FUNCTIONS AND POINTERS

func^{n} ⇒ Self contained block of statements that perform a coherent task of some kind.

Types predefined: Stored in library

user defined:

⇒ [redacted]

Library files ⇒ Definition.

⇒ function prototype declaration

```
xxxx functionname (int, int* );
```

↳ Return type ↳ Type of arguments to be passed.

Local declaration: func^{n} accessible only to the func^{n} in which it is declared.

Global declaration: func^{n} accessible in whole program.

⇒ function definition

```
xxxx function name (int a, int *b)
{ int c; → declaration 2
  return ( );
  ~~~~ → unreachable code.
}
```

↳ declaration 1

⇒ function call

```
c⇒ function name (a, b);
```

↳ variable to collect the returned value.

↳ passed values

NOTE :-

⇒ If a program contains only one func^n, it must be main().
⇒ In case of many func^n, one and only one func^n can be main().
⇒ Program execution always begins with main().
⇒ Each func^n is called in a sequence specified by the function calls in main().
⇒ After each func^n has done its thing or if the return statement is found, the control returns to main().
⇒ main() func^n can be called from other functions
⇒ A func^n can be called any number of times.
⇒ A func^n cannot be defined in another func^n.

USES OF FUNCTIONS

a] Avoids Rewriting.
b] Separates the code into modular functions
c] Better use of memory.

PASSING VALUES BETWEEN FUNCTIONS

⇒ The mechanism used to convey information to the function is the argument. or parameters.

passing collection

sum = calsum (a,b,c); → actual arguments
int SUM (int x, int y, int z) → formal arguments.

⇒ TYPE, ORDER & NUMBER of the actual & formal arguments must be same.
⇒ Actual & formal arguments with same name are treated differently.

RETURN

→ Transfers the control back and returns the value present in parenthesis.
⇒ Only one value can be returned.
⇒ return (c, a, b) *
⇒ return a+b; *
⇒ There can be more than one return statements (if-else)
⇒ return ; — value is returned.

⇒ Changing values of formal arguments doesn't change the value of actual arguments. (∵ a copy is passed).

SCOPE RULE OF FUNC^N

⇒ Scope of a variable is local to the func^n in which it is defined.

CALLING CONVENTION

Convention indicates :
(i) order of passing of arguments.
(ii) which ~~performs~~ func^n performs the cleanup of variables when control returns
⇒ Arguments can be passed from LTR OR RTL
⇒ STANDARD calling convention
=
cleanup of stack ⇒ called func^n.

USING LIBRARY FUNCTIONS

=> Prototype declaration helps the compiler in checking whether the values being passed and returned are as per the prototype declaration.

=> Library functions are divided into many groups and a file is provided for each grp.
.h files → declarations of lib funcns.

```
#include <stdio.h>
int main()
{ int i=10, j=20;
  printf("%d %d %d\n", i, j);
  printf("%d\n", i, j);
  return 0;
}
```

garbage value

This program gets successfully compiled b/c printf accepts variable no. of arguments & even with the mismatch the call still matches with prototype of printf present in stdio.h.

=> By default the return type of any funcn is assumed to be int.

CALL BY VALUE & CALL BY REFERENCE

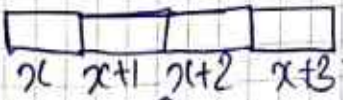
CALL BY VALUE	CALL BY REFERENCE
values of variables is passed to the funcns.	address of variables is passed to the funcns.
values are copied from AA to FA changes has no effect on values of AA	values of AA can be manipulated using addresses * MORE THAN ONE VALUE CAN BE made to return.

=> mixed call: when values of some variables & addresses of other variables are passed.

Pointers

⇒ In DOS 64KB Memory is allocated to a program ∴ the addresses can have a value in range of
[0, 65535]

⇒ [redacted]

⇒ int ⇒ 4 bytes 

→ address in sequence

⇒ Address can be printed using a format specifies %u [unsigned integers]
it has range for 0 - 65535 while %d has range of -32768 ⊖ 32767 & ∴ address can't be negative.

⇒ printf ("%u", address);

⇒ Address of operator (&) [redacted] operator

⇒ → unary operator, operand ⇒ name of variable
gives ⇒ address of number.

⇒ value at address operator/ [redacted] operator (*)

⇒ unary operator

⇒ operand ⇒ addresses

⇒ gives variable value.

⇒ *(&i) = i

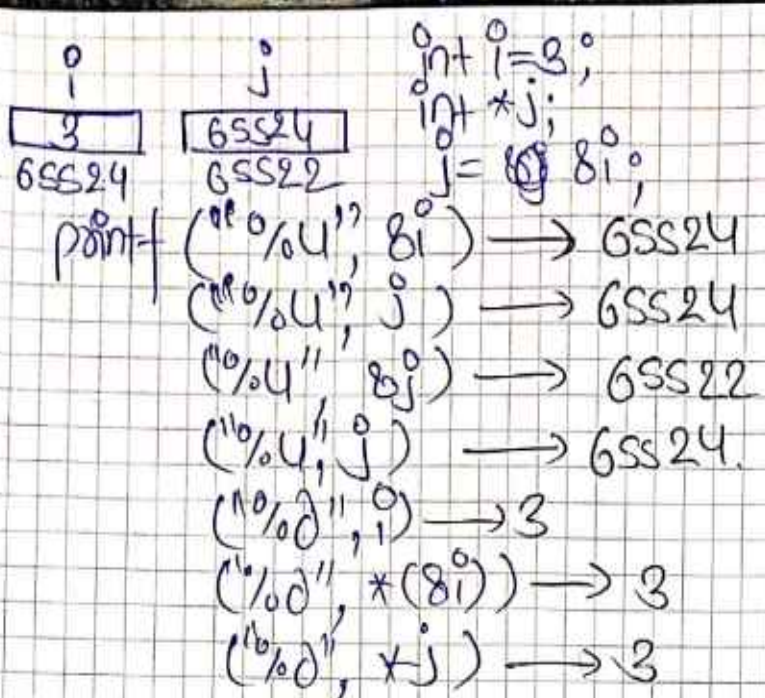
⇒ &x = 7; [redacted] addresses can be equated to any const, because they are also constants.

⇒ Declaration

int *j; [redacted] not an operator

→ pointer carries address of a variable of int type.

→ It means, that the value stored at the address contained in j is an int.



⇒ A pointer is a variable containing address of another variable.

BASE Address

⇒ The address of 1st byte / 1st memory block of a variable.

⇒ [redacted] of a variable is stored in the pointer.

⇒ Address of a particular type of a variable can be stored in same type of pointer because by base address we can't tell [redacted].

Extended concepts of pointers

⇒ level of Indirection :- 2 if a pointer variable contains the address of a pointer variable.

```
int x=5, *p, **q, ***r;
                  ↳ level of indirection
p=&x;
q=&p;
r=&q;
```

⇒ A pointer variable can store address of another variable of Indirection one less than the Indirection level of itself.

x → an int

p → pointer to an int

q → pointer to a pointer to an int

r → pointer to a pointer to a pointer to an int

```
int i=3, *j, **k;
j = &i;
k = &j;
```

%u, &i ⇒ 65524 %u, &k ⇒ 65520 %d, **k ⇒ 3
%u, j ⇒ 65524 %u, j ⇒ 65524
%u, *k ⇒ 65524 %u, k ⇒ 65522
%u, &j ⇒ 65522 %d, i ⇒ 3
%u, k ⇒ 65522 %d, *(&i) ⇒ 3
%d, *j ⇒ 3

Pointers Arithmetic

⇒ we can't add +/x/÷ 2 pointers
⇒ we can subtract, type should be same and gives the no of memory blocks of a particular type.
⇒ Integer x/÷ pointer ⇒ not allowed.
⇒ Integer +/- pointer ⇒ allowed.
⇒ pointer + 1 ⇒ address of next memory block

```
int x;
p = &x; = 1000
```

(p+1 ⇒ 1002)

⇒ [pointer + n ⇒ pointer + n(size of(type))]
⇒ [pointer 1 - pointer 2 ⇒ literal subtraction / size of data type]

Application of pointers

func^n call: function name(&a, &b);
func^n defin^n: void function name(int *x, int *y)
{
}
func^n declaration: void function name(int*, int*);

RECURSION (Circular definition)

⇒ we call the series of calls to the recursive func^n as being different invocations of rec().

Recursion and Stack

The compiler uses a data structure called Stack for implementing normal as well as recursive func^n calls.

STACK ⇒ LIFO → pop operation.
↳ push operation

```
int a=5, b=2, c;
c=add(a,b);
printf("sum=%d", c);
return 0;
}
int add(int i, int j)
{ int sum;
  sum=i+j;
  return sum;
}
```

pushing of values
b → a → printf's address → sum

popping of values
sum → address of statement where control should return → a → b.

⇒ add(): - pops sum and address.
⇒ main(): pops the two integers. [CDecl calling convention]

⇒ while writing recursive funcns you must have an if statement somewhere in the recursive func^n to force the func^n to return without recursive call being executed.

⇒ use printf func^n to trace the recursive operations

⇒ Problems which can be done with recursion can also be done with loop whereas vice-versa may not be true.

⇒ No. of calls = No of copies of func^n in RAM.

⇒ BASE CASE ⇒ Case where Recursion ends.

Adding fonctns to the library

→ write fonctn definin in .c file
→ compile & create an .obj file
→ add by issuing command. → switch
C:\>tlib maths.lib + c:\fact.obj
library filename ↳ path of .obj file
→ declare the prototype of the fonctn in a .h file
→ using fonctn #include "c:\fact.h"
→ can be deleted using - switch.

Creating Own library

⇒ define fonctns in a .c file. Donot define main().
⇒ create a .h file & declare the prototypes.
⇒ options > Applications > Library > olc
⇒ compile to create the .lib file
⇒ using fonctn #include "myfonc.h"
↳ mention path if .c & .h files are not in same directory.

⇒ project > add item > .c file > add > .lib file > add > done.

DATA TYPES REVISITED

⇒ To fully define a variable ⇒ type + storage class

Integers, long and short

⇒ 16 bit compiles ⇒ Range of integer -32768 - 32767
⇒ 32 bit compiles ⇒ -2147483648 - +2147483647
⇒ Out of two/four bytes used to store an integer, the highest bit 16th/32nd bit is used to store the sign of integer.
1 ⇒ negative
0 ⇒ positive.

Short and long integers [extends & shrinks ranges]

⇒ usually short & long integers would occupy 2 and 4 bytes respectively.
⇒ Each compiler can decide appropriate sizes depending on the OS and hardware.

Rules for shorts & longs

⇒ shorts atleast 2 bytes big
⇒ longs atleast 4 bytes big
⇒ shorts ≯ ints (4byte)
⇒ ints ≯ longs

Shorts ≤ Int ≤ longs
≥2 4 ≥4

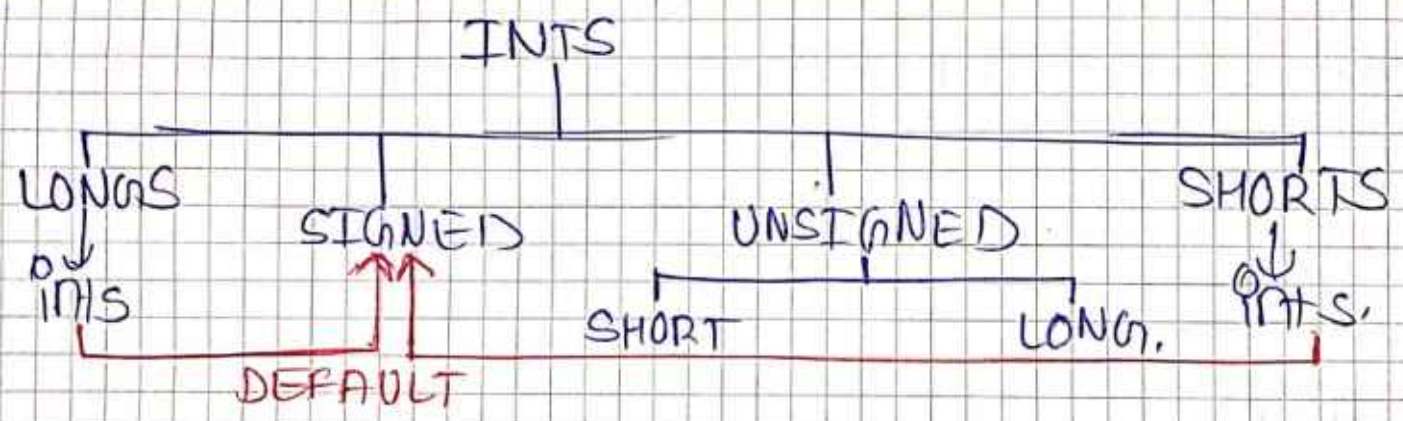
16 bit S⇒2 I⇒2 L⇒4
32bit S⇒2 I⇒4 L⇒4

⇒ declaration
long int a; → slower. (or) long a;
short int a; → Faster (or) short a;

* In situations where the constant is small enough to be an int but we still want to give it as much storage as long. In such cases we add suffix 'L' or 'l' at the end of no. eg) 25L

Integers, signed and unsigned

- unsigned: when we know that the value stored will always be +ve
  unsigned int a; (or) unsigned a;
- Range shifts from (-2147483648/7) to [0, 4294967295]
- The leftmost bit is now free and is not used to store the sign of the number.
- occupies 2 bytes
- short unsigned int & long unsigned int exist.
- By default, short int → signed short int
  long int → signed long int



Chars, signed and unsigned

Both occupy 1 byte, but have different ranges.

```
char ch = 'A';
```

Binary equivalent of the ASCII / unicode value of 'A' is stored in ch. (i.e. binary 0065)

signed char ≈ normal char Range ⇒ [-128 to 127]

unsigned char Range [0, 255]

```
char ch = 291;
printf("%d %c", ch, ch); ⇒ 35 #
```

When value > +127, an appropriate value from other side of range is picked up & stored in ch.

```
int main()
{ char ch;
  for(ch=0; ch<=255; ch++)
    printf("%d %c\n", ch, ch);
  return(0);
}
```

[Infinite loop]
+127+1 ⇒ -128.

Floats and Doubles → 8 bytes
└ 4 bytes

⇒ Range of float [-3.4e38, +3.4e38]
⇒ Range of double [-1.7e308, +1.7e308]
⇒ long double ⇒ size = 10 bytes
⇒ [-1.7e4932 to +1.7e4932]

DATA TYPE	RANGE	SIZE	FORMAT
Signed char	-128 to +127	1	%c
unsigned char	0 to 255	1	%c
short sint	-32768 to +32767	2	%d
short unsigned int	0 to 65535	2	%u
signed int	-2147483648 to +2147483647	4	%d
unsigned int	0 to 4294967295	4	%u
long signed int	-+ 2147483647	4	%ld
long unsigned int	0 to 4294967295	4	%lu
float	-/+ 3.4e38	4	%f
double	-/+ 1.7e308	8	%lf
long double	-/+ 1.7e4932	10	%Lf

Some Issues

⇒ depending upon the microprocesor for which the compiler targets its code the accuracy of floating pt calculations may change.

⇒ Negative side is always stored as the 2's compliment of its binary.

eg) -128 ⇒ 10000000

1's compliment ⇒ 01111111

2's compliment ⇒ 10000000 → 8bit no. can be accomodated in a char.

+128 ⇒ 010000000 → 9bit no. (for +)

⇒ when we try to store +128 in a char only the right most 8 bits gets stored. But when 10000000 gets stored the left most 1 is treated as the sign bit thus +128 is stored as -128. -128.

⇒ If we exceed the range the from positive side we end up on the negative side & vice versa.

STORAGE CLASSES IN C

⇒ storage classes have defaults.

⇒ Kinds of memory locations: Memory & CPU registers.

⇒ Significance of storage classes:

a] Preferred storage

b] Default inital value without assignation.

c] scope of the variable

d] life of the variable.

⇒ Storage classes:

1] Automatic sc

2] Register sc

3] Static sc

4] External sc

Automatic storage classes

⇒ ST: memory
DIV: GARBAGE value
SC: Local to the block where variable is defined
L: End of block.

⇒ declaration auto int a; (or) int a;

⇒ int main()
{ auto int i=1;
{ auto int i=2;
{ auto int i=3;
printf("%d", i); O/P ⇒ 3
}
printf("%d", i); → O/P = 2
}
printf("%d", i); → O/P = 1
}

* all i = different

⇒ For variables with same name, the most local variable is given priority.

Register storage classes

Storage: CPU registers
DIV: Garbage
SC: local to block of definition
L: End of block.

⇒ value stored in CPU register can be accessed faster.
⇒ If a register can't be allotted then variable acts as auto.
⇒ declaration register int a;
⇒ If the microprocessor is 16bit it can't store float or double.

Static storage class

=> Storage: memory
DIV: 0
SC: Local to variable in which it is defined
L: End of program

```
void increment();
int main()
{ increment(); → initialized to 1
  -//- ---- -//-
  -//- ----
  return(0);
}
void increment()
{ auto int i=1;
  printf("%d\n", i);
  i=i+1; → 2
}
```
→ 2 is lost

o/p 1 1 1

```
void increment();
int main()
{ increment(); → initialized to 1 only once
  -//- ----
  -//- ----
  return(0);
}
void increment()
{ static int i=1;
  printf("%d\n", i);
  i=i+1;
}
```

o/p 1 2 3

=> Static variables are also local to the block in which they are defined but they don't ~~disappear~~ disappear when the func is no longer active.
=> If control comes back to the same func again, the static variables have the same values they had last time around.

```
=> int *fun();
int main()
{ int *j;
  j=fun();
  printf("%d", *j); →
}
int *fun()
{ int i=35;
  return(&i);
}
```

```
int *j;
j = fun();
printf("%d\n", *j);   output => 35
printf("%d\n", *j);   output ≠ 35
j = fun();
printf("%d\n", *j);   output = 35
return(0);
}
int *fun()
{ int k = 35;
  return(&k);
}
```

[Avoided by using k as static]

⇒ when the control returned from fun() the first time, though k went dead it was still left on the stack.
We then accessed this value using its address that was collected in j.
But when we precede the call to printf() by a call to any other func^n, the stack is now changed, hence we get the garbage value.

External Storage class

⇒ Storage : Memory
⇒ DIV : 0
⇒ Scope : global
⇒ Life : EOP
⇒ Declared outside all the functions.

eg)
```
int i;  → declaration
int main()
{ printf("%d", i);
  return(0);
}
```

```
int main()
{ extern int i;  → statement
  printf("%d", i);
  return(0);
}
int i;  → declaration
```

=) When we declare a variable, no space is reserved for it.
=) When we define a variable, space is reserved for it.
=) We declared extern int i; b/c its use is encountered in printf before its definition.
=) A variable can be declared several times but can only be defined once.
=) It's the local variable that gets the preference over the global variable.
=) A static variable can also be declared outside all func^n for all practical purposes, it will be treated as an extern variable.
There however, the scope of this variable is limited to the same file in which it is declared i.e. the variable would not be available to any func^n that is defined in a file other than the file in which the variable is defined.

Some Issues :-

=) All variables that are defined inside a func^n are normally created on the stack each time the func^n is called. These variable die as soon as control goes back from the function.
If variables are defined as static they are created in a place in memory called 'Data Segment' they die only when program execution comes to an end.
=) Variable defined outside all func^ns, it is available to the other func^ns in same & other files.
In other files the variables should be declared as extern
=) If we place static in front of an external variable it makes the variable private and not accessible to use in any other file.

⇒ auto int j;
static int i; } definitions
register int k;
extern int l; } declaration.

Which to use when

⇒ Used here to economise the memory space & improve the speed of execution.

⇒ static: To make value of variable persist btw different func calls.

⇒ register: for variables that are used very often.

⇒ extern: Variables that are used by almost all funcns in the program.

The C Preprocessor

→ Preprocessor: Program that processes our source program before it is passed to the compiler.

Build process: combination of steps involved from writing a C program to executing a C program.

C program ⇒ Source code

The preprocessor works on source code & creates an "Expanded Source Code"

ESC ⇒ stored in .I file.

features of a preprocessor ⇒ preprocessor directives.

directives can be placed anywhere

⇒ preprocessor directive

#preprocessor command <file_name> → special folder
<file_path>
"file_name". → any directory

#include "c:\abc\s.c"

Preprocessor directives

- Macro Expansion
- File Inclusion
- Conditional compilation
- Miscellaneous directives.

MACRO EXPANSION

#define UPPER 25 (macro definition) or (MACRO)

during preprocessing, every occurrence of UPPER in program is replaced by 25

#define PI 3.1415

area ⇒ PI * r * r ; → corresponding macro expansion

↳ macro template

⇒ It is customary to use capital letters for macro template. Eases programmers.

⇒ macrotemplate ___ macroexpansion
↓
BLANK or tabs

⇒ Macro definition is never terminated with a ;

⇒ By using macros, it becomes easier to make changes to a particular constant value in a program. (particularly in larger programs)

⇒ Variables vs Macros

A variable can be used to store a constant value but it is not used bic

a] It is inefficient, ∵ compiler can generate faster & more compact code for constant than for variable.

b] If something never changes, its hard to consider it as a variable

c] Danger that the variable may inadvertently get altered somewhere in the program.

Uses of macros

i) To define operators

#define AND &&

#define OR ||

ie ((F<5) AND (X<=20 OR Y<=25))

(ii) To duplicate a condition

#define ARANGE (a>25 && a<50)

ie (ARANGE)

(iii) To replace an entire statement

#define FOUND printf("The Yankee doodle virus\n");

ie (Character == 'Y')

FOUND

MACROS WITH ARGUMENTS

⇒ Macros can have arguments, just as functions can.

eg) #define AREA(x) (3.14*x*x)

a = AREA(y1);

↳ expansion ⇒ a = 3.14 * y1 * y1. [y1 gets substituted for x]

eg) #define ISDIGIT(y) (y>=48 && y<=57)

if (ISDIGIT(ch))

Writing Macros with Arguments

⇒ Do not leave a blank between the macro template & its argument

eg) Area (x) (3.14*x*x)

template expansion

⇒ Enclose entire macro expansion within parenthesis.

eg) #define SQUARE(n) n*n

I = 64/SQUARE(4) ⇒ expansion 64/4*4 ⇒ 16*4 = 64

⇒ Macros can be split into multiple lines.

```
#define HLINE for(i=0; i<79; i++) \
              printf("%c", 196);

#define VLINE {\
               goto(x,y); \
               printf("%c", 179); \
              }
```

⇒ If you are unable to debug a macro, then you should view the expanded code of the program (present in .I file) can be generated using command prompt by saying

CPP pro1.c

↳ C preprocessor

gets generated in C:\TC\BIN. for TC & TC++

Macros vs functions

⇒ In a macro call, the pp replaces the macro template with its macro expansion, in a stupid, unthinking, literal way.
In func^n call, control is passed to a func^n along with certain arguments, some calculations are performed in the func^n & a useful value is returned back.

⇒ Macros makes the program run faster but increase the program size, whereas func^n make the program smaller & compact.

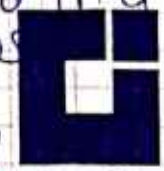
⇒ If we use macro honored times in a program, the macro expansion goes into our source code at hundred different places the ↑ the program size.
If a func^n is used at hundred of place, it would take the same amt of space in the program. But passing & collecting arguments & values takes time & ∴ slows down the program.

⇒ If macro → sweet & simple → use it
If macro → fairly large & is used fairly often → use func^n

FILE INCLUSION

⇒ This directive causes one file to be included in another.
⇒ #include "filename"
⇒ It simply causes the entire contents of 'filename' to be inserted into the source code at that point in the program.
uses :-
① very large program's produc^n into smaller files
② Commonly needed func^ns & macro defin^ns can be stored in a file & that file can be included in every program that we write.

⇒ When creating our own library of functions which we will to distribute to others. The funcns are defined in a ".c" file and their corresponding prototypes declarations & macros are declared in a .h file. This way the funcns definitions in the .c file remain with you and are not exposed to the users of these funcns.



⇒ Included files commonly have a .h extension which stands for 'header file'. The prototypes of all the library funcns are grouped into different categories and then stored in different header files.

Ways to write #include statement

⇒ #include "filename"

⇒ #include <filename>

⇒ "" ⇒ looks for the file in current directory as well as in the specified list of directories as mentioned in the include search path that might have been set up

⇒ < > ⇒ looks for the file in the specified list of directories only.

⇒ The include search path is nothing but a list of directories that would be searched for the file being included.

⇒ We can also specify multiple include paths separated by ;

eg) c:\tc\lib; c:\mylib; d:\libfiles.

⇒ Path can contain a maximum of 127 characters.

⇒ Both relative & absolute paths are valid.

eg) '..\dir\incfiles'

CONDITIONAL COMPILATION

⇒ we can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands #ifdef & #endif.

⇒ form

```
#ifdef macroname
    statement 1;
    ---"---
    ---"---
#endif.
```

⇒ If macroname has been #defined → block of code is processed otherwise not.

USES OF #ifdef

a] To comment out obsolete lines of code. when program is changed at last minute. To remove the old code we may comment it out but it may result in nesting of comments which is not allowed. The soln is to use conditional compilation.

eg)
```
int main()
{#ifdef OKAY
      { } compiled only if OKAY is defined.
 #endif
 }
```

b] To make programs portable by isolating the lines of code that must be different for each machine by marking them off with #ifdef.

```
int main()
{ #ifdef INTEL
      code suitable for an intel PC
```

#else code suitable for a Motorola PC
#endif
code common to both computers
}

2) working of #ifdef-#else-#endif is similar to the ordinary if-else control instructions of C.
2) sometimes instead of the #ifdef, the #ifndef directive is used. It means if not defined. Works exactly opposite to #ifdef.

3) Let myfun() fun^n be defined in 'myfile.h' which is #included in a file myfile1.h
If we include both myfile.h & myfile1.h, the compiler flashes an error 'Multiple declaration for myfun()'
To avoid this we can write following code in the myfile.h header file.

```
/* myfile.h */
#ifndef macroname
#define macroname
   myfun()
   {/*some code*/}
#endif.
```

1st time → true ↓
2nd time → false X.

#if and #elif DIRECTIVES

2) The #if used to test whether an expression evaluates to a nonzero value or not.
If result nonzero then subsequent lines upto #else/#elif /#endif are compiled, otherwise skipped.

eg)
```
int main()
{ #if TEST <= 5
  {
```

```
#else
   }
#endif
  }
```

⇒ other expressions like (LEVEL == HIGH || LEVEL == LOW) (or) (ADAPTER == VGA) can be used.

⇒ we can have nested conditional compilation directives.

```
#if ADAPTER==VGA
#else
  #if ADAPTER==SVGA
  #else
  #endif
#endif
```

```
#if ADAPTER==VGA
#elif ADAPTER==SVGA
#else
#endif
```

MISCELLANEOUS DIRECTIVES

a] #undef

b] #pragma

#undef

⇒ use to undefined name to become undefined.

#undef macro template

#pragma directive

⇒ use to turn on or off certain features

a] #pragma startup & #pragma exit

⇒ Allow us to specify functions that are called upon program startup (before main()) or program exit (just before the program terminates).

```
#include <stdio.h>
void fun1();
void fun2();
#pragma startup fun1
#pragma exit fun2
int main()
{ printf("inside main\n");
  return(0);
}
void fun1()
{ printf("inside fun1\n"); }
void fun2()
{ printf("inside fun2\n"); }
```

output

```
inside fun1
inside main
inside fun2
```

सी-डॉट C-DOT

⇒ functions fun1 & fun2 should neither receive nor return any value.

b) #pragma warn:
- tells the compiler whether or not we want to supress a specific warning.

eg)
```
#include <stdio.h>
#pragma warn-rvl  /*return value*/
#pragma warn-par  /*parameter not used*/
#pragma warn-rch  /*unreachable code*/
```

If we replace the - sign with a + sign then these warnings would be flashed on compilation.

CONCATENATE (##)

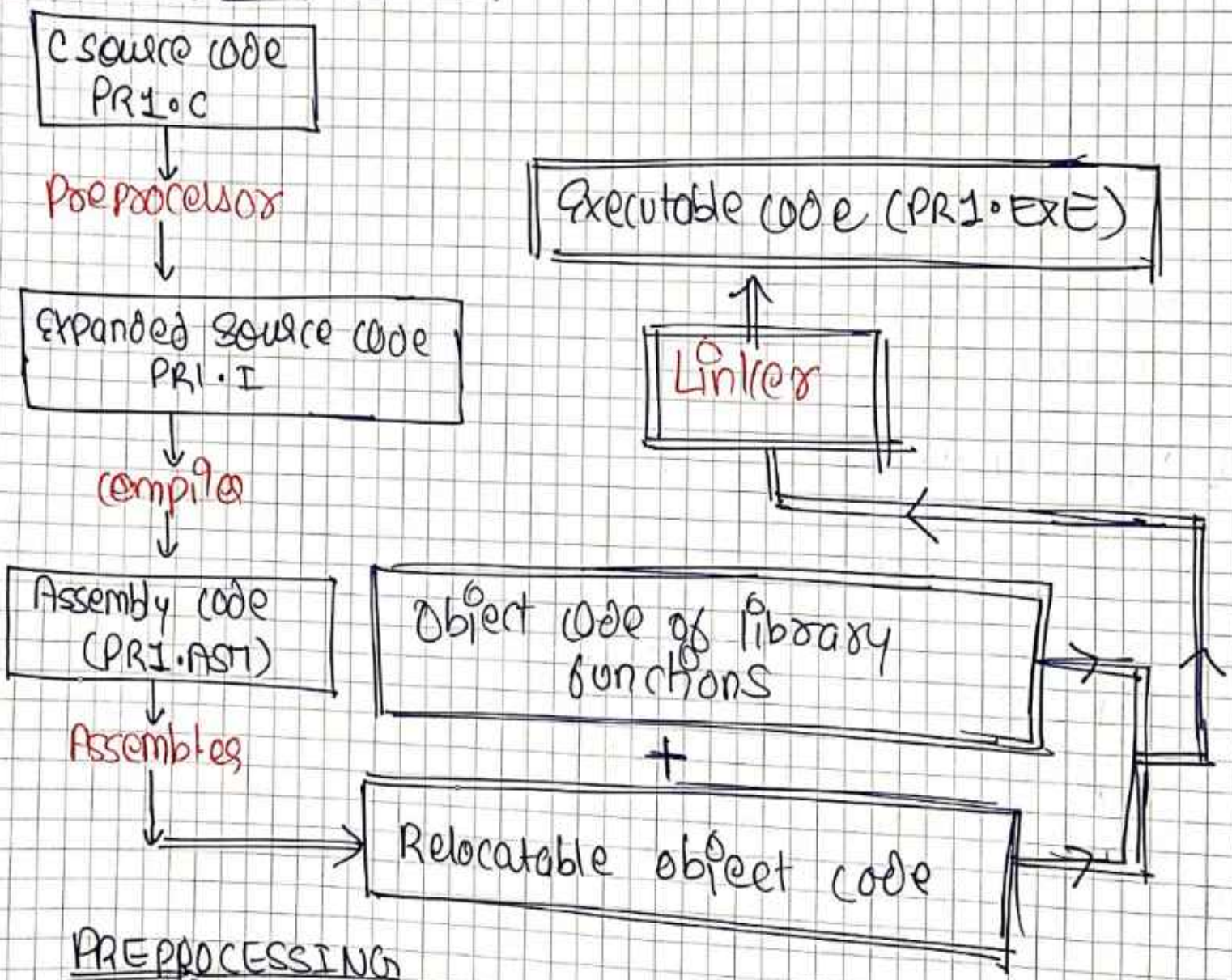
```
#define action(a,b) a##b + a*b
main()
{ printf("%d", action(3,4));
}
```

3 4 + 3*4

DEFINING MACRO LIKE A FUNCTION

```
#define ABS(a) (a)<0 ? -(a) : (a)
{ printf("Abs(6-16) : %d %d", ABS(-1), ABS(1));
}
```

THE BUILD PROCESS



PREPROCESSING

⇒ The C source code is expanded based on pp directives.
⇒ The expanded src code is also in C language.
⇒ extension .I may vary from compiler to compiler.

COMPILATION

⇒ Compiler identifies the syntax errors in the expanded code.
⇒ An error free expanded code is translated into an equivalent assembly language problem.
⇒ Compilers targeted towards different processors may generate different assembly language code.
⇒ Assembly code is typically stored in .ASM file.

ASSEMBLING

⇒ Assembler translated .ASM file into Relocatable object code.
⇒ Relocatable object code is stored in .OBJ file.
⇒ Relocatable : no specific memory addresses have ~~been~~ yet been assigned to the code and data sections in the relocatable code.
⇒ All addresses are relative offsets.
⇒ The .OBJ file gets created in a specially formatted binary file.
⇒ The object file contains header & several sections :

a] Text section : contains machine language code equivalent to the expanded source code.
b] Data section : Contains global variables & their initial values.
c] BSS (Block started by Symbol) section : Contains ~~information about symbols~~ found uninitialized global variables.
d] Symbol table : contains information about symbols found during assembling of the program. Information include :
⇒ Names, types & sizes of global variables.
⇒ Names & addresses of functions defined in the source code
⇒ Names of external functions like printf() & scanf().

⇒ .OBJ can't be executed directly inspite of containing machine language instructions b/c:

⇒ ~~extern~~ external fonc^ns like printf() are not present in the .OBJ file.

⇒ .OBJ file may use global variables defined in another .OBJ file

⇒ .OBJ file may used fonc^n defined in another .OBJ file

⇒ Parts of the symbol table may be incomplete b/c all the variables & fonc^ns may not be defined in the same file. The references to such variables & functions (symbols) that are defined in other source files are later on resolved by linker.

LINKING

(i) find definition of all external fonc^ns - those which are defined in other .OBJ files, & those which are defined in libraries (eg printf())

(ii) find definitions of all global variables - those which are defined in other .OBJ files and those which are defined in libraries

(iii) Combine data sections of diff. .OBJ files into a single data section.

(iv) Combine code sections of diff .OBJ files into a single code section

⇒ Addresses of all variables & fonc^ns in the symbol table of the .OBJ file are Relative addresses. This means that address of a symbol (variable or fonc^n), is in fact only an offset from start of the section (data or the code section) to which it belongs.

⇒ Same addressing scheme is used with forms present in each .OBJ file.

2) When linker combines 2 .OBJ files, it has to re-adjust the addresses of global variables and functions similarly, readjustment of addresses will be done for functions.

3) Even after readjustment, the addresses of variables & fcns are still relative in the combined data & code sections of the .EXE file.

4) During linking if the linker detects errors such as misspelling the name of library fcn in src code or using the incorrect no. or type of parameter for a fcn, it stops the linking process & doesn't create the binary executable file.

सी-डॉट
C-DOT

LOADING

⇒ When we execute an .EXE file, it is first brought from the disk into the memory (RAM) by an OS component called Program loader.

2) Like .OBJ, .EXE is also a formatted binary file. format of this .EXE is OS dependent.

3) Windows uses PE ⇒ Portable executable
Linux uses ELF ⇒ Executable & linking format.

4) Hence .OBJ & .EXE created for one OS, can't be used for another OS.

ARRAYS

⇒ There are certain logics that cannot be dealt with, without the use of an array.

⇒ An array is a collective name given to a group of similar quantities

⇒ Each member in an array is referred to by its position in the group.

⇒ The notation to refer to i+1th element is arrayname[i]
here arrayname ⇒ Subscripted variable
i ⇒ Subscript.

⇒ The similar elements of an array could be all ints/ all floats/ all chars etc.

⇒ The array of characters ⇒ string whereas an array of ints or floats is called simply an array.

ARRAY DECLARATION

⇒ type arrayname[n]; → tells the compiler that we are dealing with an array.
type of variable → dimensions [no of elements of a particular type]

Accessing elements of an array

⇒ nth (n+1)th element ⇒ array name[i]

Entering data into an array

array size

```
⇒ for (i=0; i<=n; i++)
  { printf ("Enter value");
    scanf ("%d", &marks[i]);
  }
```

↳ passing address of certain array element.

Reading data from an Array

```
for(i=0; i<n; i++)
{ sum=sum+arrayname[i];
}
```


Note:-

⇒ An array is also known as subscripted variable.

⇒ However big an array is, its elements are always stored in contiguous memory locations.

ARRAY Initialisation

```
int arrayname[n]={2,4,12,5,45,5};
int arrayname[]={2,4,12,5,45,5};
```

⇒ If array elements are not given specific values, they are supposed to contain garbage values.

⇒ If the array is initialised where it is declared, mentioning the dimension of the array is optional.

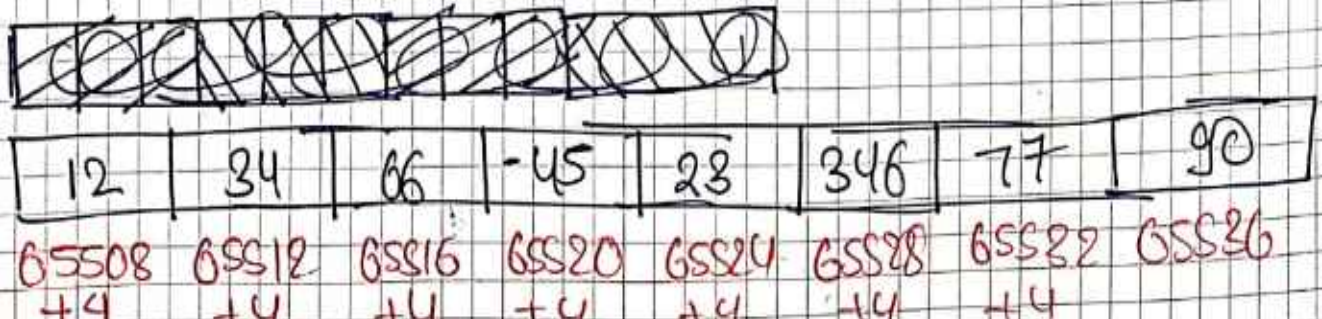
ARRAY ELEMENTS IN MEMORY

eg) `int arr[8];`

⇒ 32 bytes memory gets immediately reserved in memory.

& since array is uninitialised all 8 values are garbage values

⇒ b/c storage class of this array is assumed to be auto.



Pointers & Arrays

=> Every time a pointer is incremented, it points to the immediately next location of its type. & vice versa.

operations on pointers

a] number + pointer.

eg] int i=4, *j, *k;
j => &i;
j = j+1;
j = j+9;
k => j+3;

b] pointer - number

eg) int i=4, *j, *k;
j=&i;
j=j-2;
j=j-5;
k=j-6;

c] pointer - pointer

=> One pointer can be subtracted from another variable provided both variable point to elements of the same array.

=> The resulting value indicates the number of elements separating the corresponding array elements.

d] comparison of pointer variables

=> pointer variables can be compared provided both variables point to the objects of the same data type. Such comparisons are useful when both pointer variables point to elements of the same array.

=> The comparison can test for either equality or inequality

=> A pointer variable can be compared to 0 (expressed as NULL usually)

```
#include <stdio.h>
int main()
{ int arr[] = {10, 20, 30, 72, 45, 36};
  int *j, *k;
  j = &arr[4];
  k = (arr+4);
  if (j==k)
  else
  return 0;
}
```

INVALID Operations on pointers

a] pointer + pointer
b] number x pointer
c] pointer/number.

Accessing an array using pointers

```
int num[] = {24, 14, 12, 44, 56, 72};
int i, *j;
j = &num[0];
for (i=0; i<=5; i++)
{
    printf("address = %u", j);
    printf("element = %d\n", *j);
    j++;
}
return 0;
}
```

⇒ Accessing array elements by pointers is always faster than by subscripts.

⇒ If the elements are to be accessed in a fixed order or any definite logic → should be accessed using pointers.

⇒ If there is no fixed logic in accessing the elements, it would be easier to access them using subscripts.

Passing an entire array to a function

```
{ int num[] = {24,24,12,44,56,17};
  display(&num[0], 6);
  return 0;
}
void display(int *j, int n)
{ int i;
  for(i=0; i<=n-1; i++)
  { printf("element = %d", *j);
    j++;
  }
}
```

⇒ Just passing the address of zeroth element of the array to a func^n is as good as passing the entire array to the func^n.

⇒ It is also necessary to pass the total number of elements in array to the func^n.

⇒ address of zeroth element ⇒ Base address.

⇒ Base address can be passed by just passing the name of array.

display(&num[0], 6) = display(num, 6)

⇒ *num = *(num+0) ⇒ 24

⇒ *(num+i) ⇒ i^{th} element of array ⇒ num[i]

⇒ num[i] = *(num+i) = *(i+num) = i[num]

⇒ num+i = i+num [passing address]

[WAYS OF PASSING VALUES]

TWO DIMENSIONAL ARRAYS

→ The 2D array is also called a matrix.

⇒ eg) int stu[4][2]; → 2nd subscript ⇒ column number
→ first subscript ⇒ row no.

conceptual visualisation

	COL 0	COL 1
ROW 0	1234	56
ROW 1	1212	33
ROW 2	1434	80
ROW 3	1312	78

⇒) 2D array is collection of a number of 1D arrays placed one below the other.

Initialising a 2D array

```
int stu[4][2]={
                {1234,56},
                {1212,33},
                {1434,80},
                {1312,78}
              };
```

(OR)

```
int stu[4][2]={1234,56,1212,33,1434,80,1312,78};
```

⇒ while initialising a 2D array it is necessary to mention the column dimension, whereas the first row dimension is optional.

eg) int arr[2][2]={ ~ }; ✓
int arr[][2]={ ~ }; ✓
int arr[2][]={ ~ }; ✗
int arr[][]={ ~ }; ✗

BOUNDS CHECKING

⇒ There is no check to see if the subscript used for an array exceeds the size of the array.

⇒ Data entered with a subscript exceeding the array size will simply be placed in memory outside the array, probably on top of other data, or on the program itself.

⇒ As our botheration is not compiler's to see to it, if we do not reach beyond the array size

---

PASSING ARRAY ELEMENTS TO A FUNCTION.

⇒ Passing can be done by calling the fun^n by value or by reference.

Call by value

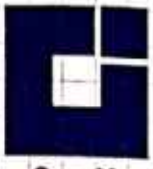
```
#include <stdio.h>
void display(int);
int main()
{ int i;
  int marks[] = {55,65,75,56,78,78,90};
  for(i=0; i<=6; i++)
     display(marks[i]);
  return 0;
}
void display(int m)
{ printf("%d", m); }
```

call by reference

```
#include <stdio.h>
void display(int*);
int main()
{   int j;
  int marks[] = { };
  for(i=0 ; i<=6; i++)
    display(&marks[i]);
  return 0;
}
void display(int *n)
{ printf("%d", *n);
}
```

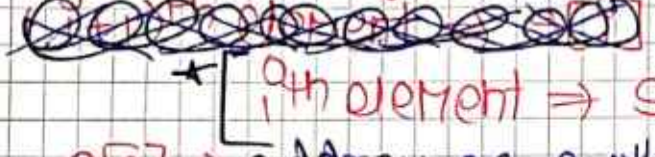
Memory Map of a 2D array

⇒ Memory doesn't contains Rows and COLUMNS.
⇒ Whether, it is a 1D or a 2D array, the array elements are stored in one continuous chain.



Pointers of 2D arrays

⇒ C language can treat parts of arrays as arrays.
⇒ Each row of a 2D array can be thought of as a 1D array.
eg) int s[5][2]; 5x2 → columns in each R.
⇒ If we can imagine s as a 1D array then it is

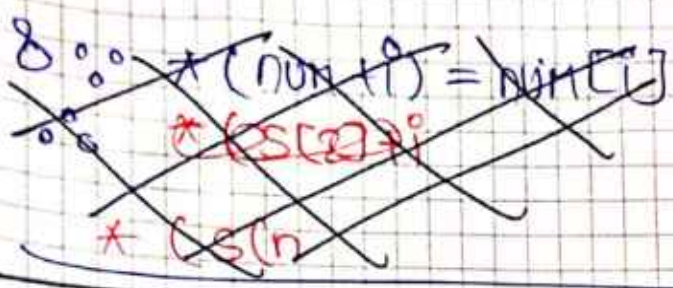


s[0] ⇒ address of zeroth 1D array.
s[i] ⇒ address of ith 1D array

eg] int s[4][2]={
{},
{},
{},
{}
};

```
int i;
for(i=0; i<=3; i++)
printf("add of %dth 1D array is %u\n", i, s[i]);
```

⇒ Here each 1D array starts 4 bytes farther along than the last one.
⇒ Accessing individual elements.
s[i] ⇒ address of ith 1D array.
Address of (i+1)th element ⇒ s[n] + i
value of (i+1)th element ⇒ *(s[n]+i)



∴ *(num+i) = num[i]

∴ *(s[n]+i) = *(*(s+n)+i)

Summing up

Address of (i+1)th element in nth Row ⇒ s[n] + i

values of this (i+1)th element ⇒ s[n][i]

⇒ *(s[n]+i)

⇒ *(*(s+n)+i)

Pointer to an Array

```
#include <stdio.h>
int main()
{ int s[4][2] = {
                  {1234,56},
                  {1212,33},
                  {1434,80},
                  {1312,78}
                };
  int (*p)[2];      (p is a pointer to an array of 2 integers)
  int i, j, *pint;
  for (i=0; i<=3; i++)
  { p = &s[i];
    pint = (int*) p;
    printf("\n");
    for (j=0; j<=1; j++)
    printf("%d", *(pint+j));
  }
  return 0;
}
```

Pointer to an array is immensely useful when we need to pass a 2D array to a func^{n}.

Passing 2D array to a function

There are 3 ways of passing a 2D to a func^{n}

```
#include<stdio.h>
void display(int *q, int, int);
void show(int(*q)[4], int, int);
void print(int q[][4], int, int);
int main()
{ int a[3][4]={ 1,2,3,4,
                5,6,7,8,
                9,0,1,6
              };
  display(a,3,4);
  show(a,3,4);
  print(a,3,4);
  return 0;
}
void display(int *q, int row, int col)
{ int i,j;
  for(i=0; i<row; i++)
  { for(j=0; j<col; j++)
      printf("%d", *(q+i*col+j));
    printf("\n");
  }
  printf("\n");
}
void show(int (*q)[4], int row, int col)
{ int i,j;
  int *p;
```

i*col changes rows

```
for (i=0; i<row; i++)
  {
    p = q+i;
    for (j=0; j<col; j++)
      printf("%d", *(p+j));   → same as q[i][j]
    printf("\n");
  }
printf("\n");
}
void print (int q[][4], int row, int col)
{ int i,j;
  for (i=0; i<row; j++)
  {
    for (j=0; j<col; j++)
      printf("%d", q[i][j]);
    printf("\n");
  }
printf("\n")
}
```

⇒ General formula
* (base address + row no. * no. of columns + column no.)

⇒ int (*q)[4]; → pointer to an array of 4 integers.
q holds the base address of 0th 1D array

⇒ int q[][4]; is same as int (*q)[4];
where q is a pointer to an array of 4 integers
Only advantage is that, we can now use a more familiar expression q[i][j].

ARRAY OF POINTERS

=> The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses.

eg) `int *arr[4];` array of int pointers.

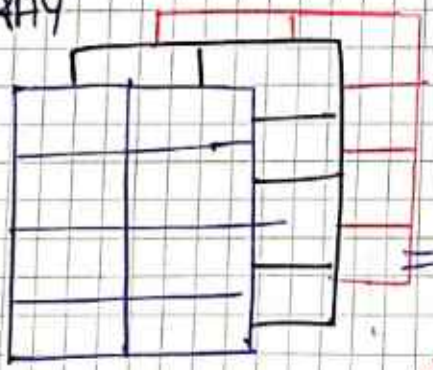
=> An array of pointers can even contain the addresses of other arrays.

eg)
```
{ static int a[] = {0,1,2,3,4};
  int *p[] = {a, a+1, a+2, a+3, a+4};
  printf("%u %u %u %d\n", p, *p, *(*p));
  return 0;
}
```

3D ARRAY → 3 × [4 × 2]

=> `int arr[3][4][2]` = 3 arrays of size 4×2 (2D)

1ST 2D ARRAY



0th 2D ARRAY

2ND 2D ARRAY

=> `arr[2][3][1];`

OR

`*(*(*(arr+2)+3)+1)`

---

for 1D arrays

=> if initialisations are more than variable → error

=> if initialisations are less than variable → assigned 0 in remaining.

STRINGS

⇒ Character arrays are many a time also called strings.
⇒ Used to manipulate text, such as words or sentences.
⇒ A string constant is a 1D array of characters terminated by a null ('\0')
eg) `char name[] = {'H','A','L','S','E','Y','\0'};`
⇒ Each character occupies one byte memory. & the last character is always \0.
⇒ \0
⇒ It is one character not two.
⇒ \ indicates that what follows it is special.
⇒ called as null character.
⇒ \0 ≠ 0
⇒ ASCII values of \0 = 0 & 0 = 48
⇒ The terminating \0 is imp bcz it is the only way the funcns that work with a string can know where the string ends.
⇒ A string not terminated by \0 is not really a string but merely a collection of characters.
⇒ Another way of ~~declaring~~ initializing strings

`char name[] = "HALSEY";`

In this declaration \0 is not necessary. C inserts the null character automatically.

More abt strings

```
char name[] = "Klinsman";
int i=0;
while(i<=7)
{ printf("%c", name[i]);
  i++;
}
printf("\n");
```

```
char name[] = "Klinsman";
int i=0;
while(name[i] != '\0')
{ printf("%c", name[i]);
  i++;
}
printf("\n");
```

```
char name[]="klinsman";
char *ptr;
ptr=name (stores the base adress of string)
while(*ptr!=0)
{ printf("%c", *ptr);
  ptr++;
}
printf("\n");
return 0;
```

⇒ The values of elements of char array or string can be accessed by
name[i], *(name+i), *(i+name), i[name]

⇒ printf doesn't prints \0.

⇒ input & output in/of a string

⇒ char name[]="klinsman";
printf("%s", name); → base address
%s ⇒ format specifier for printing out a string.

⇒ char name[25];
scanf("%s", name);

⇒ scanf fills in the characters typed at the keyboard into the array until the enter key is hit. One hitting enter key it places a \0 in the array.

Note :-
input using scanf()

⇒ length of string should not exceed the dimension of the character array. as compiler doesn't performs bounds checking on char arrays. This may overwrite something imp.

⇒ Scanf cannot receive multiword strings. because it considers space as data separator.

Gets() & Puts()

⇒ char name[25];
gets(name); → base address
puts("Hello!!");
puts(name); → base address

⇒ puts can display only one string at a time. On displaying a string puts places the cursor on the next line.

⇒ Scanf can accept multiword strings as
scanf("%[^\n]s", name);
↑ indicates that scanf() will keep receiving char into name[] until a \n is encountered

Pointers & Strings

⇒ char str[] = "Hello";
⇒ char *p = "Hello"; → address is assigned in p.
∴ we cannot assign a string to another.
⇒ we can assign a char pointer to another char pointer
eg) char str1[] = "Hello";
char str2[10];
char *s = "Good Morning";
char *q;
str2 = str1; (error)
q = s; (works)

⇒ Once a string is defined, it cannot be initialized to another set of characters. Unlike strings, such operation is valid with char pointers
eg) char str1[] = "Hello";
char *p = "Hello";
str1 = "Bye"; (error)
p → "Bye"; (works)

Standard library string functions

strlen → length
strlwr → lowercase conversn
strupr → uppercase conversn
strcat → Appending str at the end of other
strncat → Appends first n chars
strcpy → copies one string into another
strncpy → copies first n chars
strcmp → compares 2 strings
strncmp → compares first n chars.
stricmp → compares 2 strings ignoring case
strnicmp → compares first n chars ignoring case
strdup → duplicates a string
strchr → finds first occurrence of a given char
strrchr → finds last occurrence of a given char
strstr → finds first occurrence of a given str in another str
strset → sets all chars of a string to a given char.
strnset → sets first n chars to a given char
strrev → Reverses string.

Strlen()

⇒ char arr[] = "BoHboozled";
int len1, len2; → base address.
len1 = strlen(arr);
len2 = strlen("Humpty Dumpty");
⇒ while calculating length strlen() doesn't count \0.
⇒ int xstrlen(char *s)
{ int length = 0;
while(*s != '\0')
s++, length++;

```
    s++;
  }
  return(length);
}
```

---

strcpy()

⇒ The base address of the source & target strings should be supplied.

↳ `char source[]="Sayonara";`
`char target[20];`
`strcpy(target,source);`

↳ It copies the characters in the source string into the target string till it encounters the end ('\0').

⇒ `void xstrcpy(char*t, char *s)`
`{ while(*s!='\0')`
`{ *t= *s;`
`s++;`
`t++;`
`}`
`*t='\0';`
`}`

⇒ It is necessary to place a '\0' into the target string, to mark its end.

⇒ `strcpy(char*t, const char *s);` (standard library definition)
Ensures that the source string is not changed accidently.

Notes:- Marks as const pointer to string

`char str[]="Quest";`
`char *p="Quest";`
↳ pointer too const string

`str++;` ✗ `p++;` ✓
`*str='z';` ✓ `*p='M';` ✗

Strcat()

⇒ It concatenates the source string at end of the target string.

⇒
```
char source[] = "folks!";
char target[30] = "Hello";
strcat(target, source);
```


Strcmp()

⇒ The 2 strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first.

⇒ If 2 strings are identical ⇒ returned value ⇒ 0

⇒ If 2 strings are non identical ⇒ returned value ⇒ numerical difference btw the ASCII values of the 1st non matching pair of characters

⇒ `strcmp(string1, string2);`
→ Base addresses

⇒ We usually want to know whether or not 1st str is alphabetically before the second string. If it is, a negative value is returned; if it isn't, a +ve value is returned.

Two dimensional arrays of characters

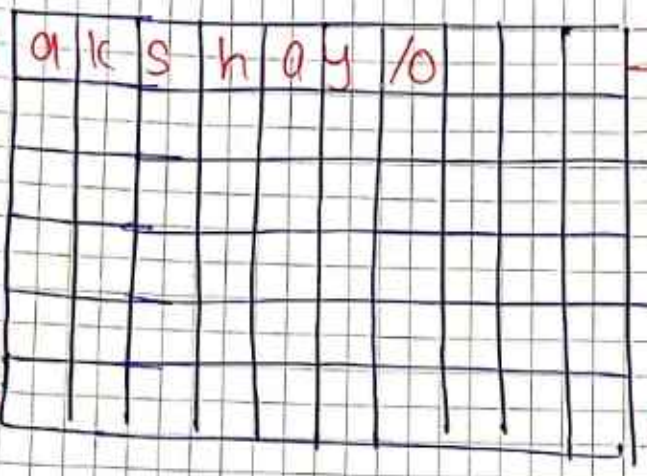
⇒ Initialization

```
char namelist[6][10] = {"Arshay",
                        "Pavan",
                        "Karan",
                        "Srinivas",
                        "gopal",
                        "Rajesh"};
```

⇒ The 1st subscript ⇒ no. of names in array.
2nd subscript ⇒ length of each item.

Taking input

```
for (i=0; i<=5; i++)
    scanf("%s", &masterlist[i][0]);
```



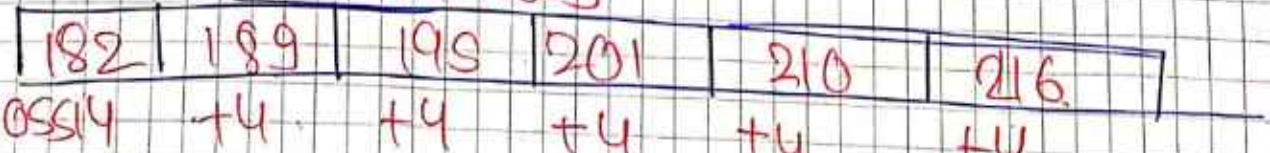
Array of pointers to string

⇒ It would contain a number of addresses.

⇒
```
char *names[] = { "akshay",
                  "rana",
                  "raman",
                  "srinivas",
                  "gopal",
                  "rajesh"
                };
```
→ base addresses of these names gets stored.

⇒ Let base addresses be 182, 189, 195, 201, 210, 216, 189

names[]



⇒ AOP makes more efficient use of memory.
∑ Memory used ⇒ 41 bytes

Manipulation using AOP

=> Exchanging 2 names

```
char names[][10] = {
                    "_",
                    "_",
                    "_",
                    "_",
                    "_",
                    "_"
                   };
int i;
char t;
for(i=0; i<=9; i++)
{ t = names[2][i];
  names[2][i] = names[3][i];
  names[3][i] = t;
}
```

total 10 exchanges

```
char *names[] = {
                 "_",
                 "_",
                 "_",
                 "_",
                 "_",
                 "_"
                };
char *temp;
temp = names[2];   address of 2nd string
names[2] = names[3]
names[3] = temp;
```

one exchange.

Limitations of AOP to strings

1. When we are using AOP to s we can initialize the strings at the place where we are declaring the array, but we cannot receive the string from keyboard using scanf().

eg)
```
char *names[6];
int j;
for(i=0; i<=5; i++)
{ scanf("%s", names[i]);
}
return 0;
}
```

doesn't work

2. On declaration, Array is containing garbage values. & it would be wrong to send these garbage values to scanf() as addresses where it should keep the strings received from keyboard.

Solution to this limitation

⇒
```
char *names[6];
char n[50];
int len, i;
char *p;
for (i=0; i<=5; i++)
{   scanf("%s", n);
    len = strlen(n);
    p=(char*) malloc(len+1);
    strcpy(p,n);
    names[i]=p;
}
```

---

malloc() → stdlib func^n

returns base address of type void* → means a pointer which is a legal address but it is not an address of any data type.

NOTE:-

⇒ With arrays, we have to commit to the size of the array at the time of writing the program.

⇒ There is no way to ↑ or ↓ the size during execution.

⇒ When we use arrays, static memory allocation takes place

STRUCTURES

=) A structure contains a number of data types grouped together.

eg)
```
struct book          ← new data type
{ char name;
  float price;
  int pages;
};                   } declaration
struct book b1, b2, b3;
```

Accessing a structure

```
scanf("%c%f%d", &b1.name, &b1.price, &b1.pages);
printf("%c%f%d", b1.name, b1.price, b1.pages);
```

General form of structure declaration

```
struct (structure name)
{ structure element 1;
  // 2;
  // 3;
  .
  .
};
```

=) The bytes allocated to structure elements are always in adjacent memory locations.

Declaration & variables creation

(i)
```
struct book
{ char name;
  float price;
  int pages;
};
struct book b1, b2, b3;
```
(data type)

(ii)
```
struct book
{ char name;
  float price;
  int pages;
} b1, b2, b3;
```

(iii) Struct
```
{ char name;
  float price;
  int pages;
} b1, b2, b3;
```

Initialization + declaration of struct variables

```
Struct book
{ ... };
Struct book b1={"Basic", 120.00, 500};
Struct book b2={"physics", 150.80, 800};
Struct book b3={0}; → all the elements are set to value 0
```

⇒ Structure type declaration does not reserves any memory space in memory. It only defines form of the structure.

⇒ Usually, structure type declaration appears at the top of the source code file.
In very large programs they are usually put in a separate header file & the file is #included in whichever program we want to use the str. type.

Accessing Structure elements

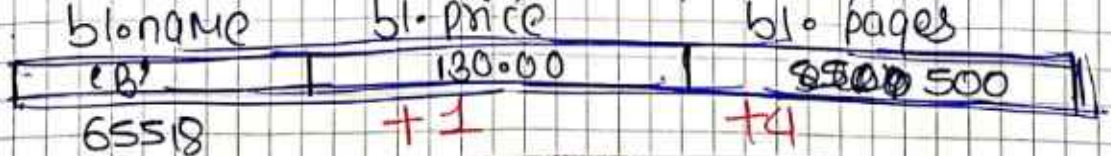
dot operator (.) ⇒ before dot operator there must always be a structure variable & after dot there must always be a structure element.

eg) b1.pages b1.price

Storage scheme of structure elements.

⇒ Whatever be the elements of a structure, they are always stored in contiguous memory locations.

eg) Struct book b1={'B', 130.0, 500};



ARRAY OF STRUCTURES

सी-डॉट
C-DOT

```
struct book
{ char name;
  float price;
  int pages;
};
struct book b[100];
scanf("%c%f%d", &b[i].name, &b[i].price, &b[i].pages);
printf("%c %f %d", b[i].name, b[i].price, b[i].pages);
```

void linkfloat() func^n

declaration: void linkfloat();

definition:
```
void linkfloat()
{ float a=0, *b;
  b=&a; (cause emulator to be linked)
  a=1*b; (supress the waning- variable not used)
}
```

=> defined to avoid the error "floating point formats not linked"

when parsing our source file, when the compiler encounters a reference to the address of a float, it sets a flag to have the linker link in the floating point emulator used to manipulate floating pt numbers in func^n like scanf() & atof().

=> Cases in which the use of float is a bit obsure & the compiler does not detect the need for an emulator. The most common is using scanf() to read a float in a array of structures.

=> linkfloat() forces linking of the floating point emulator into an application.

=> Just define the func^n anywhere in the program.

i) Additional features of structures

The values of a structure variable can be assigned to another structure variable of the same type

```
Struct employee e1={"sanjay", 20, 5500.50};
Struct employee e2,e3;
strcpy (e2.name, e1.name);  → not e2.name = e1.name
e2.age = e1.age;
e2.salary = e1.salary;
e3=e2;  (copying all elements at one go)
```

Here need copying

→ C does not allow assigning the contents of one array to another just by equation the two. for copying arrays we need to copy the contents of the array elements by elements

→ This copying of all elements elements at one go has been possible only because the structure elements are stored in contiguous memory locations.

ii) One structure can be nested within another structure

a)
```
Struct address
{ char phone[15];
  char city[25];
  int pin;
};
Struct emp
{ char name[25];
  Struct address a;
};
Struct emp e={"jeru", "582046", "nagpur", 10};
```

```
printf("name=%s phone=%es\n", e.name, e.a.phone);
printf("city=%s pin=%d\n", e.a.city, e.a.pin);
return 0;
}
```

iii) A structure variable can also be passed to a func^n. We may pass individual structure elements or the entire structure variable at one shot.

eg)
```
void display (char*, char*, int);
int main()
{ struct book
  { char name[25];
    char author[25];
    int callno;
  };
  struct book b1 = {"letusc", "ypk", 101};
  display(b1.name, b1.author, b1.callno);
  return 0;
}
void display (char *s, char *t, int n)
{ printf("%s%s%d\n", s, t, n);
}
```

→ It becomes necessary to declare the structure type struct book outside main(), so that it becomes globally available so, that we can define formal arguments as struct book b1 in the func^n being called.

v) We can have a pointer pointing to a struct. Such pointers are called as structure pointers.

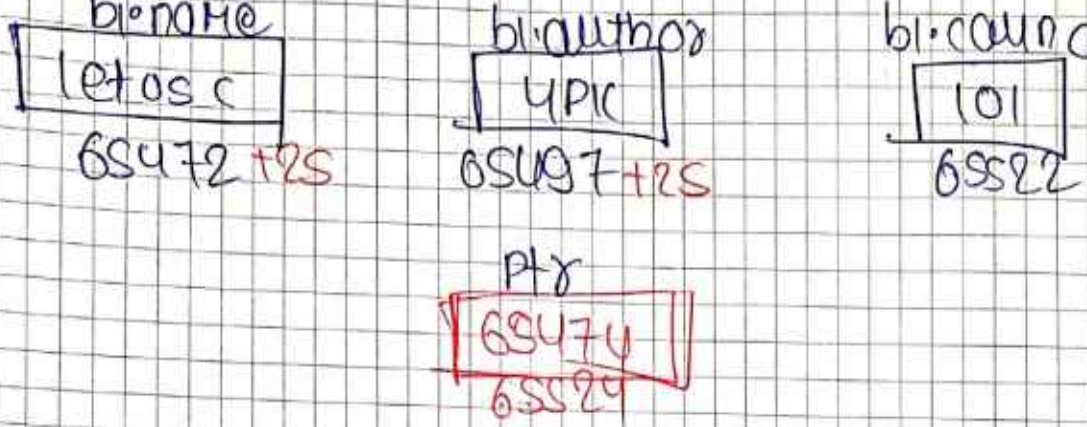
```
struct book b1 = {"letusc", "ypk", 101};
struct book *ptr;
```

```
ptr=&bl;
printf("%s %s %d\n", ptr->name, ptr->author, ptr->count);
return 0;
}
```

⇒ Arrow operator (->)

⇒ On LHS of the arrow operator there MUST always be a pointer to structure whereas on RHS there most always be a structure variable.

⇒



⇒ We can also pass the address of a structure variable to a function.

```
struct book
{ char name[25];
  char author[25];
  int count;
};
void display(struct book *);
int main()
{ struct book bl={"letusc", "YPK", 101};
  display(&bl);
  return 0;
}
void display(struct book *b)
{ printf("%s %s %d\n", b->name, b->author, b->count);
}
```

NOTE:

```
struct emp
{ int a;
  char ch;
  float s;
};
int main()
{ struct emp e;
  printf("%u %u %u\n", &e.a, &e.ch, &e.s);
  return 0;
}
```

TC/TC++ O/P ⇒ 65518 (+2) 65520 (+1) 65521

VS O/P ⇒ 1245044 (+4) 1245048 (+4) 124504C

Visual studio is a 32-bit compiler targeted to generate code for a 32bit microprocessor.

⇒ The architecture of this microprocessor is such that it is able to fetch data that is present at an address, which is a multiple of 4, much faster than data present at any other address.

Hence, VS compiler aligns every element of a str at an address that is multiple of 4.

Some programs need precise control over ~~this~~ memory areas where data is placed.

for eg. for reading the contents of the boot sector into the structure, the byte arrangement of structure elements must match the arrangement of the various fields in the boot sector of the disk.

#pragma pack directive fulfills this requirement. This specifies packing alignment for structure members. The pragma takes effect at the 1st structure declaration after the pragma is seen.

```
#include<stdio.h>
#pragma pack(1)
struct emp
{ int a;
  char ch;
  float s;
};
#pragma pack()
int main()
{ struct emp e;
  printf("%u%u%u\n", &e.a, &e.ch, &e.s);
  return 0;
}
```

#pragma pack(1) lets each structure element to begin on a 1 byte boundary as specified by the output of the program. 1245044 1245048 1245049

Console input Output

⇒ I/O facilities are different for different OS.

⇒ console I/O fonc: receive input from keyboard & write output to VDU

⇒ file I/O fonc: perform I/O operations on a floppy disk or hard disk.

Console I/O funcns

⇒ screen + keyboard = console

formatted CI/O fonc ⇒ allows the input read from the keyboard or the output displayed on VDU to be formatted as per our requirements.

Console I/O funcn

formatted funcn			unformatted funcn		
Type	I	O	Type	I	O
char	scanf()	printf()	char	getch()	putch()
char	scanf()	printf()		getche()	putchar()
char	scanf()	printf()		getchar()	
			int	—	—
			float	—	—
			string	gets()	puts()

formatted Console I/O funcns

⇒ These funcn allow us to supply the input in a fixed format and let us obtain the output in specified format.

⇒ printf("format string", list of variables);

⇒ Comments

a) characters

b) conversion specifications %

c) Escape sequences that begin with a \ sign.

- printf examines a format string from left to right, so long as it doesn't come across either a % or \ it continues to dump the characters that it encounters, on to the screen.

format Specifications

Data type		format specifiers
Integer	short signed	%d or %i
	short unsigned	%u
	long signed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal.	%o
Real	float	%f
	double	%lf
	long double	%Lf
character	signed character	%c
	unsigned character	%c
string		%s

Optional Specifiers

Specifier

=> w => digits specifying the field width.
=> . => Decimal point separating field width from precision
no. of places after the decimal point
=> p => digits specifying precision
=> - => minus sign for left justifying the output in the specified field width.

field width specifier ⇒ tells printf() how many columns on screen should be used while printing a value.

eg) %10d → print the variable as a decimal integer in a field of 10 columns

) if the value to be printed happens not to fill up the entire field, the value is right justified & is padded with blanks on left.

%-10d ⇒ left justification & the value will be padded with blanks on right.

) if the specified field width is less than what is required to print the number, the field width is ignored & the complete number is printed.

g) int weight=63;
printf("Wt is %d kg\n", weight);

%2d	63 kg
%4d	63 kg
%6d	--63 kg
%-6d	----63 kg
%1d	63---- kg
	63 kg

ecifying field width can be useful in creating tables of umeric values

) printf("%f %f %f \n", 5.0, 13.5, 133.9);
printf("%f %f %f \n", 305.0, 120.9, 3005.3);

5.000000 13.500000 133.900000
305.000000 120.900000 3005.300000

printf("%10.1f %10.1f %10.1f \n", 5.0, 13.5, 133.9);
printf("%10.1f %10.1f %10.1f \n", 305.0, 120.9, 3005.3);

5.0	13.5	133.9
305.0	120.9	3005.3

%10.1f => specifies that a float be printed left aligned within 10 columns with only one place beyond the decimal point.

=> The format specifiers can be used even while displaying a string of characters.

eg) %20s => reserves 20 columns for printing a string & then printing the string in these 20 columns with right justification.

%-20.10s => left justify the string in 20 columns & print only 1st 10 characters of the string.

Escape Sequences

\ (backslash) => considered as an escape character.

=> It causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

=> \t => takes the cursor to the beginning of next printing zone.

\n newline	\t tab
\b Backspace	\r carriage return
\f form feed	\a Alert
\' single quote	\" double quotes
\\ backslash	

=> Form feed advances the computer stationary attached to the printer to the top of next page.

=> characters that are ordinarily used as delimiters, like ';', '"', \ can be printed by preceding them with a backslash.

Note:- To print the ASCII value of a character, %d

65

General form of Scanf

scanf("format string", list of addresses of variables);

⇒ The values that are supplied through the keyboard must be separated by either blank(s), tab(s) or newline(s).

⇒ Do not include these escape sequences in the format string.

⇒ All format specifications of printf are applicable to scanf().

sprintf() & sscanf() functions

⇒ sprintf() ⇒ similar to printf(). Instead of sending the output to screen, it writes output to an array of characters

eg)
```
#include <stdio.h>
int main()
{ int i=10;
  char ch='A';
  float a=3.14;
  char str[20];
  printf("%d%c%f\n", i, ch, a);
  sprintf(str, "%d%c%f", i, ch, a);
  printf("%s\n", str);
  return 0;
}
```

sscanf() ⇒ allows us to read characters from a string & to convert & store them in variables according to specified formats.

- comes handy for in-memory conversions of characters to values
- The last argument is the string from which reading is to take place

Unformatted Console I/O functions

- There are functions which can deal with single character or strings of characters.

getch() & getche() → prototype in conio.h.

- funcns which will reach a single char the instant it is typed. without waiting for the Enter key to be hit.
- they return the char that has been most recently typed
- getche() echoes the char you typed on screen.
- getch() returns char without printing it on screen.
- getchar()

works similarly & echoes the char but requires Enter key to be typed. following the typing of char.

getchar() → macro
fgetchar() → funcn. } prototypes in stdio.h

Usage

```
getch();
ch=getche();
getchar();
fgetchar();
```

putch() & putchar() & fputchar()

- prints a char on screen.
- they can output only one char at a time

```
char ch='A';
putch(ch);
putchar(ch);
fputchar(ch);
putch('Z');
putchar('Z');
fputchar('Z');
```

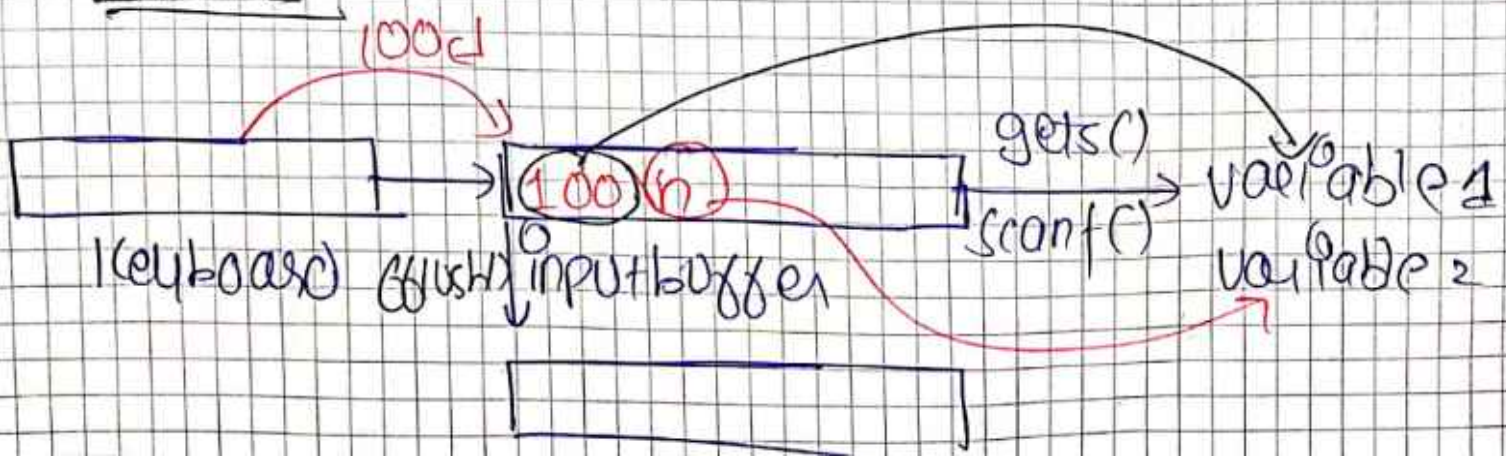
O/P:

AAAZZZ

gets() & puts()

- gets() receives a string from keyboard.
- It considers the string to be terminated when an Enter key is hit.
- Spaces & tabs are perfectly acceptable as a part of the input string.
- getch() gets a newline (\n) terminated string of chars from keyboard & replaces the \n with \0.
- puts() can output only one string at a time.
- Thn attempt to print two strings using puts(), only the first one gets printed.
- Unlike scanf(), gets() can be used to read only one string at a time.

fflush()



File handling in C

- All data stored on disk is in binary form.

File operations

a) Creation of a new file
b) Opening an existing file
c) Reading from a file
d) writing to a file
e) Moving to a specific location in a file (seeking)
f) closing.

Opening a file

- To open a file we call the func^n fopen()
- argument 1: name of file argument 2: mode of opening
  eg) fopen("PR1.c", "r");
  └ string not a char

Tasks performed by fopen():-

(i) Searches on disk the file to be opened.
(ii) loads the file from the disk into memory (buffer)
(iii) sets up the char pointer that points to the first char of the buffer

- All the information like mode of opening, size of file, place in the file where the next read operation would be performed is gathered together by fopen() in a structure called FILE.

fopen() returns the address of FILE str, which is collected in str pointer.

FILE *fp; (declaration of structure pointer)

- The FILE str is defined in stdio.h.

Reading from a file

- One opening the contents of file ~~one~~ is brought into buffer partially or wholly.
- fgetc(); reads file's contents from the memory.

  ch = fgetc(fp);

  Reads the char from the current ptr posn, advances the ptr posn so that it now points to the next char & returns the char that is read.
- ⇒ we generally use ~~get~~ fgetc() in an indefinite ~~loop~~ while loop.

  fgetc() returns a macro EOF once all the characters have been read & we attempt to read one more char.

  EOF macro is defined in stdio.h
- We can use fgetc() & getc() interchangeably.

Trouble in Opening a file

- The file may not open when we try to open it.
- while opening in "r" mode, this may happen bc the file being opened may not be present on disk at all.

  In "w" mode ⇒ failure is due to insufficient disk space, write protected disk, damaged disk etc.
- If file fails to open, fopen() returns NULL defined in stdio.h as, #define NULL 0
- Call to function exit() terminates the execn of program.
- ⇒ If 0 is passed to exit() ⇒ normal termination

  a non zero value represents ⇒ abnormal termination.
- If there are multiple exit points then the value of exit() can be used to find out from where the execution of program got terminated.

= The prototype of exit() is declared in stdlib.h.

```
fp = fopen("PRI.C", "r");
if (fp == NULL)
{ puts("can't open file");
  exit(1);
}
```

= while(1) → Indefinite loop.

```
{ ch = fgetc(fp);
  if (ch == EOF)
    break;
```

Closing the file

= fclose(fp); → closes the file

= On closing, the buffer is removed from the memory.

= One opening a file in "w" mode:-
when we attempt to write char into the file using fputc(), the char would get written to the buffer. When we close this file using fclose(), 2 operations are performed:-
a] char in buffer are written to the file on disk.
b] Buffer is eliminated from memory.

= In case when the buffer's content becomes full before closing the file, the buffer's content would be written to the disk the moment it becomes full.

= Buffer management is done by library functions.

Counting Chars, Tabs, Spaces...

```
#include <stdio.h>
int main()
{ FILE *fp;
  char ch;
  int nol=0, not=0, nob=0, noc=0;
  fp = fopen("PRI.C", "r");
  while(1)
  { ch = fgetc(fp);
    if (ch == EOF)
      break;
    noc++;
    if (ch == ' ')
    nob++;
    if (ch == ('\n'))
    nol++;
    if(ch == '\t')
      not++;
  } fclose(fp);
  return 0;
}
```

A file copy program

```
#include <stdio.h>
int main()
{ FILE *fs, *ft;
  char ch;
  fs = fopen("PRI.C", "r");
  if (fs == NULL)
  { puts("cannot open the file");
    exit(1); }
```

```
ft=> fopen("PR2.C", "W");
if (ft=NULL)
{ puts("cannot open target file");
  fclose(fs);
  exit(2);
}
while(1)
{ ch = fgetc(fs);
  if (ch== EOF)
    break;
  else
    fputc(ch, ft);
}
fclose(fs);
fclose(ft);
return 0;
}
```

(iii) Writing to file

fputc(): writes to file in contrast to putc() which writes to VDU.

⇒ To copy files with extension .EXE or .JPG we need to open these files in binary mode.

File Opening Modes

"r": Searches file [pointer set by fopen() points to the first char in file]
Reading from file.

"w": searches file → exists → contents overwritten
→ one → new file created.
writing to file.

"a": Searches file → if one → new file created
adds new contents at the end of file
fopen() sets pointer that points to the last char in file

"r+": searches file
Reading existing contents + writing new contents + modifying existing contents of the file
pointer points to the first char.

"w+": Searches file → file exists → contents overwritten
→ file one → new file created
writing new contents, reading them back, modifying existing contents

"a+": searches file → file one → new file is created
Reading existing contents + Appending new contents to end of file + cannot modify existing contents.

String (line) I/O in files

fputs(): writes strings to a file

```
#include <stdio.h>
#include <string.h>
int main()
{ FILE *fp;
  char s[80];
  fp=fopen("POEM.txt","w");
  if(fp==NULL)
  { exit(1);}
  while(strlen(gets(s))>0)
  { fputs(s,fp);
    fputs("\n",fp); }
```

```
    fclose(fp);
    return 0;
}
```

⇒ Each string is terminated by hitting Enter.

2) To terminate the execution of prog hit enter at the beginning of a line.
This creates a string of zero length.

2) we must explicitly add new line character at the end of the string b/c fputs() does not automatically add it.

Reading a string from a disk file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ FILE *fp;
  char s[80];
  fp=fopen("poem.txt", "r");
  if (fp==NULL)
    exit(1);
  while (fgets(s,79,fp) != NULL)
    printf("%s", s);
  printf("\n");
  fclose(fp);
  return 0;
}
```

(addresses where string is stored; max length of string; ptr of FILE)

⇒ On reading a line from the file, the string(s) would contain the line contents, '\n' followed by a '\0'. Thus the string gets terminated by fgets() & we do not have to terminate it specifically.

2) When all lines are read & an attempt is made to read a new line, in which case fgets() returns NULL.

The Awkward Newline

=> The discrepancy in the char count occurs because when we attempt to write a "\n" to the file using fputs(), it converts the \n to \r\n combination.

=> If we read the same line back using the fgets() the reverse conversion happens.

=> When we write the first line of the poem on a \n using two calls to fputs(), we'll get

Shining & bright, they are forever, \r\n

When reading the same line back in array s[] using fgets() the array contains

Shining & bright, they are forever, \n\0

=> Thus conversion of \n -> \r\n & vice-versa \r\n -> \n is feature of the stdio func & not of OS. ∴ OS counts \r & \n as separate chars.

Record I/O in files

=> If we desire to write/read a combination of characters, strings & numbers

=> we would organize the dissimilar data together in a structure and then use fprintf() & fscanf() library func to read/write data from/to file.

eg)
```
#include<stdio.h>
#include<conio.h>
int main()
{ FILE *fp;
  char another='Y';
  struct emp
  { char name[40];
    int age;
    float bs;
```

```
};
struct emp e;
fp = fopen ("EMPLOYEE.DAT", "w");
if (fp == NULL)
{ exit(1); }
while (another == 'y')
{ printf ("Enter name, age & basic salary: ");
  scanf ("%s %d %f", &e.name, &e.age, &e.bs);
  fprintf (fp, "%s %d %f\n", e.name, e.age, e.bs);
  printf ("Add another record (Y/N)");
  fflush (stdin);
  another = getche();
}
fclose (fp);
return (0);
}
```

⇒ As in printf(), we can format the data in a variety of ways, by using fprintf(). All format conventions of printf() func^n work with fprintf() as well.

⇒ program to read the employee records created above:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ FILE *fp;
  struct emp
  { char name[40];
    int age;
    float bs;
  };
  struct emp e;
```

```
fp=fopen("EMPLOYEE.DAT", "r");
if (fp==NULL)
{ exit(1); }
~~while(fscanf~~
while(fscanf(fp,"%s%d%f",&e.name,&e.age,&e.bs)
          != EOF)
  printf("%s%d%f\n", e.name, e.age, e.bs);
fclose(fp);
return 0;
}.
```

Text files & Binary files

1) A text file contains only textual information like alphabets, digits, & special symbols. In actuality the ASCII codes of these characters are ~~stored~~ stored in text files.

2) A binary file is merely a collection of bytes.

3) If on opening a file in Notepad, we can make out what is displayed then it is a text file, otherwise it is a binary file.

Q) program to copy text as well as binary files

```
#include<stdio.h>
#include<stdlib.h>
int main()
{  FILE *fs, *ft;
   int ch;
   ~~fs=fopen("newpr1.exe", "rb"~~
   fs=fopen("pr1.exe", "rb");
   if (fs==NULL)
```

```
{ exit(1); }
ft= fopen ("newpgm.exe", "wb");
if (ft== NULL)
{ puts ("cannot open target files");
  fclose(fs);
  exit(2);
}
while(1)
{ ch= fgetc(fs);
  if (ch== EOF)
    break;
  else
    fputc(ch, ft);
}
  fclose(fs);
  fclose(ft);
  return 0;
}
```

⇒ while opening the file in text mode we can use "r" or "rt" but since text mode is the default mode we usually drop the 't'.

⇒ Difference between text & binary mode files:

(i) Handling of newlines

(ii) Storage of numbers.

Newlines

⇒ In text mode: \n → \r\n, conversion before writing to disk. (& \r\n → \n)

⇒ In binary mode these conversions do not take place.

Storage of numbers

⇒ The only func^n available for storing numbers in a disk file is the fprintf() func^n.

⇒ Text & chars are stored one char per byte,

⇒ Numbers are stored as strings of chars.

eg) 12579 → int → 4 bytes but it occupies 5 bytes of memory i.e. one byte per character

1234.56 ⇒ 7 bytes

⇒ If large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient.

The soln is to open the file in binary mode & use the func^n which stores the numbers in binary format. (fread & fwrite)

⇒ It means each number would occupy same number of bytes on disk as it occupies in memory.

Record I/O

Disadvantages :-

a] The no. would occupy more number of bytes (text mode)

b] If no. of fields in the str ↑, writing & reading str using fprintf() & fscanf() becomes clumsy.

Writing records to file

```
#include <stdio.h>
#include <stdlib.h>
int main()
{ FILE *fp;
  char another = 'Y';
  struct emp
  { char name[40];
```

```
    int age;
    float bs;
};
struct emp e;
fp = fopen("EMP.DAT", "wb");
if (fp == NULL)
{ exit(1); }
while(another == 'y')
printf("\nEnter name, age & bs.");
scanf("%s%d%f", e.name, &e.age, &e.bs);
fwrite(&e, sizeof(e), 1, fp);
printf("Add another record (Y/N)");
fflush(stdin);
another = getche();
}
fclose(fp);
return 0;
}
```

⇒ `fwrite(&e, sizeof(e), 1, fp);`

- `&e` → address of structure to be written on disk
- `1` → no. of such element we want to write at a time
- `sizeof(e)` → size of structure in bytes
- `fp` → ptr to file we want to write to.

⇒

⇒

Operation on bits

⇒ The programming languages are byte oriented, whereas the hardware tends to be bit oriented.

Bit Numbering and Conversion

Bit :- short for Binary digit.

4 bits :- nibble ; 8 bits :- Byte ; 16 bits :- word

32 bits :- double word.

⇒ Bits are numbered from zero onwards, increasing from Right to left

Char

⇒ C language doesn't understand binary numbering system.

⇒ A nth index bit has a multiplier of 2^n

⇒ Binary numbers to Hexadecimal numbers

⇒ In hexadecimal numbering system, each digit is built using a combination of digits 0-9 and A-F.

⇒ A-F represents 10-15

⇒ Each hexadecimal digit can be represented using a 4 bit nibble.

0 ⇒ 0000
1 ⇒ 0001
2 ⇒ 0010
3 ⇒ 0011
4 ⇒ 0100
5 ⇒ 0101
6 ⇒ 0110
7 ⇒ 0111
8 ⇒ 1000
9 ⇒ 1001
A ⇒ 1010
B ⇒ 1011
C ⇒ 1100
D ⇒ 1101
E ⇒ 1110
F ⇒ 1111

eg) 1011 0110

B 6

⇒ 0xB6

Bit operations

~ one's complement
>> right shift
<< leftshift
& bitwise and
| bitwise or
^ bitwise XOR (or exclusive or)

These operators can operate on ints and chars but not on floats & doubles.

Showbits()

displays the binary representation of any integer or character value that it receives.

eg)
```
#include <stdio.h>
void showbits(unsigned char);
int main()
{ unsigned char num;
  for(num=0; num<=5; num++)
  { printf("\n Decimal %d is same as binary ", num);
    showbits(num);
  }
  return 0;
}
void showbits(unsigned char n)
{ int i;
  unsigned char j, k, andmask;
  for(i=7; i>=0; i--)
  { j=i;
    andmask = 1<<j;
    k = n & andmask;
    k==0 ? printf("0") : printf("1");
  }
```

One's Complement Operator

⇒ All 1's present in the number are changed to 0s and all 0s are changed to 1's.

⇒ ch = 32

vch = 225 %d

vch = df %x } Hexadecimal equivalents

vch = DF %X }

Right Shift Operator

⇒ It needs two operands.

⇒ It shifts each bit in its left operand to the right. No. of shifts depends upon the right operand

⇒ Blanks thus created are always filled with zeroes.

⇒ If the operand is a multiple of 2 then shifting the operand one bit to right is same as dividing it by 2 ignoring the remainder.

eg) 64>>1 ⇒ 32 27>>1 ⇒ 13

64>>2 ⇒ 16 49>>1 ⇒ 24

128>>2 ⇒ 32

Note:-

⇒ If in a>>b if b is negative then result is unpredictable.

⇒ If a is -ve then its left most bit is 1. On right shifting a it would result in extending the sign bit.

eg) -1>>4 ⇒ 11111111

↳ 11111111

-5>>1 ⇒ 11111101

↳ 11111011

-5>>2 ⇒ 11111110

-5>>8 ⇒ 11111111

Left shift operator

⇒ often used to create a number with a particular bit in it set to 1.

eg) creating a no. with its 3rd bit set to 1

```
#include <stdio.h>
int main()
{ unsigned char a;
  a=1<<3;
  printf("a=%02x", a);
  return 0;
}
```

1 ⇒ 00000001
3 times ⇒ 00001000
bit 1

→ stands for Bit value

⇒
```
#define BV(x) (1<<x)
#include <stdio.h>
int main()
{ unsigned char a;
  a=BV(3);
  printf("a=%02x", a);
  return 0;
}
```

⇒ %02x ⇒ output is printed in 2 columns, with a leading 0, if reqd. ∴ output ⇒ 08

Bitwise AND operator

⇒ while operating, the two operands are compared on a bit-by-bit basis. Hence both the operands must be of the same type. (either ints or chars)

⇒ The second operator is often called an AND mask.

&	0	1
0	0	0
1	0	1

Utility of AND

① To check whether a particular bit of an operand is ON or OFF

② To turn off a particular bit.

eg) 1 0 1 0 1 1 0 1 original
0 0 1 0 0 0 0 0 AND mask
0 0 1 0 0 0 0 0 → 5th bit was ON.

eg) 1 0 1 0 1 1 0 1 original
1 1 1 1 0 1 1 1 AND mask
1 0 1 0 0 1 0 1 3rd bit turned OFF

Bitwise OR operator

- Usually used to PUT ON a particular bit in a number.

	0	1
0	0	1
1	1	1

eg) 1 1 0 1 0 0 0 0 original
0 0 0 0 0 1 1 1 OR mask
1 1 0 1 0 1 1 1 → turned ON

```
#define BV(x) (1<<x)
unsigned char NUM = 0xC3;
NUM = NUM | BV(3);
```

Bitwise XOR operator

⇒ The

^	0	1
0	0	1
1	1	0

⇒ XOR operator is used to toggle a bit ON or Off.

⇒
```
#include<stdio.h>
int main()
{
```

Showbits()

⇒
```
void Showbits(unsigned char n)
{ unsigned char i,lc,andmask;
  for(i=7; i>=0; i--)
  { andmask = 1<<i;
    lc = n & andmask;
    lc==0 ? printf("0") : printf("1");
  }
}
```

Bitwise Compound assignment operators

a ⇒ a<<1 ⇒ a<<=1
b ⇒ b>>2 ⇒ b>>=2
c = c|0x2A ⇒ c|=0x2A
d = d&0x4A ⇒ d&=0x4A
e ⇒ e^0x21 ⇒ e^=0x21
⇒ ~= One as ~ = unary operator

Miscellaneous features

Enumerated data types

⇒ We can invent our own data type & define what values the variable of this data type can take

eg enum mar_status
{ single, married, divorced, widowed } declares data type
↓ ↓ ↓ ↓
0 1 2 3
}; enum mar_status person1, person2; } declared data types of this variable

⇒ person1 & person2 can have 4 possible values: single, married, divorced, widowed.

⇒ We can't use values that aren't in original declared.

⇒ Internally, the compiler treats enumerators as integers.

⇒ Each value corresponds to an integer starting from 0.

⇒ Way of assigning nos can be overridden by the programmer.

enum mar_status
{ single=100, married=200, divorced=300, widowed=400.
}; or single=100, married, divorced, widowed
101 102 103
enum mar_status person1, person2;

Uses of Enumerated data types

```
#include <stdio.h>
#include <string.h>
int main()
{ enum emp_dept
  { assembly, manufacturing, accounts, stores
  };
  struct employee
```

```
{ char name[20];
  int age;
  float bs;
  enum emp_dept department;
};
struct employee e;
strcpy (e.name, "Lothar Matthews");
e.age = 28;
e.bs = 5575.50;
e.department = manufacturing;
printf ("Name = %s \n", e.name);
printf ("Age = %d \n", e.age);
printf ("Basic salary = %f \n", e.bs);
printf (" Dept = %d \n", e.department);
if (e.department == accounts)
  printf ("%s is an accountant \n", e.name);
else
  printf ("%s is not an accountant \n", e.name);
return 0;
}
```

O/P:-

```
Name = Lothar Matthews
Age = 28
Basic Salary = 5575.50
Dept = 1
Lothar matthews is not an accountant.
```

⇒ There is no way to use the enumerated values directly in input/output functions like printf() & scanf().

Necessity of enums

→ macros can be used as alternatives to enums

eg)
```
#include <stdio.h>
#define ASSEMBLY 0
#define MANUFACTURING 1
#define ACCOUNTS 2
#define STORES 3
int main()
{ struct employee
  { char name[30];
    int age;
    float bs;
    int department;
  };
  struct employee e;
  strcpy (e.name, "Cothar Mathews");
  e.age = 28;
  e.bs = 5575.50;
  e.department = MANUFACTURING;
  return 0;
}
```

→ Macros have global scope

→ enums can be either global or local.

---

eg)
```
enum month
{ jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};
enum month m1, m2, m3;
```

typedef()

⇒ Used to redefine the name of existing variable type

eg) typedef unsigned long int TWOWORDS
TWOWORDS var1, var2; → declaration instead of
unsigned long int var1, var2;

⇒ usually, uppercase letters are used to make it clear that we are dealing with a renamed data type.

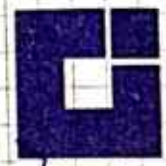
eg)
```
typedef struct employee
{ char name[30];
  int age;
  float bs;
} EMP;

EMP e1,e2;
```

⇒ typedef can also be used to rename pointer data types

eg)
```
struct employee
{ char name[30];
  int age;
  float bs;
};
typedef struct employee * PEMP;
PEMP p;
p -> age = 32;
```

typecasting

→ used to force the compiler to explicitly convert the value of an expression to a particular data type.



eg) #include <stdio.h>
int main()
{ float a;
int x = 6, y = 4;
a = x/y;
printf ("value of a = %f\n", a);
return 0;
}

value of a = 1.000000

#include <stdio.h>
int main()
{ float a;
int x = 6, y = 4;
a = (float) x/y;
printf ("value of a = %f\n", a);
return 0;
}

value of a = 1.500000

→ The expression (float) causes the variable x to be converted from type int to float before being used in the division operation.

→ value of a doesn't get permanently changed as a result of typecasting. Rather it is the value of the expression that undergoes type conversion whenever the cast appears.

eg). #include <stdio.h>
int main()
{ float a = 6.35;
printf ("value of a on typecasting = %d\n", (int) a);
printf ("value of a = %f\n", a);
return 0;
}

⇒ 6

⇒ 6.350000

Bit fields

⇒ When there are several variables whose maximum values are small enough to pack into a single memory locations, we can use bit fields to store several values in a single integer.

⇒ eg) struct employee
{ unsigned gender : 1; (M/F) → 0/1
unsigned mar_status : 2; 00 10 } 2bits
unsigned hobby : 3; → 8 hobbies
unsigned scheme : 4; → 15 schemes +1 for no scheme
};

⇒ : tells compiler that we are talking abt bit fields & the number after it tells how many bits to allocate for the field.

⇒ total memory allocated → 10 bits ⇒ 16 bits ⇒ 2 bytes

⇒ used to consume memory efficiently.

⇒ when it is known that value of a field or group of fields will never exceed a limit or is within a small range.

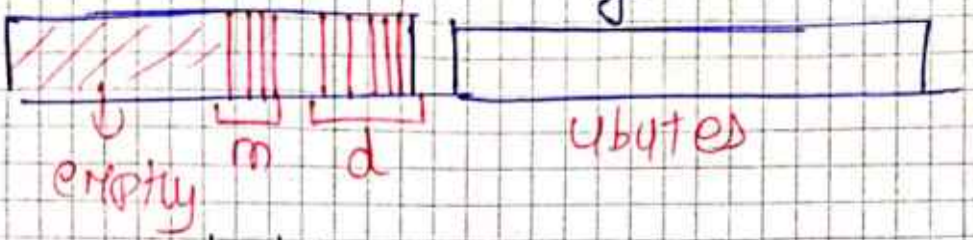
⇒ can be used in structure, union.

eg) struct date
{ unsigned int d; → 4byte } 12 bytes / 96 bits
unsigned int m; → 4byte
unsigned int y; → 4byte
};

data ∈ [1, 31] ⇒ 31 ⇒ 11111 ⇒ 5 bits
m ∈ [1, 12] 12 ⇒ 1100 ⇒ 4 bits.

struct date
{ unsigned int d : 5; } 4 bytes } total 8 bytes
unsigned int d : 4; } 4 bytes
unsigned int y;

=) Blocks get allocated in multiples of 4bytes



सी-डॉट
C-DOT

Pointer to functions

=) C functions have addresses

eg) #include <stdio.h>
void display();
int main()
{ printf("Add of func display is %u\n", display);
display();
return 0;
}
void display()
{ printf("~");
}

O/P => Address => 1125

Invoking a func using a pointer to a func

#include <stdio.h>
void display();
int main()
{ void (*func_ptr)();
func_ptr = display; // assigning address
printf("Add is %u", func_ptr);
(*func_ptr)(); // invokes the func display().
return 0;
}
void display()
{ puts("\n~"); }

O/P:
Add => 1125

⇒ void (*func_ptr) ⇒ func_ptr is a pointer to a func, which returns nothing.

⇒ invoking ⇒ (*func_ptr)(); or func_ptr();

Uses:

1. callback mechanism (in windows programming)
2. binding funcs dynamically, at runtime in C++.

Funcs returning pointers

⇒ int *fun(); → declarn fun() returns a pointer to an int.

```
int *fun()
{ static int i=20;  → defn
  return (&i);
}
```

Unions

⇒ are derived data types like structures.

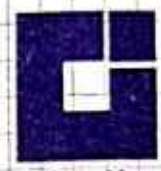
⇒ Both str & unions are used to grp a no. of diff variables together. But while a str enables us treat a number of diff variables stored at different places in memory, a union enables us to treat the same space in memory as a number of different variables.

⇒ Union offers a way for a section of memory to be treated as a variable of one type on one occasion and as a different variable of a different type on another occasion.

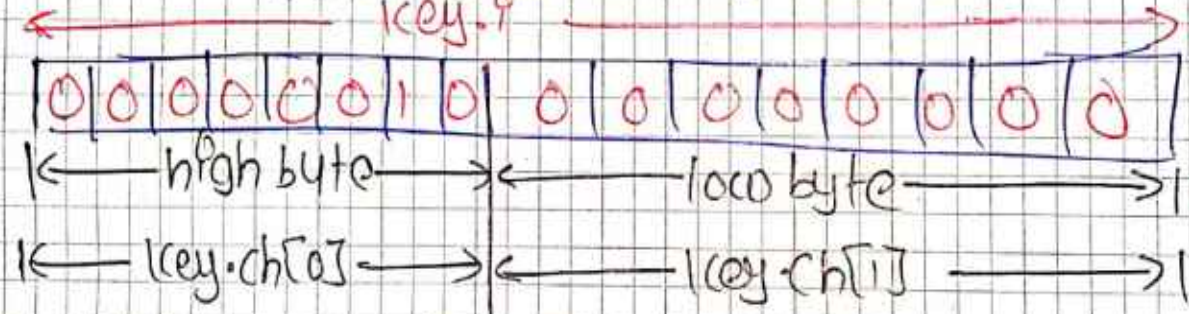
eg)

eg) #include <stdio.h>
int main()
{ union a
{ short int i;
char ch[2];
};
union a key;
key.i = 512;
printf("key.i = %d\n", key.i);
printf("key.ch[0] = %d\n", key.ch[0]);
printf("key.ch[1] = %d\n", key.ch[1]);
return 0;
}

o/p =>
key.i = 512
key.ch[0] = 0
key.ch[1] = 2



key
512 => 0000 0010 0000 0000



if the no is stored in this manner then
o/p => key.ch[0] = 2 key.ch[1] = 0
The op is opposite b/c some cpu follow little endian architecture, when a 2 byte no. is stored in memory, the low byte is stored before the high byte
=> In cpu with big endian architecture this reversal of bytes does not happen.

⇒ We can't assign different values to the different union elements at the same time.

⇒ If we assign a value to key.i, it gets automatically assigned to key.ch[0] & key.ch[1] & vice versa.

eg) key.i = 512;
printf("key.i = %d\n", key.i);
printf("key.ch[0] = %d\n", key.ch[0]);
printf("key.ch[1] = %d\n", key.ch[1]);
return(0);
}

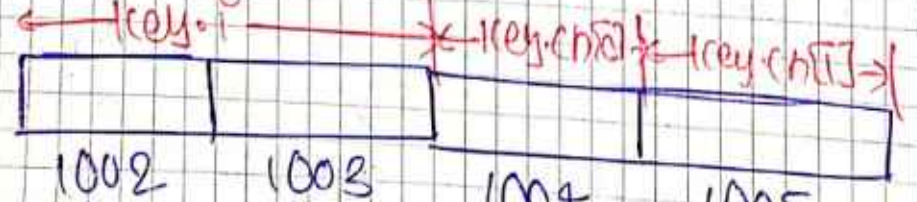
O/P:- key.i = 512 key.ch[1] = 2
key.ch[0] = 0

key.ch[0] = 50;
printf("-1 == ");
printf("-1 == ");
return 0;

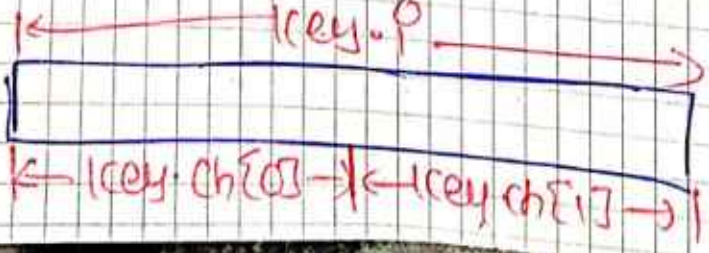
O/P:- key.i = 562
key.ch[0] = 50
key.ch[1] = 2

⇒ There can exist a union, each of whose element is of different size. In such a case, the size of union variable will be equal to the size of largest element in the union.

str:-



union:-



Union of Structures

- A union can be nested within another union or structure.
- There can be a structure in an union.

eg) #include <stdio.h>
int main()
{ struct a
{ int j;
char c[2];
}
struct b
{ int i;
char d[2];
};

union z
{ struct a key;
struct b data;
};

Utility of Unions

- To avoid wastage.

```
struct employee
{ char n[20];
  char grade[4];
  int age;
  char hobby[10];
  int crcardno;
  char vehno[10];
  int dist;
};
struct employee e;
```

name, grade, age
if grade → HSC → hobby name
└→ cr card no
if grade → SSLC → veh no
└→ dist from co.

both cannot be used simultaneously leading to wastage

```
struct info1
{ char hobby[10];
  int crcardno;
};
struct info2
{ char vehno[10];
  int dist;
};
union info
{ struct info1 a;
  struct info2 b;
};
struct employee
{ char n[20];
  char grade[4];
  int age;
  union info f;
};
struct employee e;
```

The Volatile Qualifier

- When we define variables in a program the compiler may optimize the code that uses the variable [by using cpu register to store value of var]
- If we declare the variable as volatile, then it serves as a warning to the compiler that it should not optimize the code containing this variable.
- The variables value would be loaded from memory into register operations would be performed on it & the result would be written back to the memory allocated to the variable.

declaration:- volatile int i;

⇒ We may declare the variable as volatile when the variable is not within the control of the program & is likely to get altered from outside the program.

eg) volatile float temperature;

temperature might be modified through the digital thermometer attached.

Functions with variable number of arguments.

⇒ Three macros available in the file stdarg.h called the va_start, va_arg & va_list are used.

⇒ They provide a way for accessing the arguments of the func^n when a func^n takes a fixed no. of arguments followed by a variable no. of arguments.

fixed no of arg → accessed in normal way.

optional arguments → accessed using va_start & va_arg

va_start:- used to initialize a pointer to the beginning of the list of optional arguments.

va_arg:- used to advance the ptr to the next argument

eg)
```
#include<stdio.h>
#include<stdarg.h>
int findmax(int, ...);
int main()
{ int max;
  max= findmax(5,23,15,1,92,50);
  printf("max = %d\n", max);
```

... indicates that the no. of arguments after the first argument could be variable.

5 → count of arguments that follow the first argument.

```
max = findmax(3,100,200,29);
printf("max = %d\n", max);
return 0;
}
int findMax(int tot_num, ...)
{ int max, count, num;
  va_list ptr; → pointer declaration
  va_start(ptr, tot_num); ⟹ sets up ptr that points to the 1st variable argument in the list
  max = va_arg(ptr, int);
  for(count = 1; count < tot_num; count++)
  { num = va_arg(ptr, int);
    if(num) max)
      max = num;
  }
  return(max);
}
```

⟹ this statement assigns the integer being pointed to by ptr to max.

max = 23, &
ptr points 5.