An Overview of C

Balhans Jayaswal
Computer Division
BARC

Topics Covered

General .

Level - Case - Structured - Format - Terminator - Continuation - Comments - Idnetifiers - Files.

Basic data types

Integers/Characters/Boolean - Real - Arrays/Strings/Pointers.

Variables

extern - static - auto - register - const - volatile - const volatile - typedef.

Operators

Arithmetic - Relational - Boolean - Bitwise - Assignment - Ternary logic - sizeof - Typecast.

Expressions

Simple expressions - Compound expressions - Constant expressions.

Topics Covered

Statements

Null statement - Simple (Terminated expressions, Function calls) - Control statements - Compound statements (blocks).

Functions

Call by value - Declaration - Definition - Usage - Header files - Forcing a call by reference by using data pointers - Function pointers.

User defined datatypes …. enum - struct - union - Objects and Members.

Preprocessor directives … Include file - Define macros - Conditional compilation - Extensions using #pragma.

C – Library stdlib - stdio - io - ctype - string - mem - alloc, etc.

General

Level
Case
Structured
Format
Terminator
Continuation
Comments
Idnetifiers
Files.

Features of C

Level

It is an intermediate level language. The language enjoys a great success as an educational tool for general programming at the school and graduate level.

Case

It is case sensitive. As a convention, all the keywords are in lowercase. In C the variables defined as Temp, temp and TEMP are all different.

Structured

C facilitates structured programming. It has powerful control structures, and allows user to create own data structures.

Features of C

Format

It has free format. Statements can begin from anywhere, can be broken into any number of lines, till terminated with semicolon.

```
a = 5 ;
      or
a   =   5
                    ;
```

Terminator

Semicolon is used as statement terminator. Eg. See above.

Continuation

A character string or a macro cannot simply be broken into several lines. Backslash as last character specifies continuation to next line.

```
"This is a string"      "This is split \
                                into two lines"
```

Comments in C

Comments are useful in a program to keep a track of what the various parts are intended to achieve.

It greatly aids in understanding of the program by the developer and others who might want to make modifications.

Comments in C are to be enclosed inside /* and */ Comments can appear anywhere, even embedded inside expression, but not embedded inside identifier names. Nesting is not allowed.

```
/* This is a comment */
 /*
Several lines
                                        of comments
          a  /*embedded*/  =  5 ;          */
```

Identifiers and Files

Identifiers

These are names of data variables, functions, macros, etc.

These begin with: underscore A…..Z a…..z

These continue with: underscore A…..Z a…..z 0…..9

Length can be upto 32 characters. Beyond 32, characters are ignored.

Files

Source code files -

Have code and data definitions - End with .c

Source header files -

Have code and data declarations - End with .h

Compilation command:

cc, gcc, tcc, …..

Basic Data Types

The various data types allowed in C make it quite powerful programming language. Basic data types in C are

- Integers – signed, unsigned, short, long
- Characters
- Boolean – true or false
- Reals – float, double, long double
- Arrays
- Strings – arrays of characters
- Pointers
- Struct
- Unions

Integers/Characters/Boolean

Sizes:	1byte	2bytes	4bytes	1word int
Names:	*char*	*short*	*long*	(same as *short unsigned* or *long*)
data range:	0…255	0…65535 (64K)	0…4billion (4G)	
Signed Data range:	-128… +127	-32768… +32767 (±32K)	-2billion… +2billion (±2G)	

INTERPRETATION

Integer interpretation : binary number, 2's compliment.

Character interpretation : see least significant byte as ASCII code.

Boolean interpretation : if any bit 1 then TRUE, else FALSE.

Result of Boolean expression : if TRUE then 1, else 0.

Eg. 'A' + 10 + (5<6) → 65 + 10 + (1) → 76

Integers/Characters/Boolean

CONSTANTS:
Integer constants : 65 = 0201 = 0x41, 0H, 0L, 0U, 0LU
Character constants : 'A' = '\201' = '\x41', 'ABCD'
Boolean constants : 0, 1
With meta character \ : '\a' '\b' '\f' '\n' '\r' '\t' '\v' '\\' '\''

DEMOTION : by truncation of higher bytes.
PROMOTION:
unsigned integer: pad higher bits with 0.
Signed integer: pad higher bits with sign bit (highest bit).

CAUTION: promotion of signed to *unsigned*

```
char    a=255 ;                    /*signed a = -1*/
unsigned short      b=a ;          /*b=65535*/
unsigned short      c=(unsigned char)a; /*c=255*/
```

Reals

Sizes	:	4byte		8bytes	8/10/16 bytes
Names	:	*float*		*double*	*long double*
Range	:	±1E±38		±1E±308	
Precision	:	6 digits		15 digits	

CONSTANTS

Double	: 0.	-.1	1E0	-.2e3	45.6e-7
Float	: 0F	-.1F	1E0F	-.2e3F	45.6e-7F
long double	: 0LF	……			

DEMOTION : convert to single precision.
PROMOTION : convert to double precision.
DEFAULT TYPE : *double*

Arrays

ARRAYS:

```
int   A[3] ;   /*A[0]...A[2]*/
int   A[4][3] ;          /*A[0][0]...A[0][2] ... A[3][2]*/
int   A[2][3][4][5][6][7]; /*array of array of...*/
```

INITIALISED ARRAYS:

```
int   A[3] = {10, 20, 30};
int   A[ ]  = {10, 20, 30};   /*same as above*/
int   A[2][3] = {1,2,3,  10,20,30};
int   A[2][3] = {{1}, {10,20,30}};
int   A[][3]  = {{1}, {10,20,30}};/*same as above*/
```

Strings

Terminator : '\0' (Null character, ASCII code = 0)

Constants :	""	"a"	"Test"	"Test string"	" \" "
Lengths	:0	1	4	11	
Sizes	:1	2	5	12	

Variables:

```
char   s[5] = {'T', 'e', 's', 't', 0};
char   s[5] = "Test" ;          /*Same as above*/
char   s[]  = "Test" ;          /*same as above*/
char   s[40]= "Test" ;          /*size=40,Length=4*/
char   s[40];   strcpy(s, "Test");
```

String Functions

Functions:

Declared in C standard library header file: string.h

strcpy(s, ”Test”);	……	copy string to s
strcat(s, ”ing”);	……	append string in s
int Len=strlen(s);	……	get length of string
int Siz=sizeof(s);	……	sizeof is an operator
strcmp(s1, s2);	……	copy string to s
strchr(‘e’, s);	……	find character inside s
strstr(“es”, s);	……	find substring inside s
strncpy(s1, s2, n);	……	copy atmost n characters

Pointers

These are addresses of data variables and functions. Though the concept of pointers has resulted in simple and elegant code, there is a tendency to avoid the excessive use of pointers as they lead to more complex programming codes and not easy during debugging.

Constants:

- NULL defined in header files stdlib.h and stdio.h
- String constants such as “Test” are pointer constants.
- Operator & before a variable gives a pointer constant.

Pointers

Array variable without [ ] is a pointer constant.

```
short    i, A[3], B[10][3] ;
```

&i	…	address of i
&A[0], A	…	address of A[0]
&A[i], A+I	…	address of A[i]
&B[0][0], B[0]	…	address of B[0] [0]
&B[i][0], B[i]	…	address of B[i] [0]
&B[i][j], B[i]+j	…	address of B[i] [j]
A, B[i]	…	address of storage of size 1 variable
B	…	address of storage of size 3 variables
A+1, B[i]+1	…	address is offset by 1 variable
B+1	…	address is offset by 3 variables

where size of each variable of short integer type is 2.

Pointers

VARIABLES:

char *p ; … pointer to 1 byte storage

short *p ; … pointer to 2 byte storage

long *p ; … pointer to 4 byte storage for integer

float *p ; … pointer to 4 byte storage for real

double *p ; … pointer to 8 byte storage

Each pointer is of size 4 bytes for storing addresses in the range 0 … 4GB

Arrays and Pointers

ARRAYS:

char *A[6]; 6 pointers pointing to 1 byte storages

char(*A)[6]; 1 pointer pointing to a 6 byte storage

char(*A[2])[6]; 2 pointers pointing to 6 byte storages

Pointer Arithmetic and Comparisons

POINTER ARITHMETIC:

Pointer ± integer	address ± integer*PointedSize
Pointer1 - Pointer2	(address difference) /PointedSize

++Pointer Pointer++ --Pointer Pointer--

where PointedSize is size of the pointed storage.

POINTER COMPARISONS:

Pointer1 == Pointer2, Pointer1 < Pointer2

Variables

- extern
- static
- auto
- register
- const
- volatile
- const volatile
- typedef.

extern

Syntax : ***extern*** **datatype var_list** ;
Location : anywhere.
Scope : global across modules.
Residence : DATA segment or BSS.
Lifetime : Till program terminates.

Note:
If unintialised variable : it is a declaration of variable.
Res=BSS
If initialised variable : it is a definition of variable.
Res=DATA
If located outside function, and *extern* is dropped, statement becomes definition of global var_list.

static

Syntax	:	*static* datatype var_list ;
Location	:	(a) Outside function.
		(b) Inside function/block.
Scope	:	(a) Within module.
		(b) Within function/block.
Residence:		DATA segment or BSS.
Lifetime	:	Till program terminates

auto

Syntax : *auto* datatype var_list ;

Location : Inside function/block.
Scope : Within function/block.
Residence : STACK.
Lifetime : Till control exits the block.

Note:

If *auto* is dropped : it is assumed to be *auto*.
If datatype is dropped: it is assumed to be *int*.

register

Syntax	:	***register*** **datatype var_list** ;
Location	:	Inside function.
Scope	:	Within function.
Residence	:	Inside registers, but maybe on STACK.
Lifetime	:	Till control exits the block.

Note:

Declared variable cannot be subjected to & (address-of).

It encourages code optimisation in fetching data from memory.

const

Syntax 1: ***const*** **datatype var_list ;**
Syntax 2: **datatype *const* var_list ;** /*same as above*/
Syntax 3: ***const*** **datatype * *const* var ;**

Location: (a) Outside function. (b) Inside function/block.

Scope: (a) Within module. (b) Within function/block.

Residence: (a) RDATA segment. (b) STACK

Lifetime: (a) Till program terminates. (b) Till block is exited.

Note:
In Syntax 3, first *const* forbids *var=ex, and second forbids var=ex.
It encourages optimisation in fetching data from memory.

volatile

Syntax 1: ***volatile*** **datatype var_list ;**
Syntax 2: **datatype *volatile* var_list ;** /*same as above*/
Syntax 3: ***volatile*** **datatype * *volatile* var ;**

Location	: (a) Outside function.	(b) Inside function/block.
Scope	: (a) Within module.	(b) Within function/block.
Residence	: (a) DATA segment.	(b) STACK
Lifetime	: (a) Till program terminates.	(b) Till block is exited.

Note: It discourages optimisation in fetching data from memory

Typedef

Syntax : ***typedef datatype_declaration***

The ‘datatype_declaration’ looks just like variable list declaration. But instead of varaibles (or objects) it declares new datatypes.e.g.

```
char  Str[10] ;
struct
{
        int a,b,c ;
} Rec, *Ptr ;
```

```
typedef char  Str_t[10] ;
typedef struct
{
        int a,b,c ;
} Rec_t, *Ptr_t ;
        Str_t   Str ;
        Rec_t  Rec ;
        Ptr_t   Ptr ;
```

Operators

- Arithmetic
- Relational
- Boolean
- Bitwise
- Assignment
- Ternary logic
- sizeof
- Typecast.

Arithmetic

Operators : + - * / % ++ --

Modulo : ±a % b result: -(b-1)…0…(b-1)

a, b must be integers. If reals, use math function fmod(). If a is negative, result is negative. Sign of b is ignored.

Pre-Increment: int a=5, b = ++a ; /*a=6, b=6*/

Post-Increment: int a=5, b = a++ ; /*b=5, a=6*/

Pre-Decrement: int a=5, b = --a ; /*a=4, b=4*/

Post-Decrement: int a=5, b = a-- ; /*b=5, a=4*/

Relational and Logical

Relational Operators	: **==**	**!=**	**<**	**<=**	**>**	**>=**
Result:	if TRUE then 1, else 0					
LOGICAL Operators:	**&&**		**||**		**!**	
Operations:	and		or		not	
Result:	if TRUE then 1, else 0					
Ternary Logic Operator	: **?**		:			
Usage	: a?b:c	…..results in b if a is true, else results in c.				

Bitwise

Operators	:	&	|	~ ^
Operations	:	and	or	not exclusive or

Shift operators: << >>

Padding : Pad with 0.
If unsigned, Pad with 0, else with sign bit.

Assignment

Operators:

=	+=	-=	*=	/=	%=
&=	|=	^=	<<=		>>=

Usage: a+=b is same as a=a+b, etc.

Result : The value assigned is also the result of the expression.

sizeof

Operator : **sizeof**
Returns : size in bytes of memory needed for storing data.

	sizeof(char)	……results in 1
	sizeof(short)	……results in 2
	sizeof(char*)	……results in 4
double a ;	**sizeof a**	……results in 8
double a[10];	**sizeof a**	……results in 80
	sizeof'A'	……results in 2 or 4
	sizeof ""	……results in 1
	sizeof "Test"	……results in 5
double sub();	**sizeof sub()**	……results in 8

This size of resulting data is determined at compilation time without actually computing the expression.

Typecast

Operators : (char) (short) (char*) etc.

Operation : Change data of one type to another type.

AUTOMATIC & EXPLICIT TYPECAST

AUTOMATIC TYPECAST

In expressions:

- char, short → int
- unsigned char, unsigned short → unsigned int
- float → double

Induced by operand:

- (char, short, int) op (char, short, int) → (int) op (int)
- If any is unsigned → (unsigned) op (unsigned)
- char, short, int) op (long) → (long) op (long)
- If any is unsigned → (unsigned long) op (unsigned long)
- (char,short,int,long) op (float,double) → (double) op (double)

EXPLICIT TYPECAST:

short a ; (long)a ……Pad higher bits with sign bit.

long a ; (float)a ……..Convert to floating point format.

Expressions

- Simple expressions
- Compound expressions
- Constant expressions

Simple expressions

Anything that results in a data value is an expression. E.g.

5	.....	results in 5
A	.....	results in value of A
A + 5	.....	results in value of A + 5
A < 5	.....	results in 0 or 1
A ? 5 : 6	.....	results in 5 or 6
A = 5	.....	results in 5
Sub()	.....	results in data returned by Sub()

Types of simple expressions

Arithmetic expression	:	results in integer or real data value.
Logical expression	:	results in 0 or 1.
Pointer expression	:	results in a storage address value.

Compound expressions

Syntax: **(ex1, ex2, … exn)**

Commas are used to separate the sub-expressions. Brackets can be discarded if other punctuations of C appear there.

Computation is done from left to right.

Result of compound expression is that of its last sub-expression.

Eg.

b = (a=5, a+6) ……set a=5, b=11, return value 11

if (x>y) a=1,b=2,c=3; …set values to all, or to none.

NOTE:

In C syntax, wherever an expression can appear, in its place, a compound expression can appear.

Constant expressions

These are expressions composed entirely of constants (numbers).

These are evaluated at compile time.

In C syntax wherever a constant can appear, in its place, a constant expression of same datatype can appear.

Eg.

```
double  a=1.25*2, b[10], [sizeof(b)/2+1];
```

is same as: `double  a=2.5,  b[10], c[41];`

Statements

- Null statement
- Simple Statements
  (Terminated expressions, Function calls)
- Control statements
- Compound statements (blocks).

Null Statement

Null statement: ; or {}

It is a no-op statement.

Internally it provides a code address for jumping to that location.

Simple statements

Any expression terminated with a semicolon. Eg.

Expression	Statement
5	5;
A	A;
A + 5	A + 5;
A < 5	A < 5;
A ? 5 : 6	A ? 5 : 6 ;
A = 5	A = 5;
Sub()	Sub();

Control Statements

These statements provide control structures in programming.

Logic:

- ***if***
- ***switch***

Loop:

- ***for***
- ***while***
- ***do***

Jump:

- ***goto***
- ***break***
- ***continue***
- ***return***

If Statements

Syntax: *if* (ex logical) st

Syntax: *if* (ex logical) st1 *else* st2

These statements are used for switching the control of program depending on one or the other condition.

The logical expression is evaluated, one set of statements is performed if the expression is true whereas another set olf statements is performed if the codition is false. For example,

```
If (a>b)
    printf (“a is greater than b);
Else
    pritnf(“b is greater trhan a”);
```

switch statement

Syntax:

```
switch (ex Integer)
{   case  Integer
          Const1 : st11    st12    …..
          break ;
    case  Integer
          Const2 : st21    st22
          break ;
    default: st1      st2 }
```

default branch may be absent. *break* causes control to jump out of the *switch* control structure . Any or all of the statements (including *break*) may be absent,but there must be at least one statement just before the closing brace

for statement

Syntax : *for* (exInitialiser ; exLogical ; exNext) st

Syntax : *for* (exInitialiser ; exLogical ; exNext)

```
{        .....
         .....      continue ;
         .....      break ;
.....
}
```

Any or all of the expressions can be absent.
If ex Logical is absent, it is taken to be forever TRUE.
continue causes control to jump to next iteration of the loop. *break* causes control to jump out of the loop, terminating it.

while statement

```
Syntax:     while (exLogical)   st
Syntax:     while (exLogical)
            {       .....
                    .....   continue ;
                    .....   break ;
            .....
            }
```

The while condition is checked and Iteration is performed as long as exLogical is TRUE. The program comes out of the loop once the while condition is false.

do statement

Syntax: ***do*** **st** ***while*** **(exLogical) ;**

Syntax:

```
do
{       .....
        .....   continue ;
        .....   break ;
.....
}
while (exLogical) ;
```

Iteration is done as long as exLogical is TRUE. The do while statement is performed atleast once, even if the logiocal expression n is false.

goto statement

Syntax:

```
goto label ;
.....
label : st
```

```
label : st
.....
goto label ;
```

Label is an identifier that is followed by a colon and a statement.

goto cannot be used for jumping across functions.

Programmers are generally advised to avoid the use or use minimum of the goto statements.

return statement

Syntax: ***return ;***

Inside *main*() function, it causes execution to terminate. *;*

Inside other functions, it causes control to return to calling fucntion.

For terminating execution from anywhere, call function *exit*().

Compound Statements (blocks)

Syntax: { st1 st2 …… }

There can be any (including zero) number of statements inside a block.

In C syntax, wherever a statement can appear, in ts place.

A compound statement can appear.

Functions

- Call by value
- Declaration
- Definition
- Usage
- Header files
- Forcing a call by reference by using data pointers
- Function pointers.

Call by value

C supports ‘call by value’ only.

For producing a ‘call by reference’ Address of data variable is sent as actual arguments.

Pointer is used as formal parameter.

It is still a ‘call be value’ only in which the value passed is an address.

Function declaration

Syntax (old) :
datatype functionname () ;
Syntax (ANSI) :
datatype functionname (arg_datatype_list) ;

It consists of just a declarator, followed by semicolon.
If datatype is absent, it is assumed to be *int* .
If function does not return anything, its datatype should be *void* .
If in ANSI – C style, it is also called function prototype declaration.

Function - definition

Syntax (old):

datatype functionname (arg_name_list)

arg_datatype_declarations

{ st1

st2 …….

}

Syntax (ANSI):

datatype functionname (arg_declaration_list)

{ st1

st2 …….

}

It consists of function declarator with argument declarations followed by a block of statements that constitute the function body.

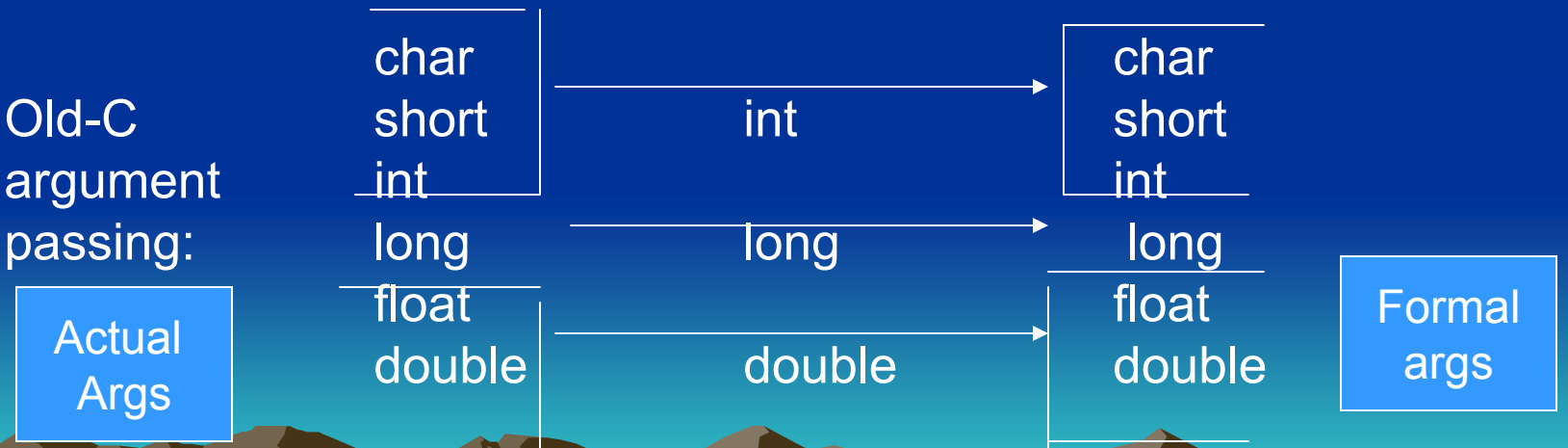
Function - usage

Syntax:

functionname (arg_value_list) …… Function call.

functionname …… Function code address.

Normally, declaration or definition should precede usage.
Old-C convention demands that arg_values be explicitly typecasted.
In ANSI – C, this typecasting is automatically, internally done.



Header files

All standard library function declarations are in standard header files.
These are kept inside standard include directory.
These need to be included before their functions can be used.

Syntax:

#include **<filename.h>** /*search standard include dir*/
#include **"filename.h"** /*search current, then std.dir*/

e.g.

```
#include <stdio.h>
main ()
{
   int c ;
   while ((c=getchar()) != EOF) putchar(c);
}
```

Forcing a call by reference by using data pointers

Eg.

CALL BY VALUE

Using Old-C convention

```
int sum();

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

Using ANSI - C convention

```
int sum (int, int);

int sum (int a, int b)
{

return a+b ;
}
```

Usage:

```
int  x=1, y=2, z ;
z = sum(x,y);                    /* z=3 */
```

FORCED CALL BY REFERENCE

FORCED CALL BY REFERENCE

Using Old-C convention

```
swap();

    swap (a, b)
int *a, *b;
{
    int  tmp=*a ;
    *a=*b, *b=tmp ;
}
```

Using ANSI - C convention

```
void swap (int*, int*);

void swap (int*a, int*b)
{

int  tmp=*a ;
        *a=*b, *b=tmp ;
}
```

Usage:

```
int  x=1, y=2 ;
swap (&x, &y);          /* x=2, y=1 */
```

Function pointers

Declaration syntax:

datatype (*Ptr) (arglist) ;

'Ptr' will store code address of a function that

Takes arglist as specified in declaration.

Returns datatype as specified in declaration.

For example,

Function: datatype Function (arglist) ;

Initialisation: Ptr = Function ;

Usage:

(*Ptr) (args) ; /*same as: Function(args) ; */

User Defined Datatypes

Topics Covered

enum

struct

union

Objects and Members

Enum type

It is used for giving names to integer constants, and their group.

Syntax 1: *enum* datatypename

```
{
        constname1 = n1,
        constname2 = n2,
        .........
} var_list ;
```

Syntax 2: *enum* datatypename var_list ;

Syntax 3: *typedef* *enum* datatypename

```
{
        constname1 = n1,
        constname2 = n2,
        .........
} another_datatypename ;
```

Syntax 4: another_datatypename var_list ;

enum

In Syntax 1 and 3, ‘datatypename’ is optional.
- If ‘datatypename’ is specified, Syntax 2 can also be used.

In Syntax 1, var_list is optional.
- If not specified, Syntax-2 will be needed for specifying it.

Initialisers n1, n2 are integer constants, and are optional.
- If n1 is missing, it is assumed to be 0.
- If n2 is missing, it is assumed to be (n1+1), and so on.

Eg.
```
enum  MyColorType
{
        BLACK,                  /* =0 */
        BLUE,                   /* =1 */
        GREEN,                  /* =2 */
        CYAN,                   /* =3 */
} ScreenColor, TextColor=CYAN ;
enum MyColorType BorderColor=GREEN, FillColor;
```

Struct type

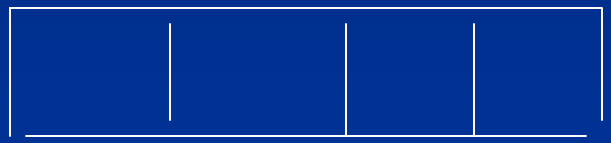
It is used for grouping data related to each other,and naming the group.

Comparison with arrays:

Array

Groups related data.

All data fields are of same size and data type.



By passing address to a function function

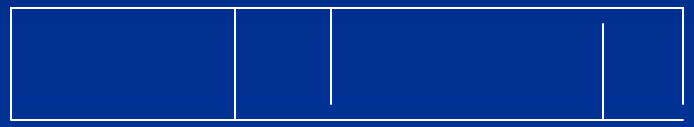
all the fields can be accessed.

The datafields are accessed by accessed by using an integer index.

Structure

Groups related data.

Data fields may differ in size and/or datatype.



By passing address to a

all the fields can be accessed.

The datafields are naming individual fields.

Syntax of struct

Syntax 1: ***struct*** **datatypename**

```
    {
        datafield_list_declaration
    } var_list ;
```

Syntax 2: ***struct*** **datatypename** **var_list ;**

Syntax 3: ***typedef*** ***struct*** **datatypename**

```
    {
        datafield_list_declaration
    } another_datatypename ;
```

Syntax 4: **another_datatypename** **var_list ;**

In Syntax 1 and 3, ‘datatypename’ is optional.
- If ‘datatypename’ is specified, Syntax 2 can also be used.

In Syntax 1, var_list is optional.
- If not specified, Syntax-2 will be needed for specifying it.

‘datafield_list_declaration’ has same syntax as variable declaration.

Syntax of Struct

Syntax 1:
struct **datatypename**
{
datafield_list_declaration
} var_list ;

Syntax 2:
struct **datatypename**
var_list ;

Syntax 3:
typedef ***struct*** **datatypename**
{
datafield_list_declaration
} another_datatypename ;

Syntax 4:
struct
another_datatypename
var_list ;

Examples of Struct

```
typedef struct  _FuelRodRec
{   shortPosX, PosY ;     /*position*/
float      Length ;
float      FuelRadius ;
float      CladThickness ;
float      CoolantThickness ;
float      FuelConductivity, FuelSpHeat ;
float      CladConductivity, CladSpHeat ;
float      CoolantHeatRemCoef ;
} FuelRodRec, FuelRodPtr ;
```

```
/*------------Object definitions--------*/
struct _FuelRodRec
FuelA[50], FuelB[60];
FuelRodRec  FuelA[50], FuelB[60];
/*Same as above*/
/*-----------Pointer definitions----------*/
struct _FuelRodRec  *Ptr ;
FuelRodRec  *Ptr ; /*Same as above*/
FuelRodPtr   Ptr ; /*Same as above*/
```

union

It is used for accessing data by typecasting to various datatypes.

Syntax 1: ***union*** **datatypename**

```
{
    datafield_list_declaration
} var_list ;
```

Syntax 2: ***union*** **datatypename** **var_list ;**

Syntax 3: ***typedef union*** **datatypename**

```
{
    datafield_list_declaration
} another_datatypename ;
```

Syntax 4: another_datatypename var_list ;

Union

In Syntax 1 and 3, ‘datatypename’ is optional.

If ‘datatypename’ is specified, Syntax 2 can also be used.

In Syntax 1, var_list is optional. If not specified, Syntax-2 will be needed for specifying it.

‘datafield_list_declaration’ has same syntax as variable declaration.

These datafields are overlapping datafields on same storage space.

Struct and Union

These datafields are overlapping datafields on same storage space.

Eg. **struct**

```
Struct  dt
{
     short s ;
     char  a, b ;
     char  c[2] ;
     }Obj, *Ptr;
```

Obj

union

```
union  dt
{
  short s ;
   char  a, b ;
   char  c[2] ;
}Obj, *Ptr ;
```

s

Obj

Objects and Members

Structure variables are also called objects, and datafields, members. In above example

dt is a datatype.
Obj is an object of that datatype.
s is a data member of that object.

Syntax for accessing object members: Obj.s Obj.c[0]
Syntax for accessing pointer members: Ptr->s Ptr->c[0]
Syntax for accessing member addresses:
&Obj.s &Ptr->c[0]

Preprocessor Directives

Topics Covered

Include file

Define macros

Conditional compilation

Extensions

using #pragma.

Include file

Syntax 1: *#include* <filepath>

Syntax 2: *#include* “filepath”

In Syntax 1, filepath is searched in the standard include directory.

In Syntax 2, filepath is searched

first in current directory,

if not found, then in the standard include directory.

The contents of the specified file are read and inserted.

Insertion can be done even in the middle of a declaration or expression.

Include File

E.g.

File= XX

File= data.dat

```
double Data[ ]=
{
#include“data.dat”
}
```

```
10,20,30,
40, 50
```

```
a
#include “xx”
5 ;
```

Include files

It is used for including header files containing function declarations. Eg:

```
#include <stdio.h>
main ()
{
   int c ;
   while ((c=getchar()) != EOF) putchar(c);
}
```

Define macros

Syntax 1: *#define* macroname substitution
Syntax 2: *#define* macroname(arg_list) substitution
Syntax 3: *#undef* macroname

In Syntax 1 and 2, ‘substitution’ is optional.
- If present, ‘macroname’ is substituted with ‘substitution’.

If ‘arg_list’ is supplied
- Opening bracket must be just after `macroname’.
- In ‘substitution’, args are also substituted.
- In ‘substitution’, avoid using each arg more than once.

Use of brackets is recommended
- To enclose the entire ‘substitution’ expression
- To enclose individual args inside ‘substitution’

Macro definition

If macro definition spills into next line, end the previous line with \ .
Avoid recursion in macro calls. *#undef* undefines a macro.

```
#define  BEGIN                {
#define  END          }
#define  PI           3.141593
#define  SUM(a,b)     ((a)+(b))
#define  DBG(f)                {if (IsDebug) \
                      printf("%f ",f);}
int  IsDebug=1 ;
main ()
BEGIN
    DBG(SUM(PI,1))          /*prints 4.141593*/
END
```

Conditional Compilation

Syntax 1: *#ifdef* macroname

```
.....
#else
.....
#endif
```

Syntax 2: *#ifndef* macroname

```
.....
#else
.....
#endif
```

Syntax 3: *#if* logical_constant_expression

```
.....
#elif  logical_constant_expression2
.....
#else
.....
#endif
```

Explained

The *#elif* and *#else* clauses are optional.

If `macroname' has been used in a *#define* statement

'*#ifdef* macroname' evaluates to TRUE.

'*#ifndef* macroname' evaluates to FALSE.

If `macroname' has been used in a *#undef* statement, or never used

'*#ifdef* macroname' evaluates to FALSE.

'*#ifndef* macroname' evaluates to TRUE.

The 'logical_constant_expression' is computed at compile time

If it results in a non-zero value, it evaluates to TRUE.

If it results in 0, it evaluates to FALSE.

If it is an undefined macro, it evaluates to FALSE.

Extensions using #pragma

Syntax: *#pragma* extension

This is used for specifying compiler specific directives

Directive to accept assembly language code.

Directive to affect warning notification of some type.

Directive to affect optimisations of various types.

Directive to affect code generation for debugging purpose.

Compiler command line options have equivalents in *#pragma* extensions.