**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(applications)**

* 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 Or Advantages :-**

* 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**)

// TO COVER UP AT LAST MOMENT

\*\*C vs CPP vs JAVA\*\*

\*\*Advantages and Disadvantages\*\*

**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 . (In JAVA this is called **Ternary operator**)

- 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/Activation record .**

**-** 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** to make the problem smaller.

ex:- Sum(n) = n + n-1 + n-2 + n-3 ...... + 0 , we can also write this expression as following ,

Sum(n) = n + Sum(n-1) -------(a), and

Sum(n-1) = n-1 + Sum(n-2) , so this is how the problem of sum is getting smaller ,as we don’t need to add all elements from 0 to n .

also , fibonnaci(nth) = Fibonacci(n-1th) + Fibonacci(n-2th) -------(b)

It means , to get sum of nth number , nth fibonnaci number or other stuffs with recursion then we simply have to find the general-formula like (a) and (b) and simply call the recursive call as it is in the formula . like ,

to get sum of n numbers : return n + sum\_function(n-1) , and

to get the n fibonnaci : return fib\_fun(n-1) + fib\_fun(n-2) ,

- in this way we don’t need to think too much .

- Just get the general-formula and apply in function as it is .

- 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 .

- In a recursive function the statement written before of the recursive call will be executed alongwith the start of recursion , where as the statements written after the recursive call will get executed when the recursion starts rolling-back .

**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.

void fun(int n)

{

**if(n==0)**

**return ;**

else

printf(“Something”);

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

}

int main()

{

fun(3);

}

**Tail Recursion**

int fun(int n)

{

if(n==0)

return ;

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

**printf(“Something”);**

}

int main()

{

fun(3);

}

**Non-Tail Recursion**

Nothing to get evaluated after recursive call

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

**Base Case**

**\*\*Must go through the Recursion pdf provided in the DSA-GoogleClassroom**

**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[0][0] , 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] then**

**int pint = (int \*)p [**Or we can directly use \*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)**

**More...**

- Array name in C is implemented be a constant pointer . Increment and decrement is not possible on constant types . Hence , **if there is an array int a[10] , then a++ will give lvalue error**

- Array name gives the base address but if we use ‘**address of’** operator before the array name then it gives the **address of the whole array** .

i.e for int arr[6] , arr gives the base address and &arr gives the address of the whole array

hence **, if arr = 2000 then arr + 1 = 2004 but &arr+1 = 2024**

- In C , **array Parameters(passed in function**) are treated as pointers , hence by passing the array name into **sizeof()** function , it gives the size of pointer only rather the size of whole array, i.e ,

**sizeof(arr)/sizeof(arr[0])** would turn into **sizeof(int \*)/sizeof(int)** that equals to **4/4 = 1**

Hence , don’t pass array parameter in sizeof() to get the array size

**Strings**

- A string is a 1-D array of characters terminated by a **‘\0’ (null / String Terminator).** e.g **char name[] = {‘R’,’A’,’M’,’\0’}**

- ‘\0’ and 0 are different , ASCII value of ‘\0’ is 0 whereas ASCII value of ‘0’ is 48

- ‘\0’ helps the function to know that where the string actually ends , and we can print or traverse the string elements by just putting condition as **while(name[i] != ‘\0’)** and hence one has nothing to do with the size of the string for the same job .

- The ‘\0’ also occupy 1 byte in the string , hence , the size of string “ABC” = 3 + 1 = 4

- scanf() can’t receive multi-words string , **gets()** and **puts()** function is used for this purpose

- If we want to receive multi-word string with scanf , the we need to go like this , **scanf(“%[^\n]s”,name) ,** in which [^\n] indicates that , scanf() will keep receiving characters into name until **\n**(new line due hitting enter) is encountered

**char name[] = {‘R’,’A’,’H’,’U’,’L’,’\0’};**

**char name[] = “RAHUL”;**

**char \*name = “RAHUL”;**

**-** We can directly assign a string to a **char pointer** as shown above unlike int pointers , hence a char pointer holds address as well as the string at that address

Let , char str[] = “Quest” ; and char \*p = “Quick” ;

- Here **, str is a constant pointer** , it can’t be changed and **“Quest” is not a constant string**

- Also **, p is a pointer to a constant string** , i.e. , “Quick” can’t be changed but **p can be changed**

str++ ; [error , constant pointer can’t be changed]

\*str = ‘Z’; [works , because string is not constant]

p++; [works , because pointer is not constant]

\*p = ‘M’ [error , because string is constant]

**-** [This is **Address level** operation]**We cannot assign a string to another whereas , we can assign a char pointer to another** , e.i ,

char []str1 = “hello” and char str2[10];

**str1 = str2 [gives error , b/c here constant pointer is trying to change] , but**

char \*p = “Heloo” and char \*o ;

**o = p [works fine , b/c here the pointeris not constant , hence it will change]**

**-** [This is **Value level** operation ]**We cannot manipulate the characters of a string assigned to a char pointer , but it is possible with char array .**

char \*str = “Hello” , **str[0] = ‘M’ [won’t work , b/c here constant string(value) is trying to change]** , but

char str[] = “Hello” , **str[0] = ‘M’ [works fine , b/c here the string(value) is not constant]**

- **Once , the string is defined , it can’t be initialized/updated with another string , but its not true with char pointers** , i.e ,

name = “SAM” **[gives error]** , but

p = “Bye” **[works fine]**

**- Clearly , we cannot do the character level manipulation with char pointer , b/c the string would be constant , however , we can directly change the whole string assigned the pointer .**

**Standard Library String Functions**

**strlen()** - It counts the number of characters in the string

**strcpy()** - copies the contents of one string to another

**strcat()** - concatenates the source string at the end of the target string

**strcmp()**

- compares the two strings , whether they are same or not.

- if the two strings are identical , it returns 0 , else it returns the numeric difference b/w the ASCII values of the first non-matching pair of characters

ex:- **strcmp(“Jerry”, “Ferry”)** , will return **ASCII(‘J’) - ASCII(‘F’) i.e 4**

**strcmp(“Jerry”, “Ferry King”)** , will return **ASCII(‘\0’) - ASCII(BLANK) i.e 0 - 32 = -32**

**- By calling this function , we usually wanna know that whether or not the first string is alphabetically before the second string . If it is , negative value is returned else positive value is returned**

- **‘A’ is character but “A” is a string ending with ‘\0’**

**2-D Array of characters**

- We can do implementations like traversing and taking input of characters in 2-D string , like 2-D integers arrays , but in 2-D char arrays/2-D Strings some steps and implementations are different and shorter than 2-D arrays

**2 ways of declaration and initialization of multiple strings**.

=> **By using 2-D character array**:-

char str[6][20] = {

“Ramesh”,

“Suresh”,

“Tiwari”,

“Pradeep”,

“Zaheer”,

“Tarun”

}

for printing each string , we do : **for(i=0;i<=5;i++)**

**printf(“%s”,&str[i][0]);**

for taking input : **for(i=0;i<=5;i++)**

**scanf(“%s”, &str[i][0]);**

- no need to go by each character of every 1-D string and using double loop , for printing or taking input , unlike 2-D Array . **Single loop is required**

**- This way leads to wastage of memory , as the length of the strings is already fixed in the second subscript , no matter this much length is used or not .**

=> **By using array of pointers to String**

char \*str[] = {

“Ramesh”,

“Suresh”,

“Radhika”,

“Alok”,

“Kumar”}

For printing names : **for(i=0;i<5;i++)**

**printf(“%s” , str[i]);**

**- While using array of pointers to string , we cannot receive strings from keyboard as input unlike 2-D char array . i.e ,**

if we declare **, char \*str[6] ;** then , **scanf(“%s”,str[1]);** is not gonna work **,**

**- because , when we declare such array of pointers , it contains garbage values(instead real addresses) and these garbage values couldn’t be used as address and send to scanf to receive strings in them .**

=> pros:

- Easy to process and manipulation in the strings

- Efficient memory usage

=> Cons :

- Cannot change strings .

- Though , the relative positions of strings in the array can be changed

- Cannot receive strings from keyboard easily [**Can be done by allocating space for each strings using malloc() and then assign the addresses returned by malloc() to the array of pointer to strings elements**]

**Structures**

- It is a special data-type which lets us group dissimilar data-types together

**struct book**

**{**

**char name;**

**float price;**

**int pages;**

**}**

**while((dh=getchar())!=’\n’);** - We use to flush out the keyboard buffer

Following function must be define anywhere in order to prevent “Floating point Formats Not Linked” error **void linkfloat()**

{

float a = 0 , \*b ;

b = &a ;

a=\*b ;

}

- Declaration of a structure does not reserve any space in the memory , as it only defines the form of the structure

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

- Array of structures : struct book b[5];

Declaration of structure type and definition of structure variablescan be achieved in one statement

ex:- struct book

{

char name ; float price ; int pages;

}b1 , b2 , b3 ;

- Like primary variables and arrays , structure variables can also be initialized where they are declared

ex:- struct book

{

char name ; float price ; int pages;

};

struct book b1 = {“Maths”,125.5,520 ;

struct book b2 = {“Hindi”,123,400};

struct book b3 = {“English”,130,500};

Copying one structure variable to another in piece-meal as well as at one go

structure employee

{ char name[] ; int age ; float salary};

struct employee e1 = {“ram”,24,200} , e2 , e3;

copying elements one by one

**strcpy(e2.name , e1.name) ;** as e2.name = e1.name would be wrong

e2.age = e1.age; e2.salary = e1.salary ;

at one go , **e3 = e2;**

- Copying of all structure elements at one go has been possible because the structure elements are stored in contiguous memory locations

- To a function , we can either pass individual structure elements or the entire structure variable at one shot or the base address of the structure . like ,

struct var{

char name[10];

int price;

int pages;

};

struct var b ;

**display1(b.name , b.price , b.pages);**

**display2(b);**

**display3(&b);**

- if **var** stores a structure variable , then to access the structure elements we use dot(.) operator , like

**var.name** , **var.price** ,var**b.pages** , as , dot(.) operator requires structure variable on its left side.

- if **var** is a structure pointer or pointer to structure , we access the elements like , **var->name** , **var->price** , **var->pages ,** as there must be a pointer to structure at left of arrow(->) operator

**Packing structure elements (Let us C)**

- a 32-bit micro-processor is able to fetch data that is present at an address , which is multiple of 4 much faster than the data present at other address.

- Hence , regardless of datatypes a 32-bit compiler aligns every element of a structure at an address that is mulptiple of four.

- hence , any element of the structure regardless of data-types hold 4 bytes of memory and if all the bytes are are not occupied by any element then the remaining location is remained vacant and termed as a hole.

- To fix this thing , we use **#pragma pack** directive .Or ,we can say **#pragma pack** is used to pack the **padded** structure elements.

**\*Head to letusC for more .....**

**Structure Padding(Neso Accademy)**

struct abc{

char a ; // 1 byte

char b ; // 1 byte

int c ; // 4 byte

}var;

expected size of structure = 6 bytes , but

originally it is 8 , because of the padding .

- Processor doesn’t read 1 byte at a time from memory

- It reads 1 word at a time

- for 32-bit processor 1 word = 4 bytes and for 64-bit processor 1word = 8 bytes

- In above structure two CPU-cycles are required to access value of variable c , and it causes wastage of CPU-cycle

- To prevent this wastage of cycles , we use concept of **padding**

**-** with concept of padding , the c variable get accessed in single cycle only, although **, it creates some empty rooms**

expected alignment (no padding)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | c | c | c | c |  |  |

Size of structure , total bytes = 6

actual alignment(padding)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| a | b | empty | empty | c | c | c | c |

Size of structure , total bytes = 8

**Packing structure elements(Neso Accademy)**

- Because of structure padding , the **size of the structure becomes more than the the size of the actual structure** . because it creates some **empty rooms** in order to **save the processors cycles** and **some memory get wasted**

- we can avoid wastage of memory by using **structure packing** which is done simply by adding **#pragma pack(1)** in our code although there will be wastage of cycles

- #pragma is special purpose directive used **to turn on or off certain features**

- here we are switching on the feature **pack()** by #pragma , to prevent the padding

Hence , now the structure size = 6 and not 8

**\*\*Head to Neso Academy for reference.......**

**\*\*Head to Let us C for Structure’s Applications part**

**Console Input/Output**

- I/O facilities are OS dependent

- There are numerous I/O library functions available

Types of I/O functions

a. **Console I/O functions** : These are the functions defined to receive input from keyboard and write output to screen

b. **File I/O functions** : Functions that are defined to perform to I/O operations on a disk

- keyboard and screen are together called console

- Types of Console I/O functions

1 . Formatted I/O functions

2 . Unformatted I/O functions

- The basic difference between the two is , Formatted functions allow the input that is read from the keyboard or the output displayed on the screen to be formatted as per our requirements

- means with formatted functions you can fix the format of the output to be displayed on the screen , like , **spaces between** to values , **number of decimal** places , etc

- **scanf() for input** and **printf() for output** are the two formatted I/O functions used for char , int , float and string datatypes

**Formatted Console I/O functions**