Note: In VSCode BreakPoints in red dot is meant for debugging the codes in order to pause the process

History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

Dennis Ritchie is known as the founder of the c language.

It was developed to overcome the problems of previous languages such as B, BCPL, etc.

Initially, C language was developed to be used in UNIX operating system. It inherits many features of previous languages such as B and BCPL.

C language Tutorial with programming approach for beginners and professionals, helps you to understand the C language tutorial easily. Our C tutorial explains each topic with programs.

The C Language is developed by Dennis Ritchie for creating system applications that directly interact with the hardware devices such as drivers, kernels, etc.

C programming is considered as the base for other programming languages, that is why it is known as mother language.

C language is widely used for developing system software, application software, games and embedded systems.

It can be defined by the following ways:

1. Mother language

2. System programming language

3. Procedure-oriented programming language

4. Structured programming language

5. Mid-level programming language

To run C program in vscode:

open new terminal

gcc main.c

.\a.exe (Upon pressing a+tab) // in Windows OS

.\a.out // in Linux OS

Or simply type in terminal if you want to give a manual name to your executable program:

Ex:

gcc .\main.c -o cprogram

.\a.exe (Upon pressing a+tab) // in Windows OS

.\a.out // in Linux OS

.\cprogram.exe

C program behind the compilation process, it removes comments, expands macros and any hash include contents are resolves.

The basic structure of the program in C has the following:

Macros in C:

A macro is a symbol that is recognised by the preprocessor and replaced by the macro body

- Structure of simple macros:

#define identifier replacement\_list

- Examples:

#define BUFFERSZ 1024

#define WORDLEN 64

Types of macros in C:

1. Predefined macros

2. Object-like macro

3. Funtion-like macro

Ex: #define PI 3.14

Here: #define - preprocessor directive

PI - macro name

3.14 - macro value

gcc -Wall -save-temps main.c -o cx3eno

1. Preprocessing - .i file (Comments are removed, macros are expanded).

2. Compilation - .s file (Assembly level instructions are converted).

3. Assembly - .o file (Machine level instructions are binary - 0s and 1s).

4. Linking - .exe file (Executable files are then converted to .exe files - dynamic linking [dll] & static linking).

5. Loading - (Program is loaded in RAM - Random Access Memory or Memory and runs from there).

C Language is made up of individual tokens as it is the basic building blocks and it has the following tokens. They are:

ex:

#include <stdio.h> [#include is a preproccesor statement, no need to use semi-colons in preproccesor statements]

int main() [int main() is a main function. The execution of the program starts from here.]

{

int a, b; [identify a and b as integer]

printf [printf is used to print whatever is input by the programmer will show on console.]

( [paranthesis]

"Hello World!\n" ["Hello World" is a string literal which will display on console. \n - newline character]

); [paranthesis with end statement - ;]

scanf("%d", &a); [scanf is a function which is used for taking input values from the user. Whatever user inputs values from the keyboard will return it to a. %d acts as a placeholder for an integer arugment in input and output. &a - address of a.]

It also takes address of that operator. Any integer variable does not directly take variable from scanf. Instead it takes address from operator and then inputs the value into it from the keyboard.

return [one of the reserved keywords]

0; [constant - 0. program has run successfully which will tell the main() function]

[printf ( ) are white spaces]

Note: one white-space character is a must in c language.

}

Note: argc, argv[] is a command line statements.

Indentation in C:

Indentations in C are very useful in understanding the structure of a program. Inside each set of braces, lines are indented by the same amount. It is easy to see visually which statements belong to each function or control structure.

1. Keyword - reserved words. 32 reserved keywords. No constants, no identifiers nor any variables can be made. They are:

auto break case char

const continue default do

double else enum extern

float for goto if

int long register return

short signed sizeof static

struct switch typedef union

unsigned void volatile while

2. Identifier - is the name of any variable or function through which we can identify them. Does not allow $, %, @ as identifiers.

C is a case-sentive language meaning Cx3eno and cx3eno are treated differently. ; - semicolon is used for terminating a statement.

C identifiers represent the name in the C program, for example, variables, functions, arrays, structures, unions, labels, etc.

An identifier can be composed of letters such as uppercase, lowercase letters, underscore, digits, but the starting letter should be either an alphabet or an underscore.

3. Constants - return 0. Now 0 here is a constant.

4. String Literals - "Hello World!\n" here is a string literal. It has it's own strings.

5. Symbols - scanf("%d", &a); - %d is a format specifier which act as a placeholder for an integer argument in a formatted input and output statement &a here is a symbol.

Variables - A name given to a memory location.

Declared by writing [type variable\_name; - int a, b, c, d;]

Initialized and declared by [type variable\_name = value; int a = 4;]

Variable is nothing but a name in given memory location.

Rules:

1. Can contain alphabets, digits, underscore (\_).

2. A variable name can start with an alphabet and underscore only.

3. Can't start with a digit.

4. No whitespaces and reserved keywords is allowed.

5. Valid variable names: int harry, float harry123, char \_harry34;

6. Invalid variable names: int harry shubham, $harry, int 77harry, char long;

Data Types - The data type is a collection of data with values having fixed values, meaning as well as its characteristics.

1. Basic Data Type: int, char, float, double

2. Derived Data Type: array, pointer, structure, union

3. Enumeration Data Type: enum

4. Void Data Type: void

5. User-defined Data Type: structure, union, enum and typedef

Different data types also have different ranges up to which they can store numbers.

These ranges may vary from compiler to compiler.

Below is a list of ranges along with the memory requirement and format specifiers on the 32-bit GCC compiler.

Data Type Size (bytes) Range Format Specifier

char 1 -128 to 127 %c

signed char 1 -128 to 127 %c

unsigned char 1 0 to 255 %c

short 2 -32,768 to 32,767 %d

signed short 2 -32,768 to 32,767 %hi

unsigned short 2 0 to 65,535 %hu

int 2 or 4 -32,768 to 32,767 %d

signed int 2 or 4 -32,768 to 32,767 %d

unsigned int 2 or 4 0 to 65,535 %d

short int 2 -32,768 to 32,767 %hd

signed short int 2 -32,768 to 32,767 %d

unsigned short int 2 0 to 65,535 %u

long int 4 -2,147,483,648 to 2,147,483,647 %ld

signed long int 4 -2,147,483,648 to 2,147,483,647 %ld

unsigned long int 4 0 to 4,294,967,295 %lu

float 4 1.2E-38 to 3.4E+38 %f

double 8 1.7E-308 to 1.7E+308 %lf

long long int 8 -(2^63) to (2^63)-1 %lld

unsigned long long int 8 0 to 18,446,744,073,709,551,615 %llu

long double 10 3.4E-4932 to 1.1E+4932 %Lf

1 Bit = 0.125 bytes

1 byte = 8 bits

1 KB(Kilobyte) = 1024 bytes

1 MB(1 Megabyte) = 1024 KB

1 GB(Gigabyte) = 1024 MB

1 TB(Terabyte) = 1024 GB

1 PT(Petabyte) = 1024 TB

Note: The long, short, signed and unsigned are datatype modifier that can be used with some primitive data types to change the size or length of the datatype.

The range and size of these data types depend from architecture to architecture.

Format Specifiers Type of Output

%d or %i A decimal integer or signed integer

%c Character

%d Signed integer

%e or %E Scientific notation of floats

%f Float values

%g or %G Similar as %e or %E

%hi Signed integer (short)

%hu Unsigned Integer (short)

%i Unsigned integer

%l or %ld or %li Long

%lf Double

%Lf Long double

%lu Unsigned int or unsigned long

%lli or %lld Long long

%llu Unsigned long long

%o Octal representation

%p Pointer

%s String

%u Unsigned int

%x or %X Hexadecimal representation

%n Prints nothing

%% Prints % character

Note:

Ex:

scanf("%f"); o/p-> 197.850006

scanf("%.f"); o/p-> 197

scanf("%.1f"); o/p-> 197.3

scanf("%.2f"); o/p-> 197.65

%.2f tells printf to print a floating point with 2 decimal places.

Basic Operators and Functions:

Operators -

+, -, \*, /

Functions -

int a;

printf("%d", 3+7); <- used for printing values

scanf("%d", &a); <- used for taking inputs from user

sizeof() <- The sizeof operator gives the amount of storage, in bytes, required to store an object of the type of the operand. This operator allows you to avoid specifying machine-dependent data sizes in your programs.

Operators -

An operator is a symbol used to perform operations on a given programming language.

In this tutorial series, we will look into operators used in the C programming language. Ex: 3 + 6 = 9 ([+] here is an operator [3 and 6 are two operands]).

Types of Operators in C -

1. Arithmetic Operators - An arithmetic operator performs mathematical operations such as addition, subtraction, multiplication, division etc on numerical values.

+ Addition

- Subtraction

\* Multiplication

/ Division

% Modulus

2. Relational Operators (Used in statements) - A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

== Is equal to (Used in If statements)

!= Is not equal to

> Greater than

< Less than

>= Greater than or equal to

<= Less than or equal to

3. Logical Operators - A logical operator is a symbol or word used to connect two or more expressions such that the value of the compound expression produced depends only on that of the original expressions and on the meaning of the operator.

Common logical operators include AND, OR, and NOT.

&& Logical AND. If both the operands are non-zero, then the condition is true.

Ex: (A && B) is false. 1 && 2 becomes true.

|| Logical OR. If any of these two operands is non-zero, then the condition becomes true.

Ex: (A || B) is true. 1 || 0 becomes true.

! Logical NOT. It is Used to reverse the logical state of its operand. If condition is true, then Logical NOT will make it false.

Ex: !(A && B) is true.

4. Bitwise Operators - Converts numbers into bits and then works on it.

a b a&b a|b a^b

0 0 0 0 0

0 1 0 1 1

1 1 1 1 0

1 0 0 1 1

Binary Equivalents:

0 => 00

1 => 01

2 => 10

3 => 11

Ex: (2) & (3) => (10) & (11) => 10

Exclusive OR (XOR) = ^ This operator is used to identify any one value as true or false. If both values are true or false. It will return as 0.

Ex: 0 0 => 0

0 1 => 1

1 0 => 1

1 1 => 0

~ is the binary one's complement operator

<< is the binary left shift operator

>> is binary right shift operator

5. Assignment Operators - An assignment operation assigns the value of the right-hand operand to the storage location named by the left-hand operand.

Therefore, the left-hand operand of an assignment operation must be a modifiable l-value.

After the assignment, an assignment expression has the value of the left operand but isn't an l-value.

= Simple assignment operator. Assigns values from right side to left side operand. Ex: int a=7;

+= Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand. Ex: a+=1; o/p => 8

-= Subtract AND assignment operator. It subtracts the right operand from left operand and the result is assigned to the left operand. Ex: a-=1; o/p => 6

\*= Multiply AND assignment operator. It multiplies the right operand with the left operand and the result is assigned to the left operand.

/= Divide AND assignment operator. It divides the left operand with the right operand and the result is assigned to the left operand.

Note: True = 1; False = 0;

Miscellaneous Operators -

Sizeof() Returns the size of a variable. Ex: sizeof(a), where a is an integer, will return int's size on that architecture. Returns value depending on the size of the architecture.

& Returns the address of a variable. Ex: &a; returns teh actual address of the variable.

\* Pointer to a variable. Ex: \*a;

?: Conditional Expression. Ex: If condition is true ? then value X: otherwise value Y.

Operator Precedence in C -

Category Operator Associativity

Postfix () [] -> . ++ -- Left to Right

Unary + - ! ~ ++ -- (type)\* & sizeof Right to Left

Multiplicative \* / % Left to Right

Additive + - Left to Right

Shift << >> Left to Right

Relational < <= > >= Left to Right

Equality == != Left to Right

Bitwise AND & Left to Right

Bitwise XOR ^ Left to Right

Bitwise OR | Left to Right

Logical AND && Left to Right

Logical OR || Left to Right

Conditional ?: Right to Left

Assignment = += -=\*= /= %=>>= <<= &= ^= |= Right to Left

Comma , Left to right

Ex: BODMAS = 1+2x3 = 1 + 6 = 7

In programming languages, the associativity is worked by the precedence of the operators.

Ex: 1 + 2 x 3 = 3 - 3 (according to the precedence) = 0

Note: In programming languages, the associativity of an operator is a property that determines how operators of the same precedence

are grouped in the absence of paranthesis.

Operators In C: C Tutorial In Hindi #7

Today we are going to learn about operators. I will teach you guys the theory related to operators as well as showing you the code as examples. So we will be using VS Code to write a few lines of codes for better understanding of the topic. Let’s start with the definition:

“Special symbols that are used to perform actions or operations are known as operators.”

For example, the symbol plus (+) is used to perform addition so it is an operator.

We will discuss all sorts of operator here. Let’s start with the simpler one’s i.e. Arithmetic.

Arithmetic operators:

Arithmetic operators are used to perform mathematical operations such as addition, subtraction etc. Few of the simple arithmetic operators are :

Operator

Description

+

Addition

−

Subtraction

\*

Multiplication

/

Division

%

Modulus

We all know their purpose and how they are used in simple mathematics. Their purpose and functionality are the same, let’s see their implementation in C.

int a = 2;

int b = 3;

printf("a + b = %d\n", a+b);

The output will be:

a + b = 5

Relational Operators:

Relational operators are used for the comparison between two or more numbers. Same as Java, C also has six relational operators and their return value is in Boolean i.e. either True or False (1 or 0).

Operator

Description

>

Greater than

<

Less than

>=

Greater than or equal to

<=

Less than or equal to

==

Is equal to

!=

Is not equal to

Let’s go to VS Code now:

int a = 2;

int b = 2;

printf("a == b = %d\n", a==b);

The output is 1 i.e. True.

If we change the value of a or b the value will be false or 0.

int a = 1;

int b = 2;

printf("a == b = %d\n", a==b);

The output is 0 i.e. False.

Logical Operators:

There are three logical operators i.e. AND, OR and NOT. They can be used to compare Boolean values but are mostly used to compare conditions to see whether they are satisfying or not.

AND: it returns true when both operators are true or 1.

OR: it returns true when either operator is true or 1.

Not: it is used to reverse the logical state of the operand.

Symbol

Operator

&&

AND operator

||

OR Operator

!

NOT Operator

Example:

int a = 1;

int b = 0;

printf("a or b = %d\n", a||b);

Here the output is:

a or b = 1

Let’s see what happens if both the values are zero

int a = 0;

int b = 0;

printf("a or b = %d\n", a||b);

the output is:

a or b = 0

Bitwise Operators:

To perform bit level operations, bitwise operators are used. They convert the values we provide to them in binary format and then compare them to provide us the results.

Symbols

Operators

&

Bitwise AND

|

Bitwise OR

^

Bitwise XOR

~

Bitwise complement

<<

Shift left

>>

Shift right

Assignment Operators:

Assignment operators are used to assign values. They are going to be used in each and every one of our program.

int a = 0;

int b = 1;

Equal to (=) is the assignment operator here, assigning 0 to a and 1 to b.

Operator

Description

=

Assigns values from right side operands to left side operand

+=

It adds the right operand to the left operand and assign the result to the left operand.

-=

It subtracts the right operand from the left operand and assigns the result to the left operand.

\*=

It multiplies the right operand with the left operand and assigns the result to the left operand.

/=

It divides the left operand with the right operand and assigns the result to the left operand.

Conclusion:

These are few of the important operators that you should know about before starting actual programming. There are also many other operators such as &, % or \*(pointer). I will let you know their details when working with them but the few defined above will be used frequently so knowledge about them is important. You do not have to remember them all as you can open the Tutorial any time again when required.

Code cTuts7.c as described/written in the video

#include

int main()

{

/\* code \*/

int a, b;

a = 2;

b = 3;

printf("a & b = %d\n", a&b);

printf("a - b = %d\n", a-b);

printf("a \* b = %d\n", a\*b);

printf("a / b = %d\n", a/b);

return 0;

}

C Programming Exercise 1 - Multiplication Tables: C Tutorial In Hindi #8

This is the first exercise of C programming tutorials. This C programming series contains many exercises that will help you to become a great problem solver. So, your first task is to “Print Multiplication Table in C”. All of the concept we have studied till now will use in this task, so by solving this exercise you can examine yourself that how much effort you need to put in learning C language.

Instructions:-

Take an input (in integer form) from the user and print its multiplication table on the screen. Following is the example of the output.

Input:

Enter the number you want multiplication table of: 6

Output:

Table of 6.

6\*1 = 6

6\*2 = 12

6\*3 = 18

6\*4 = 24

6\*5 = 30

6\*6 = 36

6\*7 = 42

6\*8 = 48

6\*9 = 54

6\*10 = 60

Try to solve this exercise by yourself. These programming puzzles will improve your coding skills and makes you a good problem solver.

This exercise is a part of C programming tutorials. If you have not watched my C tutorial, then click on the link and start learning!

If you like my work, then check out my other courses and stay up to date with codewithharry.

Code as described/written in the video

# include <stdio.h>

/\*

Print multiplication table of a number entered by the user in pretty form

Example:

Input

Enter the number you want multiplication table of:

6

Output:

Table of 6:

6 X 1 = 6

6 X 2 = 12

.

.

.

6 X 10 = 60

\*/

int main()

{

/\* code \*/

return 0;

}

To comment-out in VSCODE: press fn + f1

Single-line comment in C: //

Multi-line comment in C: /\* \*/

Format Specifiers, Constants, and Escape Sequences in C:

What is Format Specifier?

1. Format Specifier is a way to tell the compiler what type of data is in a variable during taking input displaying output to the user.

2. printf("This is a good boy %a.bf", var); will print var with b decimal points in a 'a' character space.

3. Lets go to our IDE and learn more about the format specifiers.

Ex: int a = 4; float b = 3.3;

Constants in C:

1. A constant is a value or variable that can't be changed in the program, for example: 15, 23, 'a', 3.4, "code with harry" etc.

2. There are two ways to define constant in C programming:

-> const keyword

-> #define preprocessor

3. Let's see both of them in the IDE.

Escape Sequence in C:

1. An escape sequence in C programming language is a sequence of characters.

2. It doesn't represent itself when used inside a string literal or character.

3. It is composed of two or more characters starting with backslash \. For example: \n represents new line.

Ex: \n = new-line [single character].

Escape Sequence Meaning

\a Alarm or Beep

\b Backspace

\f Form Feed

\n New Line

\r Carriage Return

\t Tab (Horizontal)

\v Vertical Tab

\\ Backslash

\' Single Quote

\" Double Quote

\? Question Mark

\nnn octal number

\xhh hexadecimal number

\0 Null

Comments in C:

Single-line comment in C: //

Multi-line comment in C: /\* \*/

Ex: if you're returning back to your program after some time and you need to remind yourself of the statements. You can refer to comments.

Virtual Assistant in C:

11100 lines of code.

int func1(){

return // returning the statement <- Single-line comment (ignored)

}

int main() {

/\*

(ignored)

Author:

Purpose:

Date:

\*/ <- Multi-line comments (ignored)

}

Note: Comments are ignored in C. Include author, purpose and date in your program.

Control Statements in C:

A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input.

A computer program is made up of a series of statements. The following are the types of control statements in C:

1. Decision Making Statements(Selection/Conditional/Branching Statements):

Use if to specify a block of code to be executed, if a specified condition is true. Use else to specify a block of code to be executed, if the same condition is false.

Use else if to specify a new condition to test, if the first condition is false. If the hypothesis is true and the conclusion is false, then the conditional statement is false.

Likewise, if the hypothesis is false the whole statement is false.

A. If-else Statements:

i. if statement

ii. if-else statement

iii. if-else-if statement or ladder

iv. Nested-if statement

B. Switch Statement:

iv. switch statement

2. Iterative Statements(Loop Statements):

Loop is used to execute the block of code several times according to the condition given in the loop.

It means it executes the same code multiple times so it saves code and also helps to traverse the elements of an array.

A. Entry Controlled Loop:

i. for loop

ii. while loop

B. Exit Controlled Loop:

iii. do-while loop

3. Jump Statements:

Jump Statement in C is used in C programming language to transfer control from one part of the program to another.

i. break statement

ii. continue statement

iii. goto statement

iv. return statement

C if else statements:

It is used to perform operations based on some conditions.

Types of if statements:

1. if statement

Ex:

if(3>2) {

//CODE

}

2. if-else statement

3. if-else-if ladder

4. nested if

1. Syntax for if else statement:

Ex:

if(condition) {

//code n <- if true, if statement executes, if false then exits;

}

2. Syntax if-else-statement:

Ex:

if(condition) {

//code 1 <- if true, if statement executes and exits;

}

else {

//code 2 <- if false, else statement executes and exits;

}

3. Syntax if-else-if statement:

Ex:

if(condtion 1) {

//code 1 <- condition 1 true, if statement executes and exits;

}

else if(condition 2) {

//code 2 <- condition 1 false, condition 2 true, else if statment executes and exits;

}

else {

//code n <- condition 1 false, condition 2 false, else statement executes and exits;

}

4. Syntax switch case statement:

Ex:

int a = 2;

switch(a) { // receive expression as an input

case 1: // unmatched

printf("value is 0); <- statement 1 matched

break; // expression executes the case 1 when statement 1 is matched, breaks, exits outside of switch;

case 2: //matched

printf("value is 2); <- if statement 2 matched, statement 1 unmatched

break; // expression executes the case 2 when statement 1 is unmatched, breaks, exits outside of switch;

.

.

.

case n:

.

.

default:

printf("nothing matched"); <- statement n

break;

// regardless of statements(whatever happens), this executes the default, exits outside of switch;

}

//code n <- if the case is matched, it will exit out of switch case statement;

Rules for Switch Statement:

1. Switch epxression - (must be an) int or char.

2. Case value must be an int or char.

3. Case must come inside switch.

4. Break is not a must.

Ex:

Correct: | Incorrect:

int x; | float x;

switch(x) { | switch(x)

=//cases+code | {

} | }

Note: In switch statement expressions are used instead of conditions. Break statements are only for stopping the switch case;

statement upon meeting the value. Break statements are optional to use. Default will print regardless of given value.

Loops in C:

Ex: print 1 to 10000

Printf

Advantages of using Loops:

1> Code Reusability

2> Saves time

3> Traversing -> Hold for now

index = 0;

loop start

printf();

increment index;

loop end

until condition fails to satisfy

Basic Syntax:

int i = 0;

Loop starts -> check the condition

i < 10;

if condition true = execute the loop

i = i + 1;

if conditon false = exit the loop

because i = 10 and 10 < 10 is false and loop will end at 9 < 10

Types of loops:

1> do while loop

2> while loop

3> for loop

do while loop:

Basic syntax:

do {

// code to be executed

} while (condition)

ex:

int i = 0;

do {

i = i + 1;

printf("%d", i);

} while(i<10);

do while loop executes at least once

Output:

i = 1

i = 2

i = 3

.

.

.

i = 9

i = 10 and 10 < 10 it will exit the loop

While loop in C:

while(condition) {

// code to be executed

}

int i=0;

while(i<30) {

printf("%d",i);

i=i+1;

}

Output:

if with \n:

0

1

2

.

.

.

.

.

.

29

if without \n:

0123....29

Why use For Loop?:

1. The for loop is usde to iterate the statements or a part of the program several times.

2. It is used to traverse the data structures like the arrays and linked lists.

3. It has a little different syntax than while and do while loops.

Syntax of for loops:

the syntax of for loop in c language is given below:

for(Expression1; Expression 2; Expression 3)

{

// code to run

}

Ex:

int i;

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

printf("%d",i);

}

Here, i++; = i=i+1; they are the same.

now, the for loop will check the first expression, if it is true then it will check the second expression

and then it will print 0 and then it will revert back to i++ in ex. 3 to increment it and then it will check the ex. 2 which is true,

then it will print 1 and then it will revert back to ex. 3 and so on until it fails to staisfy the condition which is false.

Output:

0

1

2

3

4

Properties of Expression 1:

i. The expression represents the initialization of the loop variable.

ii. We can initialize more than one variable in Expression 1.

iii. Expression 1 is optional.

ii. Ex:

int i = 0, j = 0;

for (; i < 5; i++)

{

printf("%d %d\n", i, j);

}

Output:

0 0

1 0

2 0

3 0

4 0

iii. Ex:

int i, j;

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

{

printf("%d %d\n", i, j);

}

Output:

0 0

1 0

2 0

3 0

4 0

Properties of Expression 2:

i. It is a conditional expression. It checks for a specific condition to be satisfied. If it is not, the loop is terminated.

ii. It can have more than one condition. However, the loop will iterate until the last condition becomes false. Other conditions will be treated as statements.

iii. It is optional. -> [Hold for now. Use break instead.]

iv. Expression 2 can perform the task of expression 1 and expression 3. That is, we can initialize the variable as well as update

the loop variable in expression 2 itself.

v. We can pass zero or non-zero value in expression 2. However, in C, any non-zero value is true, and zero is false by default.

i. & ii. Ex:

for (; i < 5, j < 3; i++)

{

printf("%d %d\n", i, j);

j++;

}

Output:

j < 3

0 0

1 1

2 2

Ex:

for (; j < 3, i < 5; i++)

{

printf("%d %d\n", i, j);

j++;

}

Output:

i < 5

0 0

1 1

2 2

3 3

4 4

Ex: 1-n = true, 0 = false

Properties of Expression 3:

i. Expression 3 is used to update the loop variable.

ii. We can update more than one variable at the same time.

iii. Expression 3 is optional.

i.

Ex:

int i, j = 0;

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

{

printf("%d %d\n", i, j);

}

Ouput:

0 0

1 0

2 0

3 0

4 0

ii.

Ex:

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

{

printf("%d %d\n", i, j);

}

Output:

0 0

1 1

2 2

3 3

4 4

iii.

Ex:

for (i = 0; i < 5;)

{

printf("%d %d\n", i, j);

i++;

j++;

}

Output:

0 0

1 1

2 2

3 3

4 4

Note: If there is a single statement in for loop then there is no need for braces.

Break and Continue Statement in C:

Break Statement:

i. Used to bring the program control out of the Loop

ii. The break statement is used inside loops or switch statement

iii. Break statement can be used with

-> loops

-> switch case expressions

while(cond) {

//user enters his name & keeps playing until game over

}

if(name=="Shubham"){

break;

}

Note: in nested loops, you will only exit a single loop and not entirely out of the whole loop.

Ex:

for {

//Code

for {

//Code

break; <- Will only exit out of this for loop

}

}

Continue Statement:

i. Used to bring the program control to the next iteration of the loop

ii. The continue statement skips some code inside the loop and continues with the next iteration

iii. It is mainly used for a condition so that we can skip some lines of code for a particular condition

Ex:

while(cond n) {

if(name=="harry") {

continue;

}

. // code skips

. // code skips

. // code skips

. // code skips

//1000 lines

5 secs to execute.

}

Note: in continue statement, the iteration takes place while skipping the code inside the loop. if in continue statement if the code does not

meet the conditions of continue statement then it will break & skip the continue statement.

Goto Statement in C:

i. Also called jump statement in C

ii. Used to transfer program control to a predefined label

iii. Its use is avoided since it causes confusion for the fellow programmers in understanding the code

iv. Goto statement is preferable when we need to break multiple loops using a single statement at the same time

v. Lets go to the IDE and understand more

syntax:

label:

// codes

goto label: <- goes to a certain label in the code to break looping statement

to avoid:

label:

//code

.

.

goto end;

goto label;

.

.

.

end:

// code

.

.

typecasting & typeconversion in C:

// Typecasting syntax

// (type) value; <- (int) value [(int) b / (float) 75/6]

/\* What is typecasting?

Typecasting - refers to converting a data type to a different data type

Typeconversion - also known as type casting or data conversion refers to converting of calculation between int-int which gives int value

and between int-float gives float value and between float-float gives float value and

if the user wants the float value between int-int then convert any int to float \*/

Ex:

int main(int argc, char const \*argv[])

{

int a = 4;

float b = 54/5;

printf("The value of a is %f\n", (int) b);

return 0;

}

Functions in C:

What is a function?

A.k.a procedure or subroutine

i. Functions are used to divide a large c program into smaller pieces

ii. A function can be called multiple times to provide reusability and modularity to the C program

iii. also called procedure or subroutine

Ex: int main() {

-> Aata plate mein daalo ();

-> Aata gundo ();

-> Gola banao ();

.

.

.

return 0;

}

-> Aata plate mein daalo() {

}

Function: Syntax

i. the basic syntax of a C funtion is written as follows:

return\_type function\_name(data\_type parameter1, data\_type parameter2, ...) {

// code to be executed

}

return\_type = int

int printstar() {

print("\*");

return 0;

}

parameter = int number

Advantages of C Functions:

i. We can avoid rewriting same logic through functions

ii. We can divide work among programmers using functions

iii. We can easily debug a program using functions

void print tar () {

In main function we printed:

\*\*\*

\*\*

\*

}

Declaration, Definition and Call:

i. A function is declared to tell a compiler about its existence

ii. A function is defined to get some task done

iii. A function is called in order to be used

Ex:

function declaration:

only need to declare a function

int printstar ();

main () {

// function call, we don't use function use

a=printstar();

}

function definition:

only need to define a function

int printstar() {

// actual implementation

printf() ----

----

--

return 0;

}

Types of functions:

C functions:

i. Library functions

ii. User-defined funtions

-> Library functions - funtions included in c header files

-> User-defined funtions - functions created by c programmer to reduce complexity of a program

A user-defined function is a type of function in C language that is defined by the user himself to perform some specific task. It provides code reusability and modularity to our program.

Ex:

Library:

printf("Legit");

User-defined:

inst sum(int a, int b);

Function Code Examples:

i. Without arguments and without return value // arguments = 1, 2 ; return value = 3

ii. Without arguments and with return value

iii. With arguments and without return value

iv. With arguments and with return value

Ex:

int sum(int a, int b) {

return a+b;

}

main() {

int a=1, b=2; int c;

c=sum(a,b);

}

Note: A Parameter is the symbolic name for "data" that goes into a function. There are two ways to pass parameters in C: Pass by Value, Pass by Reference.

iv. Ex:

#include <stdio.h>

int sum(int a, int b); // function prototype

int main(int argc, char const \*argv[])

{

printf("The Developer");

int a, b, c;

a = 9;

b = 7;

c = sum(a, b);

printf("The sum is %d\n", c);

return 0;

}

int sum(int a, int b)

{

return a + b;

}

iii.

Ex:

#include <stdio.h>

int sum(int a, int b); // function prototype

void printstar(int n)

{

for (int i = 0; i < n; i++)

{

printf("%c", '\*');

}

}

int main(int argc, char const \*argv[])

{

printf("The Developer");

int a, b, c;

a = 9;

b = 7;

c = sum(a, b);

printstar(7);

printf("The sum is %d\n", c);

return 0;

}

int sum(int a, int b)

{

return a + b;

}

Note:

Single quote - character

Dobule quote - string

ii.

Ex:

#include <stdio.h>

int sum(int a, int b); // function prototype

void printstar(int n)

{

for (int i = 0; i < n; i++)

{

printf("%c", '\*');

}

}

int takenumber()

{

int i;

printf("Enter a number \n");

scanf("%d", &i);

return i;

}

int main(int argc, char const \*argv[])

{

printf("Welcome Developer\n");

int a, b, c;

a = 9;

b = 7;

// c = sum(a, b);

// printstar(7);

c = takenumber();

// printf("The sum is %d\n", c);

printf("The number entered is %d \n", c);

return 0;

}

int sum(int a, int b)

{

return a + b;

}

Note:

The values that are declared within a function when the function is called are known as an argument.

These values are considered as the root of the function that needs the arguments while execution, and it is also known as Actual arguments or Actual Parameters.

Recursion in C:

What is a recursive function?

-> a recursion is a function which copies itself to solve smaller problems by dividing them

i. Recursive functions or Recursion is a process when a function calls a copy of itself to work on a smaller problem

ii. Any function which calls itself is called recursive function

iii. This makes the life of a programmer easy by dividing a given problem into easier problems

iv. A termination condition is imposed on such functions to stop them executing copies of themselves forever

v. Any problem that can be solved recursively, can also be solved iteratively

A Good Intuitive Example: Preparation of Masala Dosa

Ex:

Recursive Implementation or Reursive Approach:

let's take prepare(name) as a function

// prepare(MD) is a function

prepare(aloo), prepare(dosa), prepare(sambhar chutney) are sub functions

prepare(MD) <-> prepare(aloo) + prepare(dosa) + prepare(sambhar chutney)

prepare(aloo) = prepare(fried onions) + prepare(masalas)

+ prepare(raw aloos) // these are sub functions within another function named prepare(aloo)

Now, after the functions are called they will return it to prepare(aloo) and finally

prepare(aloo) will be returned to prepare(MD) function. And prepare(MD) function will be returned to main function.

Note: the number of return depends on the number of calls within a function; this whole process is called Recursive Implementation

Iterative Implementation or Iterative Approach:

prepare(MD) {

if(==Md){

//prepare this and that

}

return dosa;

}

Note: in iterative function, it will not call itself any number of times and neither will it call it's copies.

Why recursions?

i. any problem that can be solved iteratively, can also be solved recursively

ii. however, some problems are best suited to be solved using recursion

iii. for example, tower of hanoi, fibonacci series, factorial finding, etc.

Example 1: Factorial Calculation:

i. the case at which the function doesn't recur is called the base case

ii. the instances where the function keeps calling itself to perform a subtask, is called the reursive case. if there is no base case then that is recursive case. // Ex: 2 if there is no base case, it returns 2x1! = 1! = 1

iii. for factorial calculation the base case occurs at the parameter value of 0 and 1

n! = n x n-1 x n-2...... 1 // opposite multiplication

3! = 3 x 2 x 1 = 6 // 3 factorial

4! = 4 x 3! = 24(4 x 6)

n! = n x (n-1)! // explained above <-- 5! = 5 x 4!

0! = 1 // by definition, 0 factorial is 1 <-- base case if there comes 0 then function returns 1

1! = 1 // 1 factorial is 1 <-- base case if there comes 1 then function returns 1

Factorial(5):

5 x Factorial(4):

5 x 4 x Factorial(3):

5 x 4 x 3 x Factorial(2):

5 x 4 x 3 x 2 x Factorial(1) <-- base case(1) = 5 x 4 x 3 x 2 x 1 = 120

Iterative Strategy:

for(5>1) {

// use for loop by

}

Syntax for recursion:

void recurse()<------ <--------------

{ | recursive |

... .. ... | call |

recurse(); ------ |

... .. ... |

} |

|

int main() |

{ |

... .. ... |

recurse(); ----------------------

... .. ...

}

Note: Recursion is made for solving problems that can be broken down into smaller, repetitive problems.

Base case i s 0 and 1

Arrays in C:

What is an Array?

i. An array is a collection of data items of the same type

ii. Items are stored at contiguous memory locations

iii. It can also store the collection of derived data types, such as pointers, structures, etc

iv. A one-dimensional array is like a list

v. A two-dimensional array is like a table

vi. The C language places no limits on the number of dimensions in an array

vii. Some texts refer to one-dimensional arrays as "vectors", two-dimensional arrays as "matrices", and use the general term arrays when the number of dimensions is unspecified or unimportant

ex: 100 names

array -> 100

array of size = 100

Analogy: a box of 100 sketch pens

Now here, box is an array and sketch pens are integer, char, etc

1D Array = |1|2|3|4|5|6|7| <- list (vectors)

We can access given set of items using base address with the help of pointers

2D Array = |1|2|3|4|5|6|7|

|8|9|1|2|.|.|.| <- table (matrices)

|.|.|.|.|.|.|n|

3D Array = Cube

Why do we need Arrays?

i. Code that use arrays is somtimes more organized and readable

ii. If you were to store the marks in a test of 56 students, creating 56 variables will make program look cluttered and messy

iii. Solution to this is arrays

iv. We can create arrays of integers and store the consecutive marks corresponding to the roll number in the array

a [0] = 76

a [1] = 56

a [3] = 59

Advantage of Arrays:

i. It is used to represent multiple data items of same type by using only single name

ii. Accessing an item in a given array is very fast! -> Hold

iii. 2 Dimensional arrays makes it easy in mathematical applications as it is used to represent a matrix

Ex:

marks [56] = Array[marks];

Properties of Array:

i. Data in an array is stored in contiguous memory location // data is stored in a contiguous memory location inside a physical memory = |2|4|10|100| if int takes 2 byte in the architecture all the addresses will take 2 bytes

ii. Each element of an array is of same size

iii. Any element of the array with given index can be accessed very quickly by using its address which can be calculated using

the base address and the index

Syntax for Declaring and Initializing an Array:

i. Data\_type name[size];

ii. Data\_type name[size] = {x, y, z,...}; //size are not required in this case!

iii. data\_type name[rows][columns]; //for 2-d arrays

iv. We can also initialize the array one by one by accessing it using its index:

-> name[0]=0;

v. Lets go to VS Code and write some code

For ex:

int marks[200]

marks[0] = 100;

marks[1] = 96;

.

.

.

marks[199] = 101;

data type: int

name: marks

size: [200]

Disadvantages of using Arrays:

i. Poor time complexity of insertion and deletion operation

ii. Wastage of memory since arrays are fixed in size

iii. If there is enough space present in the memory but not in contiguous form, you will not be able to initialize your array

iv. It is not possible to increase the size of the array, once you have declared the array

Exercise 3 On Recursions: C Tutorial In Hindi #25

Exercise 3 :

Question : Write a program using recursive function to produce Fibonacci series of numbers?

fib\_recursive(int n):

if n==1 or n==2:

return n

else:

return fib\_recursive(n-1) + fib\_recursive(n-2)

To know about Fibonacci series : https://www.mathsisfun.com/numbers/fibonacci-sequence.html

So, here you have to simply produce Fibonacci series of no. 42 using both iterative and recursive logic.

Actually I gave this exercise to show that which method is good either iterative or recursive to find Fibonacci series of any no.

So try to solve it …

All the Best !!

Pointers in C:

What is a pointer?

i. Variable which stores the address of another variable

ii. Can be of type int(9, 10), char(c, d, 1, \*), float(11.3, 1.1), double array, function or any other pointer

iii. Size depends on the architecture. Ex: 2 bytes for 32 bit

iv. Pointer in C programming language can be declared using \* (asterisk symbol)

Intuitive Analogy - Stakced Boxes:

There are three boxes and you store fruits, dry fruits, screws

a = 0x45A1

p = 0xC10A [0x45A1]

Ex:

[Address of a] [Address of p]

a 0x45A1 p 0xC10A

-------------- --------------

| | <-------------- | 0x45A1|

-------------- --------------

Note: all of these are hexadecimal numbers

Pointer-to-Pointer is a method where a pointer stores the address of another pointer

a is an integer variable

p is a pointer to integer

p points to a

pointer to dry fruits meaning fruits stores address of dry fruits

-----------

int Var ------> | 7 |

-----------

Address of Var = A1

\* A1 will be stored in another variable

int \*ptr = &Var;

Now, \*ptr which is a pointer will store address of Var which is A1

a is a variable here whose address is 73

int \*ptr = &p; p which is a variable which has a different address of itself will store the address of another variable a

'&' and '\*' Operators:

a 0x45A1 <- &a p 0xC10A

------------ -------------

| 7 | | 0x45A1|

------------ -------------

int a = 7;

&a -> address of a

Here: p stores the address of a along with its own address

int \*p = &a; [stores the address of a; gives the value of an address of a variable]

i. The adress of operator '&' returns the address of a variable

ii. \* is the dereference operator (also called indirection operator) which is used to get the value at a given address

Null Pointer:

i. A pointer that is not asigned any value but NULL is known as the NULL pointer

ii. In computr programming, a null pointer is a pointer that does not point to any object or function

iii. We can use it to initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet

iv. int \* ptr = NULL;

Uses of Pointer:

i. Dynamic memory allocation

ii. Arrays, Functions and Structures

iii. Return multiple values from a function

iv. Pointer reduces the code and improves the performance

Ex:

a 0x45A1 p 0xC10A

------------ ------------

| | <--------------- | 0x45A1|

------------ ------------

Note: used in linked lists, stacks, etc.

| |N/A| |N/A| | <--- Memory

\ | /

o -> o -> o <--- Nodes

Can create linked lists; traversing, searching, etc.

a = " ";

i. 10 tasks - 1 - complete task 1 and freed

2 - complete task 2 and freed

3 - complete task 3 and freed

n

Arrays and Pointer Arithmetic in C:

C programming speed, efficiency, readability is equal to assembly language

Pointer Arithmetic:

i. There are four arithmetic operators that can be used on Pointers:

-> + +

-> - -

-> +

-> -

int a = 2;

a++; => 3

int \*ptra = &a;

ptra = ptra + 1;

size of(int);

Arrays and Pointers:

i. Consider the following declaration: int arr[10]; -> first element = arr [0]; // last element = arr [9];

ii. What is the type of arr? integer array

iii. However, arr, by itself, without any index subscripting, can be assigned to an integer pointer -> int \*ptr = arr; // [&arr[0] // ptr] same stuff

iv. What type does arr[i] have? int

arr[i] is same as \*(arr + i)

arr[i] using Pointer Arithmetic:

Note: arr[i] uses the pointer arithmetic to give the value of array

i. arr[i] == \*(arr + i)

ii. if arr is a pointer to arr[0] then arr + i is a pointer to arr[i]

Ex:

Sizeof(int) = 4

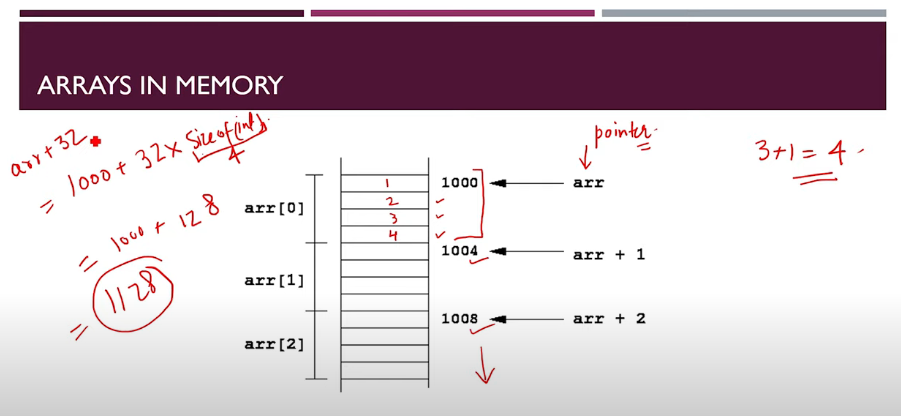
arr = 1 | 3 | 5 | 7 | 9

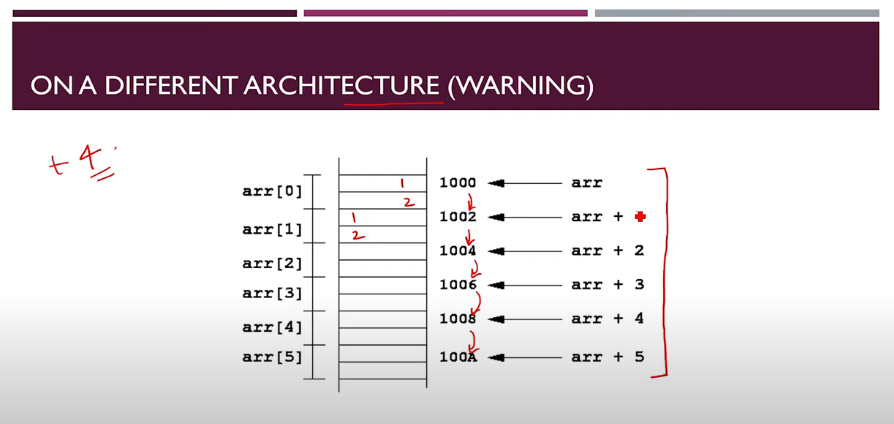
0 1 2 3 4 = 5 array elements

^ ^ ^ ^ ^

memory address = 1000 1004 1008 1012 1014

arr + i = 1004





Note: Size of integer depends on architecture to architecture

Why is recursion not always good?  
Why was recursive approach slow?

Ex:

fib(n)

1) fib(0) -> 0

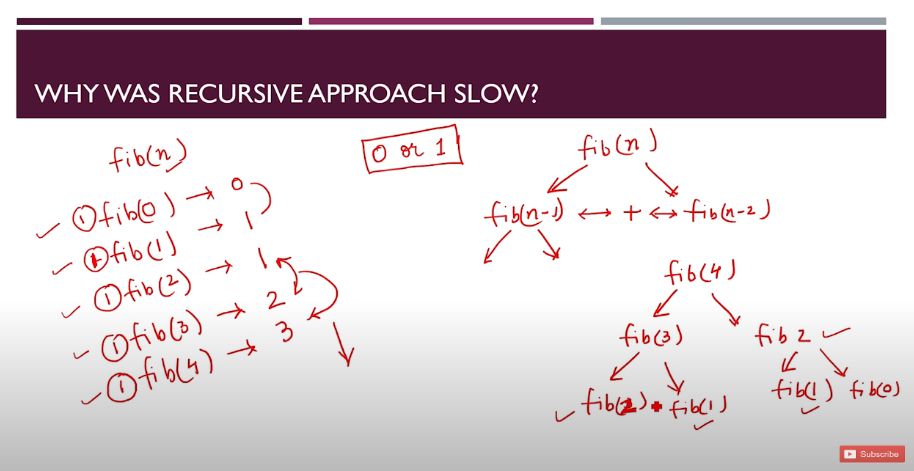
2) fib(1) -> 1

3) fib(2) -> 1

4) fib(3) -> 2

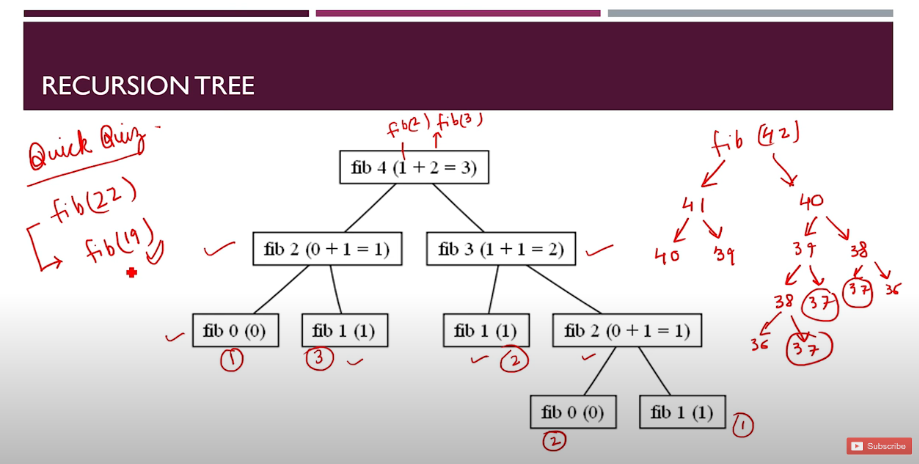
5) fib(4) -> 3

5 calls in total and if n = 4 then it will stop at 3



0s or 1s

fib (2) is calculated 2 times and fib (1) is calculated 2 times



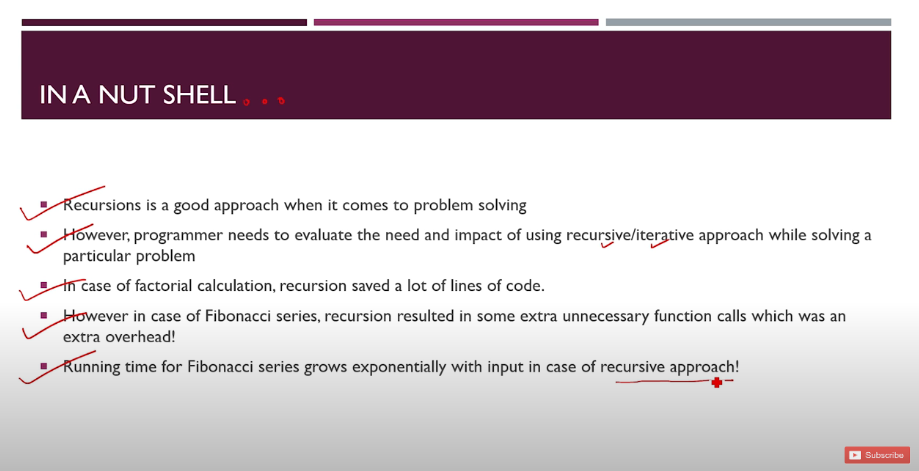
3 times 37 have been called and number of calls are exponentially increasing and overhead is more here. So through recursion we mismanaged time complexity.

Quick Quiz

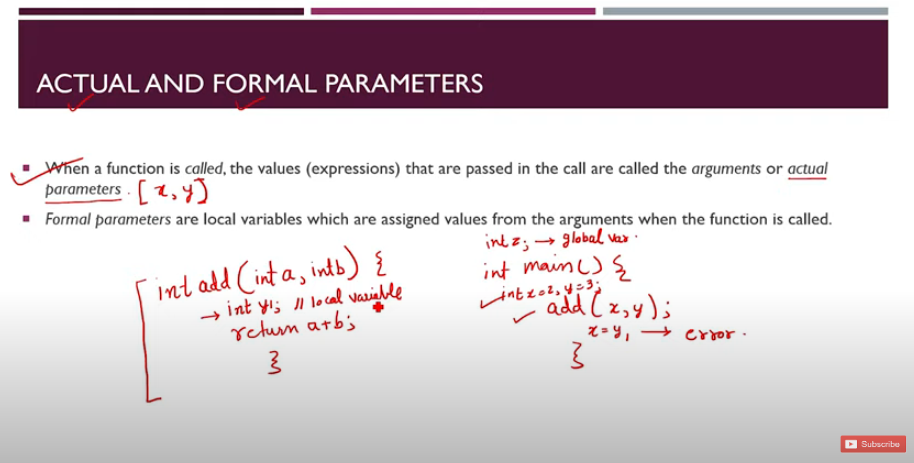
fib (22)

fib (19)

How many times is 19 called?



Actual Parameters:



Formal Parameters:

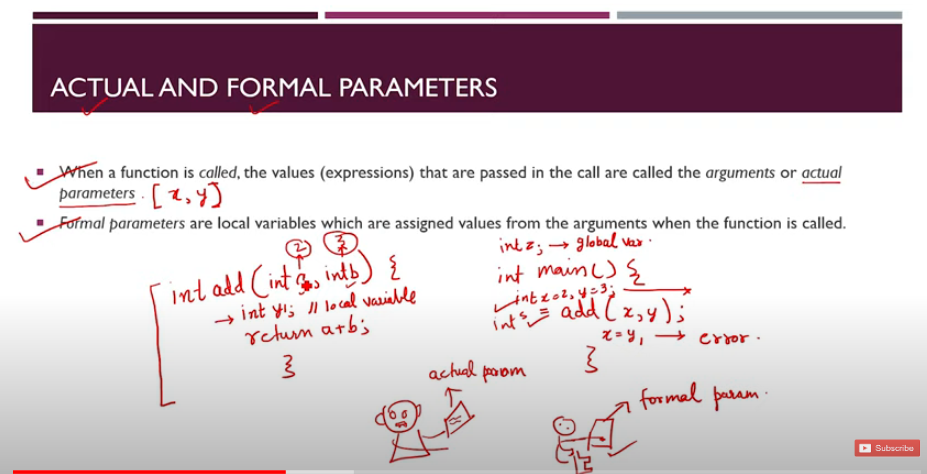
Formal Arguments = Formal Parameters

Actual Arguments = Actual Parameters

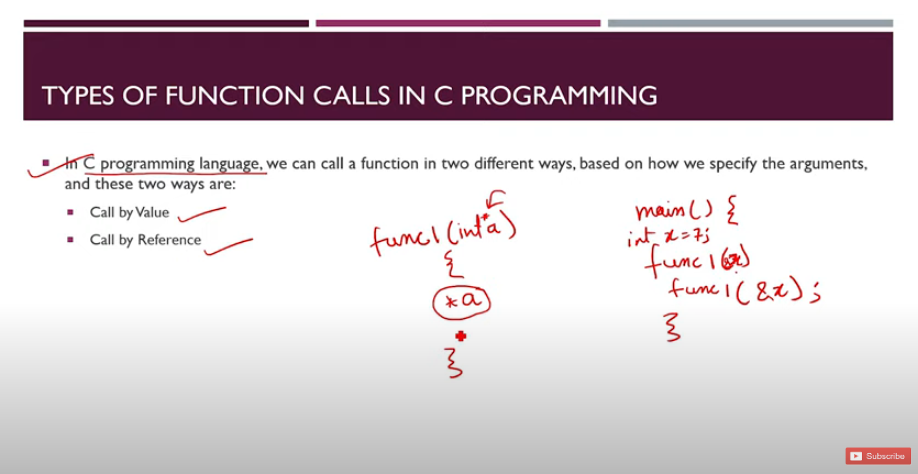
Ex: int y1 = local variable, z = global variable

// add (x, y) 🡪 Function called

Ex: Let’s say the teacher is actual parameter and the student is formal parameter so if the student copies and changes his answer will the teacher also do the same? No, actual parameters cannot change itself unlike formal parameters.



Actual parameters (x, y) are copied into formal parameters (a, b)



main () {

func1(&x) // function called, address passed

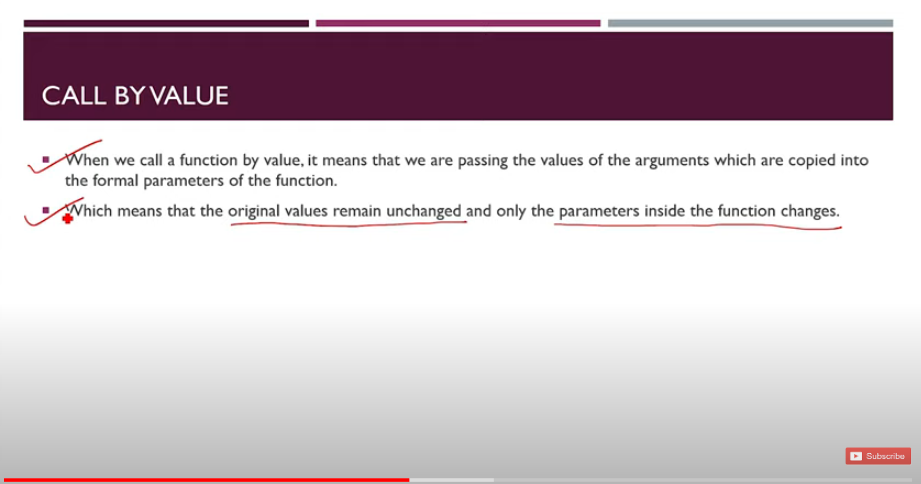
}

func1(int \*a) // function definition and &x’s address is copied and passed to int \*a pointer

{

\*a // dereferenced pointer, meaning &x actual value can now be changed since we have assigned address of &x to \*a pointer

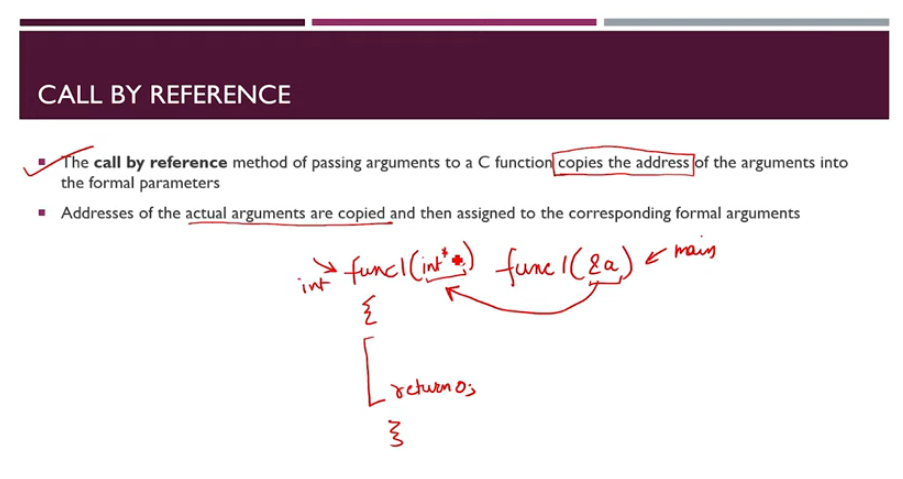
}



In call by value, values cannot be changed but in call by reference values can be changed

So far, we have the functions we have used is call by value only.

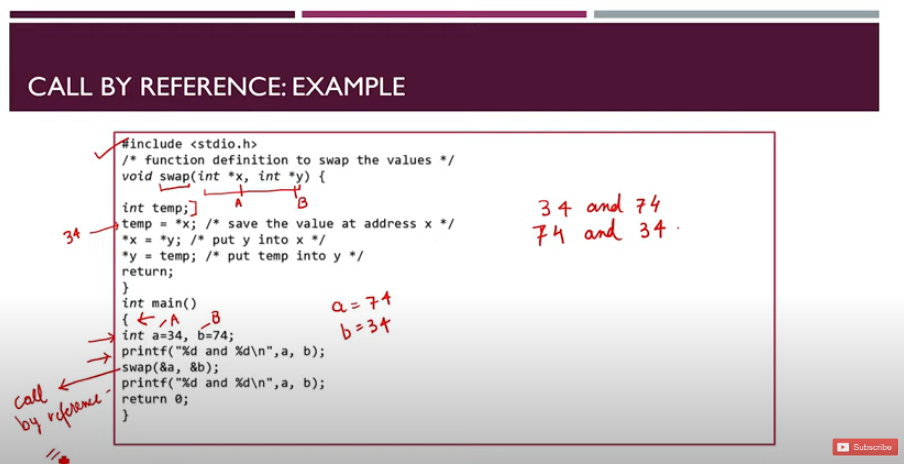
Now we are coming to call by reference



In call by value, value is copied & in call by reference, address is copied through which inside of the function body variable can be changed of the main function by passing the address.

Since here, main is a function which passes the address of &a to func1(int \*a) function and func1 function stores the address of the main function and dereferences and changes the actual value of the main function through the help of pointers.

All in all call by value and call by reference is a game of pointers.



Here swap is a function

\*x stores the value of a = 34

\*y stores the value of b = 74

Now, temp is a variable which stores the value of \*x which is 34

Then \*x changes the value of itself and stores the value of \*y which is 74

Similarly, \*y changes and stores the value of temp which is 34

Now the final output is:

a = 34 and b =74

-> After swap the value is changed to a = 74 and b = 34

