C : A Tutorial Introduction

Objective here is to learn the basic through small and simple programming examples:

```
/* Your first program  hello.c */
void  main()
{
   printf (" Hello, World\n");
}
```

1. The program displays : Hello, World
2. Function main is calling a library function printf to do the job
3. \n (newline) is considered a single character and moves the print position to the beginning of the next line

1. A C program consists of a collection of one or more functions of which one must be the function main.
2. The term void typifies the function main that it returns nothing to its caller.
3. Matching curly braces { and } indicating the beginning and end of the function.
4. Anything between the matching /* and */ is ignored by the C compiler and are used as comments for better comprehension.

Running your program

In UNIX system the source file (hello.c) may be compiled and run by:

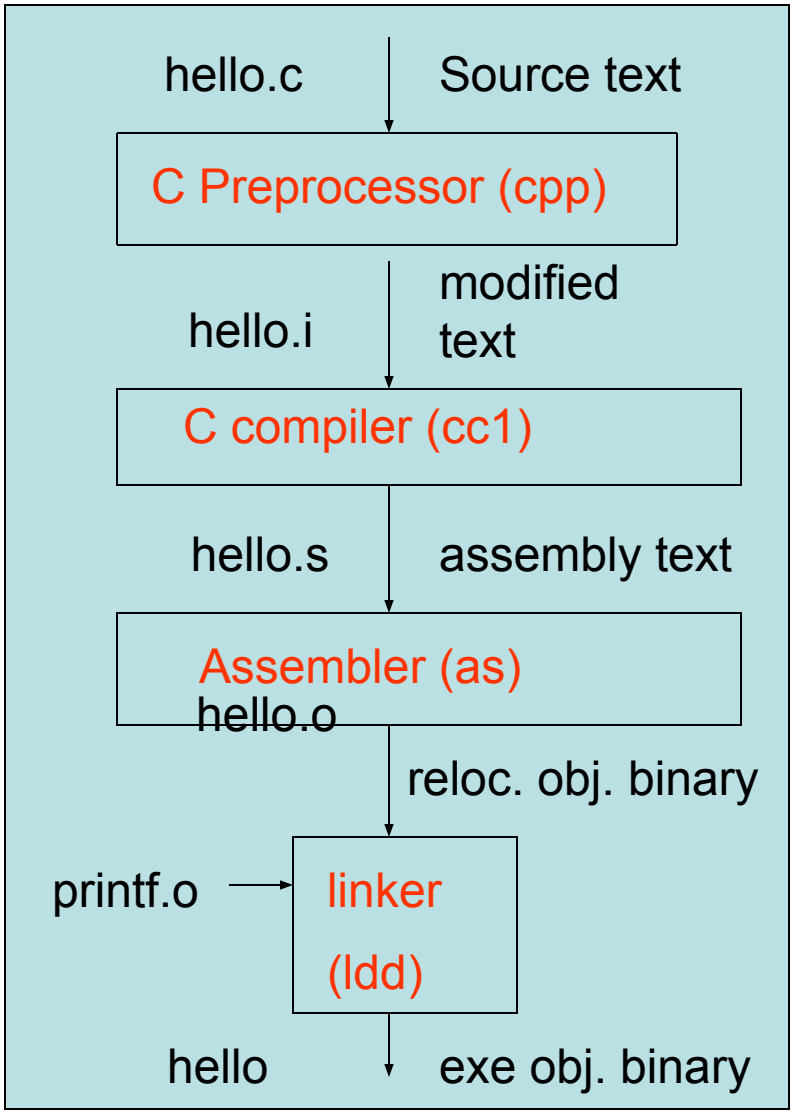
unix> gcc –o hello hello.c

unix> ./hello

It is a 4 step process as shown right.

Assembler's input is text but output is binary.

Linker links binary object files and produces binary executable.



Another simple program

```
void main() /* Print Fahrenheit-Celsius table */
{ int min, max, step; float cel, fahr;
  min = -100; max = 100; step = 5;
  fahr = min;
  while (fahr <= max) { /* begin while */
     cel = (5.0/9.0) * (fahr - 32.0)
     printf("%4.0f  %6.1f\n", fahr, cel);
     fahr = fahr + step;
  } /* end while */
}
```

The while loop continues until fahr, which is being increased by step, reaches to max. The conditional expression (fahr <= max) would be true so long the condition is satisfied; i.e., fahr is not greater than max.

Same program using different way of looping

In the previous program, a number of instructions within the body of the while (i.e, between the curly braces {}), is repeatedly executed. Same can be achieved as below with a for loop.

```
for (fahr = min; fahr <= max; fahr = fahr+step) {
    cel = (5.0/9.0) * (fahr -32.0)
    printf("%4.0f  %6.1f\n", fahr, cel);
}
```

Here, fahr is initialised to min only once. The condition fahr <= max is tested each time the loop is entered and at the end each time fahr is increased by step. The control passes to the next instruction after for when the condition (i.e., fahr <= max) becomes false.

Copy input

```
#define EOF    -1
void  main()
{   int c;
    c = getchar();
    while ( c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

This program reproduces what is being written at its input

getchar and putchar are two mathcing functions to read or write a character from the standard input and standard output devices respectively.

EOF stands for the end of file condition and the operating system returns -1 to indicate the same. To accommodate the negative value of EOF, c is defined as an integer. Note that any character is automatically promoted to an integer in an expression. An experienced programmer might squeeze the body of the program as below:

```
while ( ( c = getchar() ) != EOF)
        putchar(c);
```

Counting characters, words & lines

The copy input program can be enhanced to count characters and similar counting programs as it is reading all its input character by character.

```
#define EOF     -1
void  main()  /* Program to count characters */
{   long  nc = 0; int c;
    while ( ( c = getchar() ) != EOF)
        nc++;
    printf("%ld\n", nc);
}
/* while statement may be replaced by
    for(; (c = getchar()) != EOF; nc++);    */
```

Counting continues

Input lines are terminated by '\n' so we count lines by

```
… {  intc, nl=0;
    while ( (c = getchar()) != EOF)
        if ( c == '\n')
            nl++;
    printf ("%d\n", nl);
     }
```

Word counting can be done in this manner. Assume the white spaces are ' ', \n and \t i.e., the blank, tab and newline.

Counting words

```
#define  YES      1
#define  NO       0
void   main()
{    int c, nw =0; inword = NO;
     while ( ( c =getchar()) != EOF) {
          if ( c == ‘ ‘ || c == ‘\n’ || c == ‘\t’ )
               inword = NO;
          else if ( inword == NO) {
               inword = YES; nw++;
          }
     printf (“%d\n”, nw);
}
```

Using arrays as multiple counters

Consider the problem of counting alphanumerals in the input. We need 26 + 26 + 10 counters.

Here is a solution with arrays:

```
… int small[26], capital[26], numr[10];
    for (i=0; i<26; i++){ small[i] =0; capital[i]=0};
    for(i=0; i<10; i++) numr[i] =0;
    while((c = getchar()) != EOF)
     if ( c >='a' && c <= 'z') small[c-'a']++;
     else if (c>='A' && c <= 'Z') capital[c-'A']++;
     else (c>='0' && c<='9') numr[c-'0']++;
…
```

ASCII code of 'a' is 41H, 'b' is 42H …, 'A' is 61H, 'B' is 62H

Similarly, '0' is 30H while '1' is 31H

First we declare 3 arrays for all 62 characters.

Then each counter is set to 0 using two for loops.

A single while loop reads all characters at the input.

If-else is classifying the input

Data objects

Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, their types and their initial values. Operators specify what is to be done with them.

Variable names: Names made up of letters and digits; the first character should be a letter. Underscore (_) is a valid character and is used to increase the readability. Upper and lower case letters are different. Traditional C practice is to use lower case for variables and all upper case for symbolic constants.

Basic Data Types

C Provides a number of basic data types; they are:

- Integer type
  - int
  - short
  - long
  - unsigned
- Character type
  - char
- Real numbers
  - float (single precision)
  - double (double precision)

int specifies an integer typically reflecting the natural size of the host machine (16 or 32 bits). short and long are implementation dependent. However it is guaranteed that short cannot be more than an int and long cannot be less than an int representation. The qualifier unsigned indicates unsigned integer.

char takes a byte (8 bits)

float typically takes 32 bits and double takes 64 bits; however they are implementation dependent and might reflect special hardware aspect of the host machine

Character Constants

'x' indicates a character constant and is stored as a single byte ASCII value.

Certain non-graphics characters represented by escape sequence line \n (newline), \t (tab), \0 (null), \\ (backslash) and \' (quote). An arbitrary byte size bit patter can be used as '\ddd' e.g.,

#define FORMFEED '\014' /* ASCII FORM FEED */

"x" on the other hand is a string (multiple characters together) constant which occupies two bytes in memory; i.e., code for x followed by the null (\0) character which is the end of string character. Thus "ABC" is string constant of length 3 but takes 4 bytes in memory.

Constants

Symbolic constants may be defined as:

```
#define THISYEAR  2008
#define PI        3.14
```

A real constant can be written in scientific notation as: 123.456e7 or 0.123E3

Floating point constants are always promoted to double. Long constants are written as 123L. Large int constants are promoted to long. Octal constants are written as 037 (octal) or in hex as 0x1F (or 0X1F); 037L or 0x1FL make them long.

Declarations

All variables must be declared before it is used.

```
int lower, upper, step, i=0;
char c, word[15] = “C Programming”;
float epsilon = 1.0e-10;
int age; char backslash = ‘\\’;
```

Note that a declaration specifies a type, and is followed by one or more variables. Initialisation, if required, can be made with the declaration.

Arithmetic Operators

+, -, *, /, % representing addition, subtraction, multiplication, division and modulus operations respectively. Unary '–' is also used. Integer division truncates any fractional part (so, 15/7 produces 2) and modulus returns remainder (so, 15 % 7 produces 1). '+' and '–' have same precedence and are lower than the (identical) precedence of '*', '/' and '%' and are lower in precedence than unary '-'. Arithmetic operators group left to right. Unary operators '++' and '--' are also available. a++ means post-increment and --a means pre-decrement of the variable a.

Relational and logical operators

There are six relational operators;

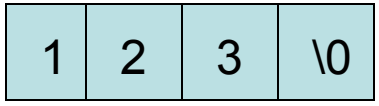
>, >=, <, <= (same precedence); just below them are the equality operators == and !=

Logical operators are && (and), || (OR), and ! (NOT). Expression connected by && or || are evaluated left to right and the evaluation stops as soon as the truth or falsehood of the result is known.

Type conversion

When different types of operands appear in a single expression they are converted to a common type that make sense. e.g., expression like i + f (i : integer and f : float); i is converted to float. In f + d, f is converted to d (double).

int and char can be freely intermixed. e.g.,

void atoi (char s[]) /* string 

to int 123 */

```
{   int i, n = 0;
    for (i=0; s[i] >= '0' && s[i] <= '9'; i++)
        n = n * 10 + s[i] – '0';
    return n;
}
```

Increment and Decrement Operators

++ and – are handy forms of incrementing or decrementing variables. They can be used both as prefix or postfix. Take care to note the difference: Say n = 5;

) x = ++n; would assign 6 to x and

) x = n++; would assign 5 to x

(In both the cases 6 will be assigned to n).

So, for simple increment use of postfix or prefix form does not matter. However, there are situations when one form or other is specifically called for.

Bitwise logical operator

&	and
|	or
^	xor
<<	left-shift
>>	right-shift
~	not

These operators are not applicable to float or double

In C any value other than 0 is considered as TRUE; so if a = 5 and b = 2 Then (a && b) yields TRUE. On the other hand (a & b) would yield 0 as the result. Note the internal representation of 5 and 2 are 101 and 010 respectively. So, bitwise anding (&) would set all these 3 bits to 0.

>> shifts the operand by one bit position to the right and the vacant space is filled with 0. So, >> may be used for integer division by 2 and << may be used to accomplish integer multiplication by 2.

Bitwise operation

To illustrate the use of bit operators let us examine the function getbits(x, p, n) which returns right adjusted n-bit field of x starting from p bit position from the right.

```
int getbits(unsigned x, unsigned p, unsigned n)
{
    return (x >> (p+1-n)) & ~(~0 << n));
}
```

For example getbits(x, 4, 3) returns 3 bits from the 4^{th} bit position.

C Keywords

The following identifiers are reserved. They may not be used otherwise:

auto break case char continue default
do double else entry
extern float for goto if
int long registerreturn short
sizeof static struct switch typedef union
unsigned void while

Storage Class

C offers 4 storage class; they are

i) Automatic; ii) External; iii) Static & iv) Register

Automatic: Within a function, normally the variables defined are all automatic variables (this includes the parameters passed to the functions).

Automatic variables are allocated storage in the stack and they come and go with the invocation of the function within which they are defined. They are not accessible from outside the function and are private to their own function.

Storage Class contd.

External: As the name suggests these variables are declared outside the body of the function and are allotted storage in data section of the program. As a result these variables are visible from anywhere. External variables are a handy for sharing/processing common data-items between a number of co-operating functions.

Note that the other alternative of passing many common parameters between a number of co-operating functions may be awkward.

Caution: As external variables are visible from anywhere inadvertent change is a possibility.

Use of external variable

Suppose a program manipulates a file and there are, say, five basic functions (fa, fb, fc, fd and fe) which do this processing. Also assume all these functions need to know the size of the file. Suppose the size is known and available in the main function as int filesize. Without the concept of external (global) variable you have no other alternative than passing the filesize as one of the input parameters to all the processing functions. This can be avoided with the help of external variable which is accessible from all the functions defined in a single source file; for e.g.;

Use of external variable

```
int filesize = 2048;
int main()
{
a = 2 * filesize; /* a global reference to filesize */
}
int x()
{
b =  4 * filesize; /* another reference */
}
int y()
{
filesize = 4096; /* assigning a new value for this global */
}
```

Storage Class contd.

Static: These are similar to automatic but retain their values between the invocation of the fnction. Static variables are two types; i) external and ii) internal. External static variables are visible in the source file where they are defined.

Register: Important and frequently used variables may be allocated storage in one of the available CPU registers for faster access (see memory hierarchy). Note that only integers or characters are allowed to be a register variable. A pointer (being an address can also be stored in a register)

Use of static variable

<to be completed shortly>

Use of the register variable

<to be completed shortly>

Control Flow

By now it is understood that in a program the flow of execution is controlled by the type of statements used. In normal cases, this flow is typically sequential. For all practical reasons this sequential flow is not always maintained and we take decisions at certain points not to execute the next instruction but branch elsewhere and continue execution sequentially from this new point. We have already encountered such statements. For example in IF statement we have branching options to break the basic sequential nature. In loop statements, also, we branch from the end of the loop to the beginning. Let us see the other statements that changes this control flow.

Control Flow contd.

Loop statement: do .. while.

while statement checks the condition at the beginning; so the body of the loop may not be executed at all. In some cases it is needed that the body of the loop should be entered at least once. Such a need can be catered through do … while statement as shown in the figure.

```
void main()
{   int a;
    scanf("%d", &a);
    do {
     r = a % 2;
     printf("%c", r);
     a = a / 2;
     }
     while ( a > 0);
}
/* Note that this program converts a decimal number a into binary  in reverse order */
```

break

In order to make an exit from a loop we may use break; e.g.; for (i=0; i < 100; i++){

```
        <stmt>
            :
        if ( <exp>)
            break;

        <stmt>
    }
```

This loop body which would otherwise be executed 100 times. It breaks if the condition is true (break is executed).

continue

Continue is used to bypass a group of statements in a loop. e.g;

```
for (i=0; i < 100; i++){
<block1>;
if (<exp>)
    continue;
<block2>;
}
```

Here, block2 will be bypassed and loop will be started from the beginning with the next value of i if the condition gets satisfied; i.e., continue is executed.

Multiway branch

This is achieved by using switch statement: e.g.,

```
while (( c = getchar()) != EOF){
    switch c {
        case 0:
        case 1: <block1>;
                break;
        case 2: <block2>;
                break;
        default: <block3>;
                break;
    }
}
```

Suppose we are reading the input and for c being the character 0 or 1 we would like to execute instructions within block1. If c is 2 we would like to execute the instructions within block2. For all other values of c we would take the default action i.e., execute the instruction inside block3. Note that the last break is not mandatory.

goto

Unconditional branch from one point to another is an unwarranted feature of any modular program and it is generally avoided. However, it can be the only way-out if you need to come out of a deeply nested block. e. g.;

```
{<b>;
   {<b>; {<b>; {goto getOut;}
    <b>; }<b>;}<b>;
};
getOut: <b>;  /* getOut is considered as a symbolic label
*/
```