Notes on C language

From: Prof Saroj Kaushik

CSE dept, IIT Delhi

Structure of C program

```
#include <stdio.h>
        /* Include files for input/output functions*/
#define const_name value
        /* constant declaration if required */
main() /* Main function */
{               /* each declarations and statements are
                separated by semi colon */
        declarations
                /* variables; arrays; records;
                function declarations etc */
        statements
                }
function definitions
```

Compiler Directives

#include statements

- used to include the header file for input/output ***stdio.h***, the standard mathematics library ***math.h*** etc.
- These files are enclosed within < >

#define

- helps in defining constant symbol.

Example

```
#include <stdio.h>
#define i 6
main()
{  /* integer declaration */
   int x, y;
   /* Assignment statements */
   x=7;
   y= i + x;
   /* output statement */
   printf("%d\n", y);}
```

Data Types

- **Standard**:
  - int, float, char, double
- **User defined datatypes**:
  - arrays,
  - structures, pointers,
  - enumerated datatype etc.

Declaration

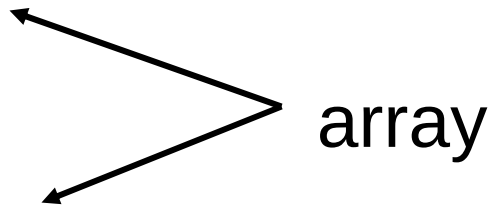
Form of Declaration:

type *list of variables;*

/* each separated by , and finally terminated by ; */

Examples:

- **int** x, y, z;
- **float** p, q[3][4];



- **char** name[20];
- **char** ch = ‘A’; /* character is enclosed within ‘ ’*/

Arithmetic Expression

- An expression is a combination of variables, constants and operators written according to the syntax of C language.
- Every expression evaluates to a value of a certain type that can be assigned to a variable.

Precedence in Arithmetic Operators

- An arithmetic expression without parenthesis will be evaluated from left to right using the rules of precedence of operators.
- There are two distinct priority levels of arithmetic operators in C.

High priority * / %

Low priority + -

Rules for evaluation of an expression

- When Parenthesis are used, the expressions within parenthesis assume highest priority.
- Parenthesized sub expression left to right are evaluated.
- If parenthesis are nested, the evaluation begins with the innermost sub expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- The associability rule is applied when two or more operators of the same precedence level appear in the sub expression.

Operator precedence and associativity

- Each operator in C has a precedence associated with it.
- The precedence is used to determine how an expression involving more than one operator is evaluated.
- There are distinct levels of precedence and an operator may belong to one of these levels.
- The operators of higher precedence are evaluated first.
- The operators of same precedence are evaluated from right to left or from left to right depending on the level.
- This is known as associativity property of an operator.

Examples

x + y * z / 2 + p

x + (y * z) / 2 + p

x + ((y * z) / 2) + p

(x + ((y * z) / 2)) + p

((x + ((y * z) / 2)) + p)

x + y - z / 2 * p

(x + y) - z / 2 * p

(x + y) – (z / 2) * p

(x + y) – ((z / 2) * p)

((x + y) – ((z / 2) * p))

Type conversions in expressions

Implicit type conversion

- C permits mixing of constants and variables of different types in an expression.
- C automatically converts any intermediate values to the proper type so that the expression can be evaluated without loosing any significance.
- This automatic type conversion is know as implicit type conversion
- During evaluation it adheres to very strict rules and type conversion.
- If the operands are of different types the lower type is automatically converted to the higher type before the operation proceeds. The result is of higher type.

Conversion rules

1. If one operand is long double, the other will be converted to long double and result will be long double.
2. If one operand is double, the other will be converted to double and result will be double.
3. If one operand is float, the other will be converted to float and result will be float.
4. If one of the operand is unsigned long int, the other will be converted into unsigned long int and result will be unsigned long int.
5. If one of the operand is long int, the other will be converted to long int and the result will be long int.
6. If one operand is unsigned int the other will be converted to unsigned int and the result will be unsigned int.

Explicit Conversion

- Many times there may arise a situation where we want to force a type conversion in a way that is different from automatic conversion.
- Consider for example the calculation of number of female and male students in a class

$$\text{Ratio} = \frac{\text{female\_students}}{\text{male\_students}}$$

- Since if female_students and male_students are declared as integers, the decimal part will be rounded off and its ratio will represent a wrong figure.

- This problem can be solved by converting locally one of the variables to the floating point as shown below.

  Ratio = (float) female_students /
  male_students

- The operator float converts the female_students to floating point for the purpose of evaluation of the expression.
- Then using the rule of automatic conversion, the division is performed by floating point mode, thus retaining the fractional part of the result.
- The process of such a local conversion is known as explicit conversion or casting a value.
- The general form is (type_name) expression

Arithmetic Expression

- x = x + 2 ⟷ x += 2
- i = i +1 ⟷ i++ or ++i

// the value of x is added with the value of i

after incrementing it by 1	then i is incremented by 1
x +(++i);	x + (i++);

after decreasing it by 1	then i is decreased by 1.
x +(--i);	x = x +(i--);

Conditional Expression

```
exp ? exp1 : exp2
```

- An expression **exp** is evaluated and
  - if the value is nonzero (or true - represented by 1) then expression **exp1** is the final value
  - otherwise **exp2** is the final value of entire expression.

Logical Operators

&&	→	AND
||	→	OR
!	→	NOT

Relational Operators

==	→	equality
!=	→	Not equal to
<	→	less than
<=	→	less than equal to
>	→	greater than
>=	→	greater than equal to

Bitwise operations

&	→	bitwise AND
|	→	bitwise inclusive OR
^	→	bitwise exclusive OR
<<	→	left shift
>>	→	right shift
~	→	One's complement

Basic Statements

- Assignment statement

  x = expression;

- Compound statement

  {s1; s2;…. };

  – Collection of statements, each separated by semi colon and enclosed in brackets

- Multiple lines comments are enclosed within /* comments */
- Single line comment can be preceded by //

Conditional statements

- **if (cond) statement;**
- **if (cond) s1 else s2**;
  - Here **cond** is a boolean condition which can have non zero value representing true and 0 representing false.
  - Statement may be simple or compund.

For statement

```
for (i = m1; i <= m2; i+=m3)
    { body };
```

- Here m1 : initial value;
  m2 : maximum value of I
  m3 : increment (positive or negative)

- body → sequence of statements.

Loop statements

- While statement

```
while (cond)
{ body };
```

- Do-while statement

```
do
        {body }
while cond;
```

Switch statement

```
switch (exp)
    {       case v1 : s1 ; break;
            case v2 : s2 ; break;
                 ⋮
            case vn : sn ; break;
            default  :  s    ⟵  optional
    }
```

– If the value of **exp** is vj then sj is executed and switch statement is exited using break statement.

– Execution always starts from 1 to last.

Input/Output statement

/* reads single character and stores in character variable x */

x = getchar();

/* prints single character stored in x*/

putchar(x);

/* the following functions are in standard file named **stdio.h */**

scanf(control, v1, v2, ..);

printf(control, e1,e2,...);

- Control in input/output

  control = "seq of format descriptor“

Format descriptor

Description	Meaning
%d	a decimal integer
%o	a octal integer
%x	a hexadecimal integer
%c	a single character
%s	a character string
%f	a decimal number (float or double)
\n	skip to new line

Examples:

- printf(**"%4d%7.2f\n%c\n"**, x, y, z)
- printf(**“%c %d %f”,** ch, i, x);
- scanf(**"%4d%8.2f \n**", &x, &y)
- scanf(**“%c %d %f**”, &ch, &i, &x);
  - Here & represents memory addresses

Arrays

- **Single dimensional Array**
  - Arrays in C are defined as:

    int numbers[50];
  - In C Array subscripts start at **0** and end one less than the array size whereas in other languages like fortran, pascal it starts from 1.
  - For example, in the above case valid subscripts range from 0 to 49.
  - Elements can be accessed in the following ways:-

    numbers[2] = 100; x = numbers[2];

- Multi-dimensional arrays can be defined as follows:

  int x[50][50]; // for two dimensions
- X is an array with 50 rows and 50 columns
- Elements can be accessed in the following ways:

  y=x[2][3];
- For further dimensions simply add more [ ]:

  int x[50][50][40][30].....[50];

Strings

- In C, Strings are defined as arrays of characters.
  - For example, the following defines a string of 50 characters: char name[50];
- C has no string handling facilities built in and so the following assignments are **illegal**:

```
char fn[10],ln[10],fulln[20];
fn= "Arnold";
ln= "Schwarznegger";
fulln= "Mr"+fn +ln;
```

- However, there is a special library of string handling routines **<string.h>** which may be included in header file and then various string operations can be used.
- String is enclosed in “ “.
  - Printf(“Well done”);’
- To print a string we use **printf** with a special **%s** control character:

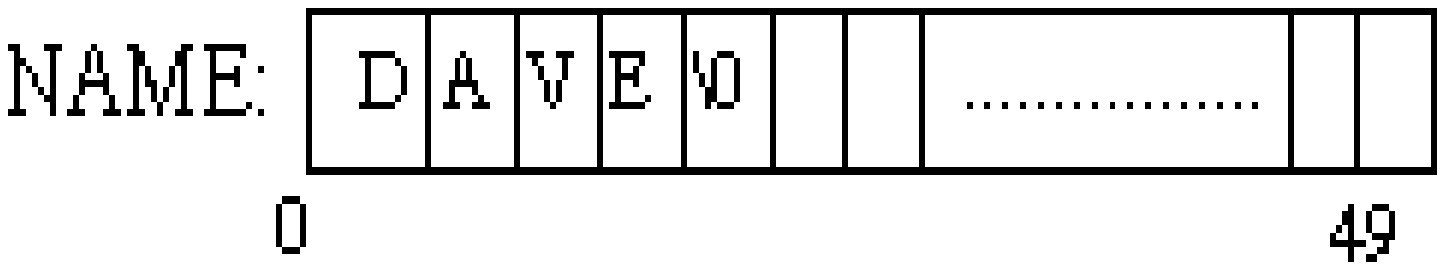
```
printf(``%s",name);
```

  - **NOTE:** We just need to give the name of the string.

- In order to allow variable length strings the 0 character is used to indicate the end of a string.
- So if we have a following declaration

  char name[50];
- Initialization can be done at the declaration time as follows:

  char name[50] = “DAVE”;
- The contents will look like:



String Handling Functions

- Include <string.h> as a header file. The following functions are available for use.
- Concatenate two strings: **strcat(s1, s2)**
- Compare two strings : **strcmp(s1, s2)**
- Length of string : **strlen(s)**
- Copy one string over other: **strcpy(s1, s2)**
  - Here contents of s2 are copied to s1
- Locating substring: strstr(s1,s2)
  - Gives the position of s1 in s2

Structure in C

- A structure in C is a collection of items of different types.
- The main use of structures is to conveniently treat such collection as a unit.
- For example:

```
struct employee
        {       char name[50];
                char sex;
                float salary;
        };
```

- The following declaration defines a variable **xyz** of struct type.

  struct empolyee xyz;

- Variables can also be declared between } and ; of a struct declaration, ***i.e.***:

```
struct employee
{       char name[50];
        char sex;
        float salary;
} xyz;
```

- struct variable can be pre-initialized at declaration:

```
struct employee
{       char name[50];
        char sex;
        float salary;
} xyz = {"john", 'm', 20000.50};
```

- To access a member (or field) of a struct, C provides dot (.) operator.
- For example,
  - xyz . sex ; xyz . salary; xyz . name

User Defined Data Types

- Enumerated Types
  - It contains a list of constants that can be addressed in integer values.
- We can declare types as follows.

  enum days {MONDAY, TUESDAY, ..., SUNDAY};
- Variables of enumerated type are defined as follows:

  enum days week1, week2;

  where **week1** and **week2** are variables

Possible uses of enumerated constants

- Enumerated constants can be assigned to variable of that type

  week1 = MONDAY;

- Conditional expression can be formed

  If (week1 == week2) ….

  if (week1 != TUESDAY) …

- Can be used in switch or for statement.

- Similar to arrays, first enumerated name has index value 0.
  - So MONDAY has value 0,
  - TUESDAY value1, and so on.
- We can also override the 0 start value as follows:

```
enum months {JAN = 1, FEB, MAR, ..., DEC};
```

  - Here it is implied that FEB = 2 and so on

```
enum colors {RED, BLUE, GREEN=5, WHITE,
                                    PINK=9};
```

  - Here RED=1, BLUE=2, GREEN=5, WHITE=6, PINK=9

```
#include <stdio.h>
main()
{
enum Color {RED=5, YELLOW, GREEN=4,
                        BLUE};
printf("RED = %d\n", RED);
printf("YELLOW = %d\n", YELLOW);
printf("GREEN = %d\n", GREEN);
printf("BLUE = %d\n", BLUE);
}
```

Output:

```
RED = 5
YELLOW = 6
GREEN = 4
BLUE = 5
```

Type Definitions

- We can give a name to enum colors as COLOR by using typedef as follows:

```
typedef enum colors COLOR;
COLOR x, y, z;
x = RED;
y = BLUE;
```

- Now, every time the compiler sees COLOR, it'll know that you mean **enum colors**.
- We can also define user named data type for even existing primitive types:

```
typedef int integer;
typedef bool  boolean;
```

- ***typedef*** can also be used with structures to creates a new type.
- Example:

```
typedef struct employee
{       char name[50];
        char sex;
        float salary;
} emp_type xyz ={"john", 'm', 2000.50};
```

- **emp_type** is new data type of **struct** employee type and can be initialized as usual:
- It can be now used for declaring variables similar to primitive data types are used.

- Examples:

  emp_type x, y, z

  – Here x, y and z are variables of type emp_type which are structures themselves.

  emp_type emp[100];

  – Here emp is an array of 100 elements with each element of type emp_type.

- Both declarations given below are same.

  struct employee x, y, z;

  emp_type x, y, z;

Unions

- A *union* is an object similar to a structure except that all of its members start at the same location in memory.
- A union variable can represent the value of only one of its members at a time.
- So an union is a variable which may hold (at different times) objects of different sizes and types.
- Example:

```
union number
        {       short shortnumber;
                long longnumber;
                double floatnumber;
        } anumber
```

- It defines a union called **number** and an instance of it called **anumber**.
- Members can be accessed in the following way:

```
printf("%d\n",anumber.longnumber);
```

- This clearly displays the value of longnumber.
- When C compiler is allocating memory for unions, it will always reserve enough room for the largest member
  - (in the above example this is 8 bytes for the double).

- Example:

```
union u_t
        {       char a;
                short b;
                int c;
        };
union u_t  x;
x.a = ‘B’;
printf("%c\n", x.a);
```

Output is: B

- In order that the program can keep track of the type of union variable being used, it is embedded in a structure and a variable which flags the union type.
- For example:

```
typedef struct { int maxpassengers; } jet;
typedef struct { int liftcapacity;} helicopter;
typedef struct { int maxpayload; } cargoplane;
typedef union
   { jet j; helicopter h; cargoplane c; } aircraft;
typedef struct
        {       aircrafttype kind;      int speed;
                aircraft description;   } an_aircraft;
```

Function

- C provides functions which are again similar in most languages.
- One difference is that C regards **main()** as a function.
- The form of a C function is as follows:

```
type fun_name(parameter along with type)
{       local declarations;
        body;
}
```

- **type :** is the type of value returned by the function and can be basic type or user defined.

- **return** statement is used in the body of a function to pass the result back to the calling program.
- Example: Write function to find the average of two integers:

```
float findaverage(float a, float b)
{       float average;
        average=(a+b)/2;
        return(average);
}
```

- We would ***call*** the function as follows:

```
result=findaverage(6,23);
```

```
#include <stdio.h>
main()
{       int i, x;
        int power (x, n);  ← function declaration
        for (i =0; i < 10; ++i)
        {       x = power(2, i);
                printf("%d%d\n", i, x);   }
}
int power(int x, n)   ←  function definition
{       int i, p;
        p = 1;
        for (i =1; i <=n; i++) p = p*x;
        return (p);   }
```

void functions

- The void function provides a way of not returning any value through function name
- Here return statement is not used:

```
void squares()
{       int i;
        for (i=1;i<10;i++);
        printf("%d\n",i*i);
}
```

- In the main function we call it as follows:

```
main( )
{ squares( ); }
```

Parameter Passing

- Default parameter passing is by value.
  - The values of actual parameters are copied in formal parameters.
  - The change is not visible in the calling program.

```
int sqsum(int a, b)
{       int sum;
        a=a*a;  b= b*b;
        sum = a + b;
        return(sum);
}
```

```
main()
{       int i, x, y,s;
        int sqsum (a,b);
        x = 5;  y = 7;
        s = sqsum(x,y);
        printf("%d%d%d\n“
        , x,y,s); }
```

- Another mechanism is to call by reference.
- It can be achieved by passing addresses of actual parameters to formal parameters.
- For such case variables in formal parameter list are represented as pointers.
- For writing functions where call by reference is to be achieved then **Void** type is used.
- In this case the function will not returning any value.
- Note that return statement is not.
- Changes to formal parameters will be visible in actual parameters of calling program.

Example:

```
void swap (int *p,*q)  ⟵ call by reference
{       int t;
        t = *p;
        *p = *q;
        *q = t;
}
```

- Corresponding call statement

```
    x = 4;
    y = 5;
swap(&x, &y);  ⟵   addresses are passed
```

Functions and Arrays

- Single dimensional arrays can be passed to functions as follows:

```
float findaverage(int size,float list[])
{       int i;
        float sum=0.0;
        for (i=0; i<size; i++) sum+=list[i];
        return(sum/size);
}
```

- Here the declaration float list[] tells C compiler that **list** is an array of float type.
- It should be noted that dimension of array is not specified when it is a ***parameter*** of a function.

- Multi-dimensional arrays can be passed to functions as follows:

```
void printtable(int xsize,int ysize, float table[][5])
{       int x,y;
        for (x=0; x<xsize; x++)
        {       for (y=0; y<ysize;y++)
                printf("\t%f",table[x][y]);
                printf("\n");
        }
}
```

- Here float table[][5] tells C compiler that **table** is an array of dimension N X 5 of float.
- Note we must specify the second (and subsequent) dimension of the array BUT not the first dimension.