**C Programming**

**Introduction**

* This is the mother language of all the other languages like C++ , JAVA , Python .
* C is a **structure oriented language** unlike C++ , C# and JAVA .

**Reasons of studying C in the age of OOP languages**

* C is faster
* Major parts of Operating systems like Windows , UNIX , Linux and Android are written in C .
* All the device driver programs are exclusively written in C
* Embedded programs in Consumer devices like microwaves , Ovens are written in C .
* Popular gaming Frameworks like DirectX are written in C .
* C provides various language elements that make one to have very close interaction with hardware devices .
* It provides good base for learning other modern languages like C++ , C# , JAVA etc.

**Specialities :-**

* Time and memory efficient.
* Can Interact with hardwares very closely .

**Important terms and definitions : -**

* Keywords -> words which carry special/fixed meaning . Its meaning is already been explained to the compiler/computer .
* There are 32 keywords available in C .
* Semicolon ‘;’ acts as the statement terminator .
* In main() function returning 0 means success .
* **#include** is a **preprocessor directive**
* %d , %c , %fare the **format specifiers**
* **Compiler** converts the the high level language to **machine language**
* **&(ampersand)** is ‘ Address of ’ operator . It gives location number(address) used by the variable in memory
* Variables are the **names given to memory locations .**
* Every compiler is **Platform specific** .
* A platform is a combination of specific OS and microprocessor(i.e **OS + Micropreocessor = Platform**)

**C Instructions**

* **Program** is a set of instructions in order to achieve some certain tasks .
* Types of instructions : -
  + Type declaration instructions – used to declare the type of variables .

ex:- int a , float b etc

* + Arithmetic Instruction – used to perform arithmetic instructions on constants and variables .

s = a + b + 32 , t = 32 \* / a etc

* + Control Instruction – used to control the sequence of execution of various statements in a program
* **Arithmetic operators** are ‘+’ , ‘-’ , ‘\*’ , ‘/’
* Modulus operator(%) is not used on float
* Using % , sign of the remainder is always same as the sign of numerator
* **ASCII codes** are used to represent any character in the memory , ex:- ASCII code for ‘a’ and ‘b’ are 01100001 and 01100010 respectively
* **ASCII value**s are the decimal equivalent of the **ASCII codes ,** ex:- ASCII values of ‘a’ and ‘b’ are 97 and 98 respectively .
* **‘b’ + ‘a’** , here , addition is performed on the ASCII values of the characters i.e , **97+98**
* **pow(a,b)** is a standard **library function** declared in **<math.h>** header file , used for exponentiation (i.e a raised to power b) .
* **pow()** only works with real numbers .
* **sqrt()** function is used for getting square-root.
* **#include** is a preprocessor directive
* In an operation between **real and integer** always yield **real** . To achieve this integer is first promoted to real .
* **Hierarchy of operators** (priority / precedence):-
  + Parantheses ‘()’
  + \* / %
  + + -
  + =
* **Priority can be changed using ()**
* **Associativity of Operators**
  + When an expression contains two operators of equal priorioty/precedence , the tie between them is settled using associativity of operators
  + \* / follows **left to right** associativity , ex:- 3/2\*5 , / is in left of \* , hence / will be processed first the \*
  + = follows **right to left** associativity , ex:- a=b=2 , rightmost ‘=’ will be performed first and then left one , i.e **at first b = 2 , then a = b**
* **Control Instruction**
  + Specify the order in which the various instructions are to be executed in a program .
  + Types:-
    1. Sequence Control Instruction – It ensures that the instructions are executed in the same order in which they appear in the program
    2. Decision Control Instruction – Takes the decision as to which instruction is to be executed next
    3. Repetition or Loop Control – helps to execute group of statements repeatedly
    4. Case Control Instruction – It allows us to perform instructions as per our choice . OR , It executes only specific **case statements** based on the **switch expression** .

**Decision Control(Control flow/Conditional Statements)**

- **if-else** and **switch-case** statement are the two main decision control instructions

if(i==5) **;**  ----------- 1

printf(“You entered 5”) **;** ----------- 2

- Statement 1 and 2 will act as **two different statements** , since null operator ( ; ) after if() statement gets the printf() operator out of the if block

- Hence , printf() is bound to get executed no matter if() gives true or false . And there **won’t be any error**

xcxc

- format of if-else :-

if(**condition** is true)

statement 1;

else

statement 2 ;

- We use **relational operators(< , > , <= , >= , == , !=)** to express the conditions in if-else statement

- ‘=’ is used for assignment whereas ‘==’ is used to compare two quantities

- We can use if() without else , but else must always be associated with an if()

- Any non-zero number is true , 0 is false .

- **‘;’** is a null statement . It doesn’t do anything on execution

- **Logical operators , &&(AND) , ||(OR) and !(NOT)** allow to get two or more

conditions

if(i=5)

statement ;

- **this statement won’t throw error**

- In an **else if** statement , the last else is optional

- **!** , this logical NOT operator reverses the result

- **!(NOT)** comes at the top in priority list of the operators

- **!** is an **unary** operator , since it needs only single operand and **rest are binary** operators

***if(!flag)*** is same as ***if(flag==0)***

***!(a <= b)*** *is same as* ***(a>b)***

***!(a>=b)*** *is same as* ***(a<b)***

**Conditional Operators**

- **?** and **:** constitute conditional operators .

- These are called as ***ternary operators*** as they take 3 arguments .

format :- **expression 1 ? expression 2 : expression 3**

**- If express 1 is true (non-zero) , then express 2 will be returned otherwise express 3 will be returned .**

**-** The limitation of conditional operators is that **only one C statement** can occur after **?** or **:**

***a>b ? g = a : g = b ;* 🡨 error , “Lvalue Required” ; to fix : a>b ? g = a : (g = b) ;**

- **sizeof()** is an operator which gives **number of bytes** occupied by an entity

**Loop Control**

- Loop control instruction is used to repeat a set of statements either a specified number of times or till a particular condition keeps satisfying .

- the variable used for loop control is called **‘loop counter ‘** or **‘index variable’**

- **for , while and do-while** are the kinds of loop control statements.

- **j = ++i** , first increments i then assigns the incremented value to j

- **while(++i<10) ,** first increments i then checks the condition

- **while(i++<10) ,** first checks the condition then increments

- **+= , -= , \*= , /= , %=** are called compound assignment operators .

- in **for loop** , **counter initialization , testing and incrementation** is done in a single line only , unlike in while loop .

- multiple initialization ,incrementation and testing could be done in for loop

example:- **for(i=1 , j = 2 ; i<=10 && j<=24 ; i++ , j+=3)**

- **break** statement is used to **terminate the loop** inside which it is .

- **continue** keyword **skips/abandons** rest/following instructions in the loop and goes for the next iteration of the loop.

- when **continue** statement is encountered the **control** is passed to the **beginning of the loop**

while with **break** , control passes to the **first statement after the loop** .

- Since **do-while** loop checks the condition after the execution of the statements , it executes its statements at least once even if the condition fails for the first time .

- **fflush()** is used to remove or flush out any data remaining in the **buffer** .

- we have to pass **stdin** as argument into the **fflush()** , since we have to flush out the standard input related buffer i.e **fflush(stdin)** .

- After taking the **number input** , the **scanf()** assigns the number to variable and **keeps the enter key unread in the Keyboard buffer** . Hence next time when we supply any character scanf () will read **the enter** from the buffer instead of taking character input . Hence to avoid this we use **fflush(stdin) .**

**-** sometimes , there might be a situation when in a loop the **number of iterations would be unknown** example: **while(x == 5) , while(y == ‘y’) ,** such loops are called **odd loops .**

- **usual uses :-**

while - to repeat something an **unknown number of times**

for - to repeat something a **fixed number of times**

do-while - to repeat something **at least once**

- **for(; ;)** and **while(1)** are infinite loops and **while()** is an error

***- what can we done using one loop can always be done using the other two***

**Case Control**

- **switch-case-default** control statement allows us to make a decision from the number of choices

- The switch statement is very useful for menu driven programs

- In the **switch()** , **float**,double expression can’t be tested (**only constant char , long int and int is used**)

- Switch-case works faster than if-else ladder

- break takes the control out of the switch() but continue doesn’t take the control to the beginning of switch .

- cases in the switch must always be unique

- **goto** keyword can take the control form any place to any other place within the function

- **exit()** from **<stdlib.h>** is a standard library function used to terminate the program execution

**Functions**

- Function is a self-contained block of statement that performs some tasks and returns back the evaluated result to the calling function.

- Basically , we create functions to avoid repeatedly rewriting the same codes and for the ease of debugging and designing

- Creating a function consists three tasks done , **Prototype declaration** , **Function Call** , **Function Definition** .

- Arguments passed to a function by the calling function are **Actual arguments** and that received by the function are **Formal arguments .**

- In C , program always begins with **main()**

- **Library functions** - printf() , scanf() etc , commonly required functions grouped together and stored in library files

- **User-defined** functions are freshly defined by the users.

- **return** statement transfers the control back to the calling function and terminates the current active function.

- A function can return only one value .

int a = 1 ;

printf(“%d%d%d”,a,++a,a++); 1 3 3(order of passing 1 to printf())

Output : 3 3 1 (order of printing)

**In C , during a function call , the arguments are passed from right to left .**

- Order of passing arguments in function is compiler-dependent not language-dependent

- Headerfiles contain the library-functions prototypes

- For example , <stdio.h> header file contains the prototypes of all the input/output functions , and prototypes of all mathematical functions are provided into the header file ‘math.h’

printf(“%d%d%d”,a,b);

printf(“%d”,a,b);

**Both the statements would be compiled without any error as printf() accepts variable arguments**

**Function Call Stack**

- Whenever a function is called , system allocates a chunk of memory to that function in which all the function codes , parameters , local-variables etc. are stored , and this memory chunk is call **Stack Frame .**

**-** As the name suggests , all the Stack Frames are stored in a stack in the memory .

- Whenever a new function is called , its frame gets added to the top of the stack , it is called **Pushed in Stack .** And the Topmost frame is called the **Active Frame**

- When the function terminates and return the control back to the calling function , then its frame gets removed from the stack and this is called **Popped from the stack**

fun1(){

fun2();

return;

}

fun2(){

fun3();

return;

}

fun3(){

return ;

}

Active Frame

|  |
| --- |
| fun3 |
| fun2 |
| fun1 |

Call Stack

- Concept of Function Call Stack especially is very important for Recursion

**Pointers**

location name

a

**int a = 5 ;**

The above declaration tells the compiler :-

1 . **To reserve a location in memory** to store int value

2 **. Associate the name a** with the memory location

3 . Store the value 3 at the memory location

location value

5

location address

6453218

- Pointers are the variables which hold the addresss of other variables

- To print the address of the variable we use **&a(& as ‘address of’ operator)** expression and **%u** format specifiers(used for unsigned integer) i.e **printf(“%u”,&a);**

**- ‘\*’** is called ‘**value at address/indirection**’ operator , as it gives the value stored at a particular address

- Addresses or Location number are always whole numbers , whether it is of character , float or int variable , hence pointers always store whole numbers

- ‘**call by value** ’ and ‘**call by reference**’ are the two types of function calls . **Mixed call** refers to pass value as well as addresses

- In call by value , we pass the copy of actual argument to the formal arguments and in this way , the changes made to the formal arguments in called function have no effect over the actual arguments in calling function

- In call by reference , the addresses of the the actual arguments are copied to the formal arguments and this way we get the access to the actual arguments and thus we can manipulate their values .

- By using pointers or by making call by reference , we are able to **indirectly return multiple values at a time** from a called function to a calling function . As , what we make changes in the values of the passed addresses to the called functions can be used and reflected also in the calling function .

- As an integer variable **‘a’** takes 4 bytes of memory , **‘&a’** gives the **address of first byte** only .

**Recursion**

- Recursion is a process in which a function calls itself **directly or indirectly**.

- Recursion is the alternative for loops and is relatively slower .

- As the concept of **Function Call Stack** will be applicable in the recursion , on each recursive call the **activation record** will be maintained inside the **stack frame** of stack .

- Each **activation record** will consist the locals of the function i.e , fresh-parameters , local variables , return address of the caller .

- Fresh set of variables are born during each function call irrespective of normal or recursive call .

- There must be the provision/way to terminate/stop the recursive function and **return back the control** if any particular condition gets satisfied(and this condition is called **Base Case**) otherwise one would fall into an infinite loop .

- We must use the if-else block under the recursive body , so that the base-case could be fitted in either of if and else blocks .

- After meeting the Base Case , the current activation record gets popped out and the control returns back to the previous caller and the Stack starts getting unwound/unroll until all the stack-frames get popped out .

**Types of Recursion**

**Direct Recursion** - When fun() calls itself i.e fun() , then it is **direct recursion** .

**Indirect Recursion** - When fun() calls another function first , suppose fun2() and then fun2() calls again fun() directly or indirectly then it is an **indirect recursion**.

**Tail Recursion** - A recursive function is said to be tail recursion if the **recursive call is the** **last thing done by the function**. There is no need to keep the track of the previous state or there is nothing left to evaluate after returning back .

**Non-Tail Recursion** - A recursive function is said to be non-tail recursion if the **recursive call is not the last thing** done by the function.After returning back there is something left to evaluate.

int fun(int n)

{

if(n==0)

return ;

**return fun(n-1);**

**printf(“Something”);**

}

int main()

{

fun(3);

}

**Non-Tail Recursion**

void fun(int n)

{

**if(n==0)**

**return ;**

else

printf(“Something”);

**return fun(n-1);**

}

int main()

{

fun(3);

}

**Tail Recursion**

**Base Case**

Print function is to get evaluated after recursive call after returning back

Nothing to get evaluated after recursive call

**Data-Types Revisited**

- **char , int** and **float** are the **Primary datatypes** , which could further consist corresponding **sub-types** .

- To completely define a variable we need : **1) Type of variable 2) Storage class of variable**

**Integers - short(2B) , int(4B) , long(4B) , signed , unsigned**

- **short** and **long** are the two variations/sub-types of **int** , which acquire atleast , **2 bytes** and **4 bytes** of memory which varies with compiler .

- **short** wouldn’t be bigger than **int** and **int** wouldn’t be bigger than **long .**

- short , int and long have further variations as , **signed** and **unsigned .**

**- In signed integers , negative numbers are stored as 2’s complement and the leftmost bit(0/1) signifies the sign of the integer as +ve/-ve .**

**- In unsigned integers all bits contribute to value unlike signed .**

**-** An int variable can store **-2147483648 to +2147483647** in **signed** and **0 to 4294967295** in **unsigned variation**

- if we want an **int** to be treated as a **long** than we need to add **‘l’ or ‘L’** as the suffix of the number

i.e . 25(int)->25L(long)

- Number without decimal point is **by default an int** . We need proper suffix to change it , i.e, **365l (for long)** etc .

**- Ranges and sizes of int , short , long are compiler dependent**

**Chars - signed , unsigned**

- char variables occupy 1 byte

- While assigning any character to a char variable , the binary equivalent of their ASCII/Unicode value gets stored.

- Range of **signed char** is **-128 to 127** and of **unsigned char** is **0 to 255**

- If a char variable is assigned with any out of bound value then the appropriate value gets assigned from the opposite bound i.e.**char a = 128**(it exceeds the upper bound i.e **>127**) , it automatically gets assigned with **-128**

**Reals - float , double , long double**

**- float** occupies **4 bytes** and range form **-3.4e38 to +3.4e38**

**- double** occupies **8 bytes** and range from **-1.7e308 to +1.7e308**

- **long double** occupies **10 bytes** and range from **-1.7e4932 to +1.7e4932**

- 3.14 is a **double** by default , to make it treated as **float** we need to add **‘f’** as the suffix , i.e **3.14f**

- Number with a decimal point is **by default a** **double** . We need proper suffix to change it i.e , **3.14f (for float)**

**Storage Classes in C**

- To fully define a variable , one needs to mention its **‘storage class’** alongwith its type .

A storage class tells :-

**Storage** : where the variable would be stored

**Default value** : what would the default initial value of the variable

**Scope** : in which functions/blocks the value of the variable would be available

**Life** : How long would be the variable exist

There are 4 storage classes in C :-

1 . Automatic 2 . Register 3 . Static 4 . External

**Automatic Storage class(auto int x)**

Storage : Memory(**in Stack**)

Default value : **Garbage value(unpredictable)**

Scope : Local to the block({})

life : till control is in the block in which variable is defined

- Automatic variables defined in a function are created in a stack each time the function is called .

**Register Storage class(register int x)**

Storage : **CPU registers**

Default value : Garbage value

Scope : Local to the block ({})

Life : till the control is in the block in which variable is defined

- Value stored in a CPU register can always be accessed faster than the one that is stored in memory . Hence , if a variable is used frequently in a program then better declare its storage class as **register**

**Static Storage class(static int x)**

Storage : Memory(**in Data Segment**)

Default value : 0

Scope : Local to the block in which the variable is defined

**Life : till execution of program doesn’t end**

**- Value of the variable persists between different function calls(of the same function) . Or , it continues to live with its latest value until the execution of program gets end .**

**-** static variablesare created in a place in memory called **‘Data Segment’** instead in stack **. Such variables die only if the program execution ends .**

**External Storage Class(extern int x)**

Storage : Memory Default value : 0

**Scope : Global Life : till execution of program doesn’t end**

**- External variables are declared outside of all functions and are available to all the functions of the same file as well as to the functions of the other files too .**

auto int i ;

register int k;

static int j ;

extern int l ;

definitions

declaration

- Out of locals and globals the **most local** variable gets **the priority** .

- Definition of variable reserves space , declaration doesn’t .

- Redeclaration is ok , but redefinition is not .

**The C Preprocessor**

- C Preprocessor is a program that processes our program before it is passed to the compiler

- Preprocessor works on the source code and creates **‘Expanded Source Code’** as per the **preprocessor directive** used in the source code

- Each Preprocessor directive begins with # symbol

Followings are some preprocessor directives:-

1 . Macro Expansion 2 . File Inclusion 3 . Conditional Compilation 4 . Miscellaneous Directives

**Macro Expansion**

**-** it is written as , **#define PI 3.1428 ,** in which PI is ‘macro template’ and 3.1428 is ‘macro expansion’

- During preprocessing every macro template gets replaced by its macro expansion .

- by using #define directive , no need to make changes at every occurrences of constant if needed .

- Compiler generates more faster and compact code for constants than for variables

- **Macros with arguments** : #define AREA(x) (3.14\*x\*x) , #define ISDIGIT(y)(y>=48 && y<=57)

- No space between macro template and its argument , and macro expansion should be enclosed within parantheses . Otherwise ,

**#define SQUARE(y) y\*y would exapand z = SQUARE(3+1) into z = 3+1\*3+1 (instead of 4\*4)**

- Macros can be split into multiple lines with ‘\’

- Expanded source code is stored into **file\_name.i** file and to generate it type **cpp(C PreProcessor) file\_name.c** in cmd and after generating the source code it gets stored into **file\_name.o** .

- Macros are faster than function but Functions occupies lesser space

**File Inclusion**

- It includes one file to another , **#inlcude“headerfile\_name”**

**-** Prototypes of all the library functions are stored into header files

- For example , prototypes of all the mathematical realated functions are stored in ‘math.h’ and of console input/output in ‘conio.h’

- Two ways to write #include statement , **#include”headerfile\_name”** and **#include<headerfile\_name>**

- In the first way , headerfile is searched into **current directory + specified list of directories** as mentioned in the **include search path** and in the second way , header file is searched only into specified list of directories .

- Include Search Path is a list of directories that would be searched for the file being included

**Conditional Compilation**

- Compiles the code only if the condition is true

**1 . #ifdef** and **#endif (preprocessing commands)**

#ifdef **macroname**

statement 1;

statement 2;

statement 2;

#endif

- If the macroname is defined then the block under #ifdef will be processed otherwise will be **skipped**

- it is used for “commenting-out” purpose

- it is also used to make the programs portable i.e to make them work on two different systems

ex :- int main()

{

#ifdef INTEL

code runs on INTEL PC

#else

code for MOTOROLA PC

#endif

common code

}

- In this example , after compilation the code it would only work for Motorola and if we wanna run the

code on an INTEL pc , we would have to define the INTEL macro .

**2 . #ifndef**

- It works opposite of #ifdef i.e , if the macro is not defined the codes under the block get processed otherwise not

- It could be used to prevent the multiple inclusions of same header files .

**3. #if** and **#elif**

- It is used to test whether an expression evaluates to a non-zero value or not .

- If the expression of #if directive gives non-zero value then its subsequent statements are compiled otherwise skipped .

ex:- #if TEST <= 5

statement 1;

statement 2;

#else

statement 3;

statement4;

#endif

- here , if the given expression is true ,then stm1 and stm2 would be compiled otherwise stm 3 and 4 would be compiled .

**Miscellaneous Directives**

**a . #undef -** used to undefine a macro which is already #defined i.e #undef macro

b. **#pragma** - used to turn-on or off certain features

1 **. #pragma startup** and **#pragma exit** - these directives 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();

Output:-

Inside fun1

Inside main

Inside fun2

**#pragma startup fun1**

**#pragma exit fun2**

int main()

{

printf(“Inside main”);

return 0;

}

void fun1(){printf(“Inside fun1”);}

void fun2(){printf(“Inside fun2”);}

- fun1() and fun2() should neither receive nor return value

- If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called

2.**#pragma warn** - this directive tells the compiler whether or not we want to suppress a specific warning.

**#pragma warn -rvl** : suppress the warnings regarding return value

**#pragma warn -par** : suppress the warning for parameter not used

**#pragma warn -rch** : --------------------------unreachable code

- If we replace ‘-’ by ‘+’ then these warnings would be flashed on compilation

**There are 4 phases for a C program to become an executable**

**1.Pre-processing (done by preprocessor)**

- Source code(file.c) is passed through Preprocessor and the Expanded source code is created .

- **Removal of comments , Expansion of Macros** and the **Included files(header files)** and **Conditional Compilation** are taken place during this phase .

- The output if preprocessor is stored in **file.i (Expanded Source Code)**

**2.Compiling (done by compiler)**

- Expanded Source Code(file.i) is passed to compiler and the **Assembly Language Code** is created

- Out file is **file.s** which is in assembly level instructions

**3.Assembly(done by Assembler)**

- Assembly Language Code(file.s) is passed to Assembler and **file.o**  , an object-file is created

- file.o contains relocatable object code in Machine Langugae .

- In this phase only existing codes are converted into machine language , **the function calls like printf() are not resolved yet** .

**4.Linking(Linker)**

- It takes Relocatable Object code from assembler and Object code of library functions and creates Executable code file in machine language i.e **file.exe**.

- Genearally , it takes one or more object files and combines them into a single object file and creates executable file .

**More...**

- If a macro is not defined then the preprocessor assign 0 to it by default.

- ‘**##**’ **Token-pasting Operator** is used to concatenate/merge **actual arguments** during macro expansion

Ex: - #define concat(a,b,c) a##b##c

{

printf(“%d”,concat(1,2,3));

}

ouput : 123

- printf() function in Macros will give Compilation error(May be)

- Redefinition of Preprocessor directive doesn’t give any error and it works and the most recent value of the macro would be taken

- **#pragma once** prevents a header file to get include multiple times , **#include guard** does the same job

- **‘#’ Stringizer Operator** turns the macro argument into string literal which is enclosed into double quotes

- Macros definition such as #define MACRO (a) (a\*a) , i.e with space b/w template and argu. , will be expanded to “(a) (a\*a)” without giving the actual result and any compilation error

- No matter how many times a header file is included , its content is included only once.

- The translator which performs macro calls expansion is called **Macro Pre-processor**

- The proper way of defining a macro with arguments is **#define macro(x) ((x)\*(x)) ,** otherwise unexpected result would come .

- C preprocessor is also known as **micro-preprocesor**

- A **dynamic linker** is the part of an operating system that loads and links the shared libraries .

- A Macro processor is a program that copies a stream of text from one place to another , making a systematic set of replacements as it does so .

**ARRAYS**

- An array is a collection of elements of similar datatypes

- Array elements are always stored in contiguous/adjacent memory locations

- the elements positions in array , i.e 0,1,2.. are often called subscripts and the array is called subscripted variable

- Uninitialized array variable will hold garbage value

- When a pointer is incremented , it points to an address immediately next location of its type

- Accessing array elements using pointer is always faster than using subscripts/index .

- Possible pointer operations:-

pointer +- number -> pointer

pointer - pointer -> number

pointer == pointer

- The base address of an array can also be passed by passing the name of the array .

**Two ways to pass an entire array to a function and to receive their elements**

void display1(**int \*i** , int n)

{

int j ;

for(j = 0 ; j <= n-1 ; j++)

{

printf(“%d”, **\*i**);

**i++** ;

}

}

void display2(**int i[]** , int n)

{

int j ;

for(j=0;j<=n-1;j++) printf(“%d”, **i[j]**);

}

**Flexible Array**

- **malloc()** function is used to create an array during execution/run-time .

main()

{

int max , i , \*p ;

scanf(“%d” , &max) ; // this is the size of the array

**p = (int \*)malloc(max\*sizeof(int));**

for(i=0;i<=5;i++)

{

p[i] = i\*i ;

printf(“%d ”,p[i]);

}

}

- **malloc()** takes the number of bytes to allocate in memory as argument i.e **max\*sizeof(int)**

**-** After allocating the location it returns the base address of the allocated chunk as void pointer , which is typecasted to integer pointer by **(int \*)** before malloc() .

- Now **p** would be holding an address and it could be used as a normal array through **p[i]** expression

**2-D Array**

- 2-D array is the collection of several 1-D arrays and termed as a **matrix**

**-** while initializing , it is necessary to mension 2nd dimension(column) , whereas the first dimension is optional

**int s[3][3] = {**

**{1 , 2 , 3},**

**{4 , 5 , 6},**

**{7 , 8 , 9 }**

**}**

- In memory the array elements are stored in one continuous chain no matter , it is 1-D or 2-D , as the 2-D arrangement of elements are conceptually true only...

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 65508 | 65512 | 65514 | 65518 | 65522 | 65526 | 65530 | 65534 | 65538 |

- In 2-d array s[3][3] = {

{1 , 2 , 3},

{4 , 5 , 6},

{7 , 8 , 9 }

}

=> s[i] gives the address of the ith 1-D array as well as base address of that 1-D array

=> **s = \*s = &s[0][0] will give address and \*\*s will give element s[0][0]**

=> \*(s + 1 ) = s[1] , address of 1st 1-D array & s[1][0] and (\*s +1) = address of s[0][1]

- Expressions for for element 8 ,

**s[2][1] = \*(s[2] + 1) = \*(\*(s+2)+1)**

=> \*(\*(s+1)) = \*(s[1]) = 4 and \*(\*s + 1) = \*(&s[0][1]) = 2

But , if address of base address is assigned to a pointer i.e , \*p = &s , then

=> p != \*p , or , p = address and \*p = 1 , the 0th element

moreover , **\*(p + 1) = 2 , \*(p + 2) = 3 , \*(p + 3) = 4** ......and so on [unlike \*(s + 1) ] .

- Declaration of Pointer to array variable = **int (\*p) [3] or \*p**

- Declaration of Array of pointer variable = **int \*p[3]**

- Array can be passed to a function in such manner , **fun(int a[][4])** or **fun(int (\*p)[4])**

**3-D Array**

- It is the collection of 2-D arrays

- In a 3-d array , first index refers No. of 2-d arrays , second and third refers no of rows and columns of the 2-d arrays

**int a[2][3][2] =**

**{**

**{**

**{1 , 2},**

**{3 , 4},**

**{5 , 6}**

**},**

**{**

**{1 , 2},**

**{3 , 4},**

**{5 , 6}**

**},**

**{**

**{1 , 2},**

**{3 , 4},**

**{5 , 6}**

**}**

**}**

- Here , a[0][0] is the address of 0th 2-d array

-> **a = \*a = \*\*a will give address and \*\*\*a will give element a[0[0][0]**

- Expressions for elements in 2nd 2-D array , 1st row and 3rd column

**a[2][1][3] = \*(a[2][1] + 3) = \*(\*(a[2]+1) + 3) = \*(\*(\*(a+2)+1)+3)**