Programming

and

Problem Solving using ‘C’

Lecture Notes

by

N S Kumar

kumaradhara@gmail.com

**Output**

We have had a look at our first program in ‘C’. Let us understand how the function printf works.

Please refer to the program ex1.c.

* printf("hello world\n");

The first argument to printf is a string. The output depends on this string. String literals in ‘C’ are enclosed in double quotes.

\n : is an example of escaping. This changes the meaning of ‘n’ - this character ‘\n’ is the new line character – causes the cursor to go to the next line on output.

* printf("hello ", "world\n"); // hello

Both these arguments are put on the activation record or stack frame. But the output depends on the first string. As there is no interpretation of the second parameter, it would not appear in the output.

* printf("hello %s\n", "world\n"); // hello world

The presence of %s in the first string makes the function printf look for the next parameter in the stack and interpret it as a string.

* printf("%s %s %s %s\n", "one", "two", "three", "four");

The function printf takes variable number of arguments. All arguments are interpreted as the number of %s matches the number of strings following the first string.

* printf("%s %s %s %s\n", "one", "two", "three");

NO! we are in for trouble. There is no argument for the 4th %s in the format string. So, printf tries to interpret whatever is there in the stack at the possible location as a string. If we are lucky, the program will crash. We have an “undefined behaviour”. C does no checking at runtime!

* printf("%5d and %5d is %6d\n", 20, 30, 20 + 30);
  + Arguments to printf can be expressions – Then the expressions are evaluated and their values are passed as arguments.
  + %d : indicates to printf to interpret the next parameter as an integer.
  + %5d : tells printf to display the integer using 5 character width.
* printf("what : %d\n", 2.5);
  + GOD if any should help the programmer who writes such code!!
  + undefined behaviour.

Let us digress to discuss the last statement in detail.

A few points to note about totally compiled languages like ‘C’.

* There is no translator at runtime in ‘C’. It exists in scripting languages like Python.

As the translator exists at runtime, we can take any valid string in Python and compile and execute at runtime. You may remember the functions eval and exec. But this is not possible in ‘C’ as the compiler does not exist at runtime.

* In ‘C’, type is a compile mechanism and value is a runtime mechanism. In Python, a value has a type and the type remains at runtime. We can query any value regarding its type in Python at run time.

Having type at runtime would require more space at runtime. Any functions based on type shall be also costly in terms of time and space. As the goal of ‘C’ is efficiency, only values are stored at runtime. There is no way to infer the type by looking at the bit pattern.

* All the code that executes at runtime should be compiled during compilation itself and cannot be added at runtime.

Let us look at our present example.

* printf("what : %d\n", 2.5);

The function printf takes varying number of arguments. In such cases, the compiler cannot match arguments and parameters for type. So, the compiler might emit some warning (if it knows what printf expects) – but cannot throw errors.

As the compiler does not know that 2.5 should be converted to an integer, it does not do any conversion.

So, the compiler puts the value of 2.5 the way it is expected to be stored – as per the standard IEEE 754. This standard uses what is called mantissa exponent format to store fractional (floating point) values.

At runtime, printf tries to interpret the bit pattern as an integer when it encounters the format %d. At runtime, no conversion can occur – we do not even know that what is stored as a bit pattern is a floating point value – Even if we know, there is no way to effect a conversion as we do not have a compiler at that point. We end up getting some undefined value.

So, in ‘C’, **you get what you deserve.**

There are a few basic types in ‘C’. int, char, long, short are called integral types and these are exactly stored in the computer. Float and double are approximately stored.

The size of a type – size required to represent a value of that type – can be found using an operator called sizeof.

The sizeof a type depends on the implementation. We should never conclude that the size of an int is 4 bytes. Can you answer this question – How many pages a book has? What is the radius of Dosa we get in different eateries?

The size of a type is decided on an implementation based on efficiency.

**Variable**:

This is the most important concept in all languages.

A variable has a name, a value, a location and a type. It also has something called life, scope, qualifiers etc. We will discuss the second part later in the course.

A variable has to have name by which we can refer to it in the program. Sometimes, the runtime makes variables with no names. These are called temporary variables. We cannot access them by name as we have no name in out vocabulary.

A variable has a type – like int, double. In Python, the type is based on the value assigned to the variable and the type of a variable can change at runtime. In ‘C’, type should specified before a variable is ever used. We declare the variable with respect to type before using it.

Example:

int a; double b;

The type of a variable can never change during the program execution. The type decides what sort of values this variable can take and what operations we can perform. Type is a compile time mechanism. The size of a value of a type is implementation dependent and is fixed for a particular implementation.

We can initialize a variable at the point of declaration. An uninitialized variable within a block has some undefined value. ‘C’ does not initialize by any default value.

int c = 100;

int d; // undefined value !!

A variable can be assigned a value later in the code.

c = 200;

d = 300;

Assignment is considered an expression in ‘C’ - whereas assignment is not an expression in Python.

Printf(“%d”, c = 400); // ok

This expression c = 400; assigns 400 to c and the value assigned to c is the value of the expression.

‘C’ follows parameter passing by value. So, we cannot call a function to change the value of a variable by passing the variable name as an argument. In such languages, we require a mechanism to pass the location – to be precise the address of the location – as argument to change the variable. Unary operator & gives the address of the variable.

Let us compare with Python. In Python, we always read the input as a string – there is no formatting of the input. We then split and pass the resultant strings to the constructors of the corresponding types. The input functions return a string back and do not change any argument – in fact there are no arguments!

We use scanf to read from the keyboard. We specify the format - %d to read an int and %f to read a float and so on.

Refer to the program ex2.c.

scanf("%d", a); // dangerous

scanf("%d", &a); // address operator

printf("a : %d ", a);

scanf like printf takes a format string as the first argument. The input should match the string. As the scanf below has a comma between two format specifiers, the input should have a comma between a pair of integers.

scanf("%d,%d", &a, &b);

printf("a : %d b : %d\n", a, b);