

# C Tokens

## History of C

The *milestones* in C's development as a language are listed below:

- UNIX developed c. 1969 - DEC PDP-7 Assembly Language
- BCPL - a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- A new language "B" a second attempt. c. 1970.
- A totally new language "C" a successor to "B". c. 1971
- By 1973 UNIX OS almost totally written in "C".

## Characteristics of C

We briefly list some of C's characteristics that define the language and also have led to its popularity as a programming language. We will be studying many of these aspects throughout the course.

- Small size
- Extensive use of function calls
- Loose typing - unlike PASCAL
- Structured language
- Low level (BitWise) programming readily available
- Pointer implementation - extensive use of pointers for memory, array, structures and functions.

C has now become a widely used professional language for various reasons.

- It has high-level constructs.
- It can handle low-level activities.
- It produces efficient programs.
- It can be compiled on a variety of computers.

Its main drawback is that it has poor error detection which can make it off putting to the beginner. However diligence in this matter can pay off handsomely since having learned the rules of C we can break them. Not many languages allow this. This if done properly and carefully leads to the power of C programming.

## C Program Structure

A C program basically has the following form:

- Preprocessor Commands
- Type definitions
- Function prototypes -- declare function types and variables passed to function.
- Variables
- Functions

We must have a main() function.

A function has the form:

*type* function\_name (*parameters*)

```
{  
    local variables  
    C Statements  
}
```

If the type definition is omitted, C assumes that function returns an integer type.

*Note:* This can be a source of problems in a program.

So returning to our first C program:

```
/* Sample program */  
main()  
  
    {  
    printf( ``She Likes C \n" );  
    exit ( 0 );  
    }
```

*Note:*

- C requires a semicolon at the end of every statement.
- printf is a *standard* C function -- called from main.
- \n signifies newline. Formatted output -- more later.

- `exit()` is also a standard function that causes the program to terminate. Strictly speaking it is not needed here as it is the last line of `main()` and the program will terminate anyway.

Let us look at another printing statement:

```
printf(``.\\n.1\\n..2\\n...3\\n");
```

The output of this would be:

```
.1
..2
...3
```

**Variables** : C has the following simple data types:

C type	Size (bytes)	Lower bound	Upper bound
<code>char</code>	1	—	—
<code>unsigned char</code>	1	0	255
<code>short int</code>	2	-32768	+32767
<code>unsigned short int</code>	2	0	65536
<code>(long) int</code>	4	$-2^{31}$	$+2^{31} - 1$
<code>float</code>	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
<code>double</code>	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

On UNIX systems all ints are long ints unless specified as short int explicitly. (Although not in the syllabus, but believe without UNIX flavour any programming paradigm is incomplete).

Unsigned can be used with all char and int types. To declare a variable in C, do:

```
var_type list variables;
```

```
e.g. int i,j,k;
```

```
float x,y,z;
```

```
char ch;
```

### Defining Global Variables

Global variables are defined above `main()` in the following way:-

```
short number,sum;
```

```
int bignumber,bigsum;
```

```
char letter;
```

```
main()
```

```
{
```

```
}
```

It is also possible to pre-initialise global variables using the = operator for assignment.

For example:

```
float sum=0.0;
int bigsum=0;
char letter=`A';
main()
{
}
```

This is the same as:-

```
float sum;
int bigsum;
char letter;
main()
{
sum=0.0;
bigsum=0;
letter=`A';
}
```

...but is more efficient.

C also allows multiple assignment statements using =, for example:

```
a=b=c=d=3;
```

...which is the same as, but more efficient than:

```
a=3;
b=3;
c=3;
d=3;
```

This kind of assignment is only possible if all the variable types in the statement are the same. You can define your own types use typedef. This will have greater relevance later in the course when we learn how to create more complex data structures. As an example of a simple use let us consider how we may define two new types real and letter. These new types can then be used in the same way as the pre-defined C types:

```
typedef real float;
typedef letter char;
```

*Variables declared:*

```
real sum=0.0;  
letter nextletter;
```

## Printing Out and Inputting Variables

C uses formatted output. The `printf` function has a special formatting character (%) - a character following this defines a certain format for a variable:

`%c` -- characters

`%d` -- integers

`%f` -- floats

*e.g.* `printf(` ` %c %d %f",ch,i,x);`

*Note:* Format statement enclosed in "...", variables follow after. Make sure order of format and variable data types match up.

`scanf()` is the function for inputting values to a data structure: Its format is similar to `printf`:

*i.e.* `scanf(` ` %c %d %f",&ch,&i,&x);`

*Note:* & before variables. Please accept this for now and remember to include it. It is to do with pointers which we will meet later.

## Constants

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The `const` keyword is to declare a constant, as shown below:

```
int const a = 1;
```

```
const int a =2;
```

*Note:*

- You can declare the `const` before or after the type. Choose one and stick to it.
- It is usual to initialize a `const` with a value as it cannot get a value.

The preprocessor `#define` is another more flexible (see Preprocessor Chapters) method to define *constants* in a program. You frequently see `const` declaration in function parameters. This says simply that the function is not going to change the

value of the parameter. The following function definition uses concepts we have not met so far but for completeness of this section it is included here:

```
void strcpy(char *buffer, char const *string)
```

The second argument string is a C string that will not be altered by the string copying standard library function.

## Arithmetic Operations

As well as the standard arithmetic operators (+ - \* /) found in most languages, C provides some more operators. There are some notable differences with other languages, such as Pascal.

Assignment is *i.e.* `i = 4; ch = 'y';`

Increment ++, Decrement - which are more efficient than their long hand equivalents, for example: `x++` is faster than `x=x+1`. Guess ! Why?

The ++ and - operators can be either in post-fixed or pre-fixed. With pre-fixed the value is computed before the expression is evaluated whereas with post-fixed the value is computed after the expression is evaluated. In the example below, ++z is pre-fixed and the w- is post-fixed:

```
int x,y,w;
main()
{
x=((++z)-(w-)) % 100;
}
```

This would be equivalent to:

```
int x,y,w;
main()
{
z++;
x=(z-w) % 100;
w-;
}
```

- The % (modulus) operator only works with integers.
- Division / is for both integer and float division. So be careful.

- The answer to:  $x = 3 / 2$  is 1 even if  $x$  is declared a float!!
- *Rule:* If both arguments of  $/$  are integer then do integer division.
- So make sure you do this. The correct (for division) answer to the above is  $x = 3.0 / 2$  or  $x = 3 / 2.0$  or (better)  $x = 3.0 / 2.0$ .
- There is also a convenient shorthand way to express computations in C.

## Comparison Operators

To test for equality is `==`

A warning: Beware of using ```="` instead of ```=="`, such as writing accidentally  
`if ( i = j ) .....`

This is a perfectly LEGAL C statement (syntactically speaking) which copies the value in "j" into "i", and delivers this value, which will then be interpreted as TRUE if j is non-zero. This is called assignment by value -- a key feature of C.

Not equals is: `!=`

Other operators `<` (less than) , `>` (grater than), `<=` (less than or equals), `>=` (greater than or equals) are as usual.

## Logical Operators

Logical operators are usually used with conditional statements The three basic logical operators are:

`&&` for logical AND, `||` for logical OR ! for Logical NOT.

Beware `&` and `|` have a different meaning for bitwise AND and OR

## The C Preprocessor

Recall that preprocessing is the first step in the C program compilation stage - this feature is unique to C compilers. The preprocessor more or less provides its own language which can be a very powerful tool to the programmer. Recall that all preprocessor directives or commands begin with a `#`.

Use of the preprocessor is advantageous since it makes:

- programs easier to develop,
- easier to read,
- easier to modify
- C code more transportable between different machine architectures.

During compilation all occurrences of begin and end get replaced by corresponding { or } and so the subsequent C compilation stage does not know any difference!!!. Lets look at #define in more detail

#define

Use this to define constants or any macro substitution. Use as follows:

```
#define <macro> <replacement name>
```

*For Example*

```
#define FALSE 0
#define TRUE !FALSE
```

We can also define small "functions" using #define. For example max. of two variables:

```
#define max(A,B) ( (A) > (B) ? (A):(B))
```

? is the ternary operator in C.

## Increment And Decrement Operators

C offers two special operators ++ and -- called increment and decrement operators, respectively. These are unitary operators since they operate on only one operand. The operand has to be a variable and not a constant. Thus, the expression a++ is valid whereas 6++ is invalid. The use of these operators results in incrementing or decrementing the value of the variable by 1. So the expression a++ increments value in a by 1, and the expression a-- decrements it by 1. These operators can be used either before or after their operand (i.e. in either 'prefix' or postfix' position), so we can have a++ (postfix) as well as ++a (prefix), and a-- as well as --a. Prefix and postfix operators have same effect if they are used in an isolated C statement. For example, the effect of the following two statements I would be same. However, prefix and postfix operators have different effects when used in association with some other operator in a C statement. For example, if we assume the value of the variable a to be 5, then execution of the statement

```
b=++a;
```

will first increase the value of a to 6 and then assign that new value to b. The effect is exactly same as if the following two statements have been executed:

```
a=a+1; b=a;
```

On the other hand, execution of the statement

```
ob = a++ ;
```

Will first set the value of b to 5 and then increase the value of a to 6. The effect now is same as if the following two statements had been executed:

```
b=a;
```

```
a=a+1;
```

The decrement operators are used in a similar way, except, of course, the values of a and b are decreased. First the prefix decrement:

```
b = --a;
```

Which is same as

```
a=a-1 ; b=a; .
```

Now the postfix decrement:.

```
b = a-- ;
```

Which is same as

```
b=a;
```

```
a= a-1;
```

Note that the increment and decrement operators can not only be used in association with an assignment (=)operator, but also with any other operator and even with printf( ). This is shown in the following program:

```
main( ) {
int a = 10, b = 10 ;
printf ( "%d", a++ ) ; printf ( "%d", ++b) ;
-}
```

In the first printf( ) firstly value of a(10) is printed and then incremented to 11. As against this in the second printf( ) firstly b is incremented to 11 and then printed out. ~

## Modulo Division Operator

C provides one more binary arithmetic operator % called Modulo Division operator. This operator is a supplement to the division operator. We use it for division. But unlike division where we get the quotient on division, here we get the remainder. For example, what is the result of the expression 5 / 3? The result is 1 since the remainder is simply thrown away when in C an integer is divided by another integer. But what if we are interested in the remainder and not the quotient? You guessed it right. That's the time when the modulo division operator is to be used, as in, 5 % 3.

Note that 5 % 3 yields 2 whereas 3 % 5 yields 3. Similarly 6 % 3 or 3 % 3 or 9 % 3 all yield 0. Got it? Be careful, though. The modulo division operator works only on ints and chars and not on floats or doubles.

If you think you have grasped the concept, then try your hand at the following program. Can you guess its output?

```
main( ) {
```

```
printf ( "%d", 4 % 3 ) ;
printf ( "%d"r 4 % -3 ) ;
printf ( "%d", -4 % 3 ) ;
printf ("%d", -4 % -3);
}
```

The output of the first two printf( )s is 1 whereas that of the next two printf( )s is -1. This is because the sign of the output is always same as the sign of the numerator irrespective of the sign of the denominator.

## Relational Operators

These operators are used to compare two operands to see whether they are equal to each other, unequal, or whether one is greater than the other. The operands can be variables, constants or expressions that ultimately get evaluated to a numerical value. Since characters are also represented internally as integers (by their ascii code), elements to be compared can be characters as well. C has six relational operators. The following figure shows these operators alongwith their meanings.

### Meaning of Operator :

The meaning operator is important stuff to remember, here they are:

- < less than
- > greater than
- <= less than equal to
- >= greater than equal to == equal to
- != not equal to

Each relational operator needs two operands for comparison of their values. Hence, these operators come under the category of 'binary operators'. Following examples show the expressions involving the relational operators.

```
a<b
a == b + 3 d >= 5.66
alphabet != 't'
(p+q) <=(t+r-5)
```

The expression  $a < b$  means "is a less than b?". The result of comparing a with b maybe either 'true' or 'false'. If it is false, the value of "the expression is treated as 0, and if true as 1. The following program confirms this fact.

```
main() {
int a = 10, b = 20, C = 30, d, e ;
d=a>b; e=b<=c; printf ( "%d %d", d, e ) ;
}
```