Tutorial 2 : 9 August

1 ASCII Character Mapping

The following is the ASCII mapping for some of the characters :
'A' - 'Z' : 65 - 90
'a' - 'z' : 97 - 122
'0' - '9' : 48 - 57
This mapping does not need to be remembered but the idea is that the characters are internally represented by codes between 0-255. Also, some of these characters are non printable, thus sometimes when printing using '%c', you might not see anything in the output.

2 Find the value of the variable p in each case

- `int p = 7;`
  `p = p + 1;`

  Solution:
  8

- `int p = 'A' + (char)230;`

  Solution:
  $= (65 + 230)\%256$
  $= 290\%256$
  $= 39$
  Explanation:
  %256 here represents that the actual calculation of RHS is done simply assuming both of them as characters and not integers. Thus the value of RHS is always confiined within the range 0-255. The main idea is that the type of LHS is not important when computing the value of RHS, but important only when we finally store the computed sum into LHS.

- `int p = 'A' + 230;`

  Solution:
  $= (65 + 230)$
  $= 290$
  Explanation:
  In this case the character is implicitly converted to an integer since integer is a larger data type.Thus no wrap around takes place. You can try more cases for such implicit casting rules.

- `int p =-5*4/5*3+-1*2;`

  Solution:
  $= ((((-5) * 4)/5) * 3) + ((-1) * 2)$
  $= (((-20)/5) * 3) + (-2)$
  $= -14$
  Explanation:
  * and / have same priority, and left associative

- `float p = 25/3;`

  Solution:
  8.0
  Explanation:
  Since both 25 and 3 are integers, 25/3 computed as integer and then stored in a float

- `float p = 25/3.0;`

  Solution:
  8.3333

- `float p = (float)25/3 + 3/-100;`

  Solution:
  $= (((float)25)/3) + (3/(-100))$
  $= (((25.0)/3) + (3/(-100))$
  $= 8.3333 - 0$
  $= 8.3333$

- `int p = !1&&1&&1||0;`

  Solution:
  $= (((!1)$ && $1)$ && $1)||0$
  $= 0$
  Explanation:
  A simple trick to remember the precedence order of !, && and || is to remember the acronym 'NAO' which stands for Not-And-Or. So the precedence are ! > && > ||

3 Scanf example 1

Code :

```
#include <stdio.h>

int main()
{
    int x,y;
    printf("Enter co-ordinates in format (x,y) : ");
    int inputCount = scanf("(%d, %d)", &x, &y);
    printf("Co-ordinates entered : (%d, %d) \n", x, y);
    printf("Number of input placeholders read : %d\n", inputCount);
    return 0;
}
```

Input:
(2,3)

Output:
Co-ordinates entered : (2, 3)
Number of input placeholders read : 2

Explanation:

- It is very important to specify &x in scanf and not just x because scanf needs an address of the variable where it will store the entry read from input. Thus &x provides it the address of the variable x. If you just specify x, it will assume the value already present in x as the address and thus you will see unexpected runtime results(not compile time errors).

- It is important that the placeholder(like %d) mentioned in the format string(the string specified as first argument) in scanf is of the same type as the variable specified. For example,

  ```
  float x;
  scanf("%d", &x);
  ```

  Here, the value of variable x will not be read appropriately even if an integer, say 2 is given as input. Also such a code will not give any compile time warning because the compiler assumes that the data type for the specified variable is appropriate.

- Providing words in the scanf format string tries to fix the format for the expected input from the user. For example, consider the 3rd instruction in the above code

```
int inputCount = scanf("(%d, %d)", &x, &y);
```

Specifying (and) in the format string will now expect the input to be provided in the same format. You can try different examples to see the kind of outputs you get on runtime.

- On success, the return value of scanf is the number of placeholders read.

To try: Run the above code with following inputs and try to explain the results:

- (2 3)
- 2,3)
- (2,3
- (2, 3)

4 Scanf example 2

Code :

```
#include <stdio.h>

int main()
{
    int num1, num2;
    float decimalNum;
    scanf("%3d %5f %d", &num1, &decimalNum, &num2);
    printf("num1 = %d\n", num1);
    printf("decimalNum = %f\n", decimalNum);
    printf("num2 = %d\n", num2);
    return 0;
}
```

Input:
1234567.89

Output:
num1 = 123
decimalNum = 4567.000000
num2 = 89

Explanation: For this example, imagine that the input is placed in a big buffer and scanf is trying to read from this buffer.

'1'	'2'	'3'	'4'	'5'	'6'	'7'	'.'	'8'	'9'

Now %3d reads the first 3 characters(ie. 123) and stores it as integer in num1. Now the buffer looks like:

'4'	'5'	'6'	'7'	'.'	'8'	'9'

%5f reads the next 5 character(ie. 4567.) and stores it as a floating number in decimalNum. The buffer will now look like:

'8'	'9'

Then %d reads the remaining characters till the next whitespace character(whitespace characters include spaces, newline and tab characters) and store it as an integer in num2.

You can try different combinations in format string and try to explain the output.

5 Something to explore:

Overflow in variables is a common problem where we try to store results with values larger than the maximum size of the variable type. For example, something like :

```
int a = 999999;
int b = 999999;
int x = a*b;
```

Therefore variable x will not be able to store the correct product value and would give wrong answers at runtime. You can see some real life examples of such errors at the following links :

- Gangnam style negative views : http://www.economist.com/blogs/economist-explains/2014/12/economist-explains-6
- Ariane 5 rocket failure : https://around.com/ariane.html