printf ( "%d %d\n", x, y ) ;
12   }
13 }
```

Output :
4 0
3 1

2.2 Problem 2

Same as problem 1 but statement 9 replaced by

```
continue;
```

Output :
4 0
3 1
1 3
0 4

3 Go through the following code and try to figure out the underlying algorithm

```
1  #include <stdio.h>
2
3  int main( )
4  {
5    int x, y, num;
6    for ( x = 1; x <= 3; x++)
7    {
8      int x, sum = 0;
9      for ( x = 1; x <= 3; x++)
10     {
11       scanf("%d", &num);
12       sum += num;
13     }
14     printf("%d\n", sum);
15   }
16 }
```

Input :
1 2 3 4 5 6 7 8 9

Output :
6
15
24