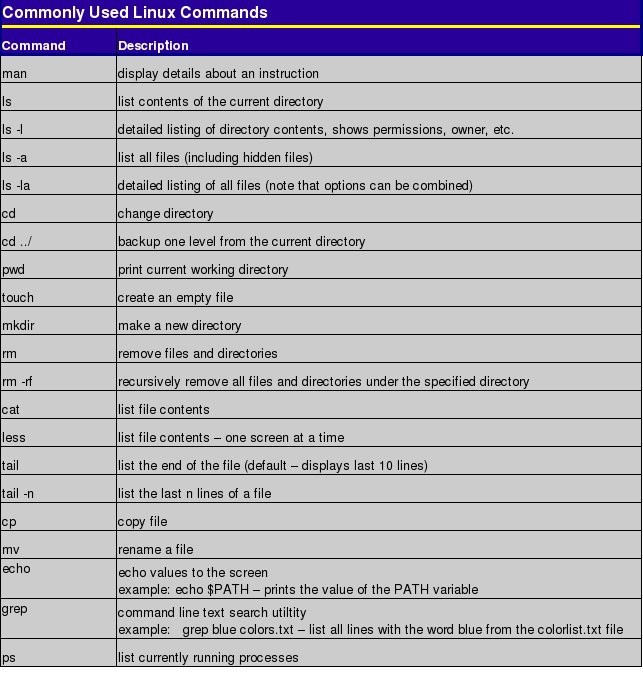
**BASIC LINUX COMMANDS**

****

****

**PREPROCESSOR DIRECTIVES**

#define is used to declare a macro (fragment of code).

#include is used to include a file.

#ifdef , #endif , #if , #else , #ifndef (conditions before compilation)

#undef (to undefine a defined macro) , #pragma (to call a function before and after main())

[#define circleArea(r) (3.1415\*(r)\*(r))]

[#define pi 3.14]

# HEADER FILES

#include<stdio.h> - standard input output header

#include<string.h> - string header

#include<conio.h> - console input output header

#include<math.h> - math header

.

.

.

( .h extension is necessary in C )

**ERRORS IN C**

* Syntax Errors (eg-missing (),{},; , printing variable without declaring , etc...)
* Run-time Errors (eg- divsion by zero , etc...)
* Linker Errors (eg- writing Main() instead of main() , etc...)
* Logical Errors (eg- Desired outputs are not obtained , etc...)
* Semantic Errors (eg- Statements are not meaningful to the compiler (void main{ int a,b,c; a+b=c; }) , etc...)

# Data Types in C

Each variable in C has an associated data type. Each data type requires different amounts of memory and has some specific operations which can be performed over it. Let us briefly describe them one by one:

Following are the examples of some very common data types used in C:

* char: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
* int: As the name suggests, an int variable is used to store an integer.
* float: It is used to store decimal numbers (numbers with floating point value) with single precision. (6 decimal places precision)
* double: It is used to store decimal numbers (numbers with floating point value) with double precision. (15 decimal digits precision) (If we add 0.16f to the value, the 16th decimal won’t be precise)

Different data types also have different ranges upto which they can store numbers. These ranges may vary from compiler to compiler. Below is list of ranges along with the memory requirement and format specifiers on 32 bit gcc compiler.

| Data Type | Memory (bytes) | Range | Format Specifier |
| --- | --- | --- | --- |
| short int | 2 | -32,768 to 32,767 | %hd |
| unsigned short int | 2 | 0 to 65,535 | %hu |
| unsigned int | 4 | 0 to 4,294,967,295 | %u |
| int | 4 | -2,147,483,648 to 2,147,483,647 | %d |
| long int | 8 | -2,147,483,648 to 2,147,483,647 | %ld |
| unsigned long int | 8 | 0 to 4,294,967,295 | %lu |
| 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 |
| signed char | 1 | -128 to 127 | %c |
| unsigned char | 1 | 0 to 255 | %c |
| float | 4 |  | %f |
| double | 8 |  | %lf |
| long double | 16 |  | %Lf |

long double – 19 decimal digits precision

**SIGNED AND UNSIGNED**

Refer the program:

{

unsigned int i=-1;

printf("%u",i);

}

Output - 4294967295

**MANAGING PRECISION**

"%0.2f", 0.123 -> 0.12 (zero padded min. width of 0, 2 decimal places).

"%6.2f", 0.123 -> \_\_0.12 (space padded min. width of 6, 2 decimal places).

"%06.2f", 0.123 -> 000.12 (zero padded min. width of 6, 2 decimal places).

"%0.6f", 0.123 -> 0.123000 (min width of 0, 6 decimal places).

GENERALLY, First number gives the padding element, space or zero (default space), Second number(can be two or more digits also) gives the min width (default 0) and the one after decimal gives the no. of decimals to be included.

To add 4 decimals to a value use %0.4f or %.4f

Eg:-

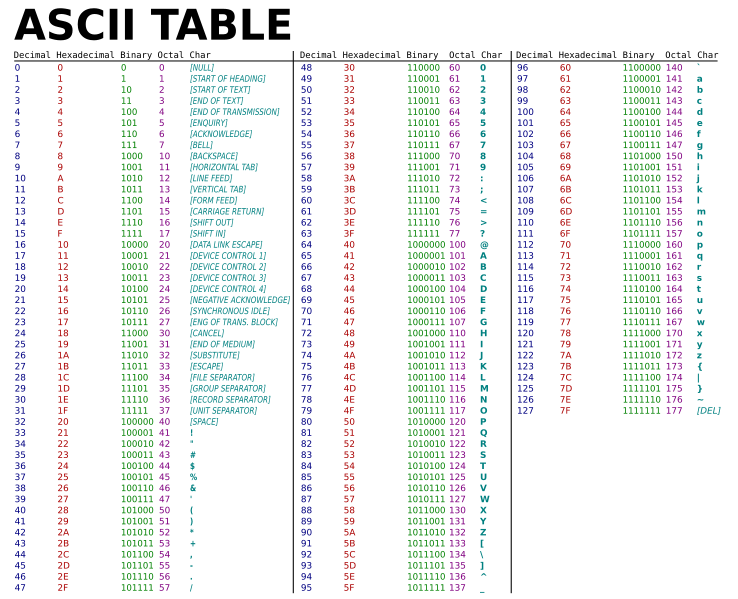
float num=5.46718722;

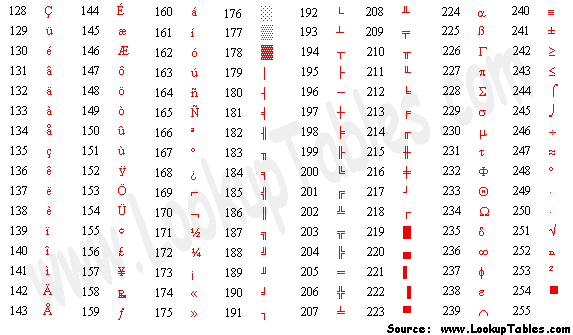
printf(“%0.4f”,num); //%0.4lf if data type is double and %0.4Lf for long double

Output – 5.4672 (rounding off)

If we add f to a number, it’ll be converted to float for that operation.

(Eg- int a=2.47372364f+2.3287466f; means the values will be converted to float, then added and then converted back to int) [ Not for lf or Lf ]





**PRECEDENCE AND ASSOCIATIVITY**

|  |  |  |
| --- | --- | --- |
| 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 |

b=++a; (incremented value is assigned)

b=a++; (value is assigned and then incremented)

a<<1; (Left shift by 1 but no change to a)

a >>= 2; (Right shift by 2 and assignment to a)

**TYPE CONVERSION**

### **IMPLICIT TYPE CONVERSION** (done by compiler) :-

**If a and b are int, res=a/b will also be int(irrespective of data type of res).**

EG:-

int a=4.7,b=5.7;

float res=a/b;

Output :- res=0.000000

**If either a or b or both is float, then a/b will be float and res=a/b will depend on the data type of res.**

EG:-

int a=4.7;

float b=5.7,res=a/b;

Output :- res=0.701754

float a=12.3,b=3.9;

int res=a/b;

Output :- res=3

### **EXPLICIT TYPE CONVERSION** (done by programmer) :-

struct person per1={‘a’,‘b’,‘c’};

struct person \*p;

\*p=&per1;

printf(“%d\t”, p); //memory location of structure pointer

printf(“%c %c %c”, \*((char\*)p), \*((char\*)p+1), \*((char\*)p+2)); //type conversion

printf(“\t%d”, p); //type conversion occurs only inside the print statement

Output :- 0x7ffd196da70d a b c 0x7ffd196da70d

**VARIABLES (IDENTIFIERS)**

* Unique
* Length is compiler dependent
* Letters, digits and underscore
* No spaces
* Upper and lowercase letters are distinguished
* First character must be a letter or underscore
* Cannot be a keyword (if,else,while,for,etc...)
* main can be used as a variable name

**LITERALS**

Values assigned to the variables. Can be changed if needed.

* Integers (Decimal, Octal(starts with 0) and Hexadecimal(starts with 0x))
* Floating-point literals (0.02, -2.0, 2.5E-3)
* Characters (‘a’, ‘F’, ‘2’, ‘{’)
* Escape sequences (\b, \f, \n\, \r, \t, \v, \\, \’, \”, \?, \0)
* String literals (“good”, “”, “ “, “x”, “Earth is round\n”)

**CONSTANTS**

Variable whose value cannot be changed. Keyword – const or using #define.

const float pi=3.14;

pi=4; //ERROR

#define three 3

...

result=sqrt(three);

**ARRAYS**

* Identifier points to the first element of array (printf(“%d”,\*arr); gives the element in arr[0]).
* Array CANNOT be initialised as int arr[]; CAN be either, int arr[size]; or, int arr[]={1,2,3}; or, int arr[3]={1,2,3};
* Unfilled spaces are filled with 0 and undefined spaces are filled with garbage values.
* Character storing arrays are called strings in C.

# STRINGS

* Properties similar to arrays. ( Identifier points to the first character )
* Initialised as, char arr[]= “Hello World”; or, char arr[6]=”Hello”; or, char arr[6]={‘H’,‘e’,‘l’,‘l’,‘o’,‘\0’}; or, char arr[]={‘H’,‘e’,‘l’,‘l’,‘o’,‘\0’}; or, char arr[size];
* Strings can also be initialised using pointers, char \*str = “Hello World”; but there is difference between \*str (a pointer and so, a variable) and str[] (An array and so, not a variable).
* Terminated by \0 (NULL).
* Declaring by char arr[] gives garbage values for undefined %c but no problem for %s. (MAY BE COMPILER DEPENDENT)
* If string is filled character-wise, \0 must be included at the end. Otherwise, garbage values will be stored in %s also.
* Can be inputted by scanf, gets(), fgets(), getline() [ similar to fgets but more reliable. Difference is that it uses address instead of identifier – getline(&str, &size, stdin); ] or getchar(). (scanf %s will only input a single word nothing after a space is included in it so use “%[^\n]%\*c”) (gets reads until a new line.) (getchar reads the same as reading an array) (fgets reads until a new line or end of file (fgets(str, 20, stdin);) (gets may show warning, ignore it)
* fgets(str); //strlen(str) gives length including a newline at the end.
* Can be outputted by printf, puts() or putchar() (puts(str) outputs the string with a new line at the end).
  + - printf() can also be carried out as follows :

char a=’a’, \*str= “Hello World %c %d”, \*strr=”Hello World”;

int b=1;

printf(str, a, b); //Gives ouput Hello World a 1

printf(strr, a, b); //Error, format specifiers required for non-string variables

printf(a,b); //Error, this works only for one string variable

printf(a); //Error, this works only for string variables

* String handling fuctions – strlen(), strlwr(), strupr(), strcpy(), strcat(), strcmp()
* strcmp() after checking both strings ( int result=strcmp(str1,str2); ) until characters are unmatched or \0 is encountered in any, gives 3 outputs

1. 0 – If all characters inlcuding \0 are equal.
2. >0 – If ASCII value of first string character is higher.
3. <0 – If ASCII value of first string character is lower.

* strlwr() - Lowercase all characters.
* strupr() - Uppercase all characters.
* strlen() - Length of string excluding \0. ( printf(“%zu”,(strlen(str)) or store it in an int l=strlen(str); )
* strcpy() - Copies contents of second string to first. ( puts(str1); gives the copied string )
* strcat() - Concatenates second string to the first string and result is stored in first.
* 2D strings are also there. (char str[20][20];) ( char str[]={“Hello”,”World”}; )

for(i=0;i<20;i++)

{

for(j=0;j<20;j++)

scanf(“%s”,str[j]); //Only the row no. needs to be specified

}

STRUCTURE & UNION

* Definition should end with “;”.
* struct – all values can be retrived at once, size is sum of all variables.
* union – only one value (finally intialised value) can be retrieved at a time, size is that of largest variable.
* Variables can be accessed using “.”.

**FUNCTIONS**

* Data type is of return value. (default - int)
* No return value means void.
* If return value; is included, program control is suddenly transferred out of the function to wherever it was called.
* Definition can be written anywhere but declaration ( void func(); is enough. No need to mention the arguments during declaration ) must be written before calling a function.
* Array can be passed as a parameter by passing the identifier name alone (reverse(arr)). In the function declaration, data type of the array and parameter name must be specified but array size need not be specified (void reverse(char arr[])).
* While passing 2D array, max no. of elements in each row has to be specified (ie, no. of columns has to be specified) (void transpose(int a[][30])).
  + Read(a,b);
  + if(Read(a,b)==0)
    - ............

Here, the function is executed twice.

INFINITE LOOPS

* while(1)
* for(;;)
* while(0) doesn’t enter the loop

STORAGE CLASSES

**GLOBAL or EXTERNAL VARIABLES**

* Variables that are alive throughout a program and can be used anywhere (in any function)
* Either defined outside of all functions ( #include<stdio.h> float pi=3.14 )
* Or we can also define it inside any function using extern keyword ( void main() { extern float pi=3.14; } )
* Used when we need a variable with the same value for all functions
* Constants (const) can also be made globally accessible
* Eg:- #include<stdio.h>

int Gvar=0;

void main()

{

int Gvar=5;

}

Here, the value of Gvar changes from 0 to 5 inside the function but if it’s called by another function it’s value will be 0 but to retain the value from one function to another simple lose the datatype. (Refer below)

#include<stdio.h>

float pi=3.1415;

void func()

{

printf(“%f ”,pi);

pi=3.14; //refer below

}

void main()

{

printf(“%f ”,pi);

pi=3; //refer below

printf(“%f ”,pi);

func();

printf(“%f\n”,pi);

}

Output – 3.141500 3.000000 3.000000 3.140000

If float pi=3.14; and float pi=3; output will be 3.141500 3.000000 3.141500 3.000000

# AUTOMATIC AND STATIC VARIABLES

* Both **auto** and **static** variables are local variables (available only in their respective functions).
* **static** variables can retain the value of the variable between different function calls.
* But, scope of **auto** variable is within the function only. It can't retain the value of the variable between different function calls.
* **auto** is the default variable type in C. (So **auto** keyword is optional)
* **auto** is filled of garbage values by default while **static** is filled with 0.
* **auto** is initialised everytime when the function (in which it’s declared) is called and is destroyed after program control leaves the function.
* **static** is initialised once and is destroyed only after the program execution.
* Eg:-

#include <stdio.h>

**void** fun(**void**)

{

**auto** **int** a=0;

**static** **int** b=0;

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

a++;

b++;

}

**void** main()

{

**int** loop;

**for**(loop=0; loop<5; loop++)

fun();

}

**Output**

a = 0, b = 0

a = 0, b = 1

a = 0, b = 2

a = 0, b = 3

a = 0, b = 4

# REGISTER VARIABLES

* Registers are faster than memory to access, so the variables which are most frequently used in a C program can be put in registers using **register** keyword.
* Properties similar to **auto** but has faster access.
* It can also store pointers like **auto**.

### COMPARISON -

* **extern** and **static** has lifetime until the program ends, while **auto** and **register** has until the function ends.
* **extern** has scope througout the program ( global scope ), while **auto**, **static** and **register** has scope in the function only ( local scope ). ( **static** also has scope between function calls )
* **extern** is declared outside or inside a function, while **auto**, **static** and **register** are declared inside a function alone.
* **extern** has visibility throughout the program, while **auto**, **static** and **register** has visibility throughout the function alone. ( SCOPE AND VISIBILITY ARE ALMOST THE SAME )

POINTERS

* char \*ptr; and char\* ptr; is the same. (Data type is not of address but of the variable)
* &(ampersand) is used to assign and \*(asterisk) is used to access.
* \*ptr gives the value of variable while ptr gives its address.
* scanf(“%d”,\*ptr); gives segmentation fault. (Same for int \*ptr = 1;)
* Scanf(“%d”,&a); a=\*ptr; is the right way.
* %p is used to print pointer (address)
* Passing to a function is carried as, void swap(int \*ap,int \*bp); and swap(&a,&b). Since the memory address is passed the variable is changed in both functions. [ Known as call by reference ]
* NULL pointers \*ptr=NULL; here, value of ptr is 0. (Actually, 0 is taken by OS but by convention, pointers that have value 0 are meant to point to nothing)
* if(ptr) proceeds if \*ptr is not NULL. Similarly if(!ptr) proceeds if \*ptr is NULL.
* Operations that can be performed are ++,--,+ and -.
* ptr++ and ptr-- points to the next or previous location (supposedly in an array). (char pointer moves from 1000 to 1001 while int pointer moves from 1000 to 1004)
* Pointers can be compared ( ptr<=&a[i] ).
* Pointer to pointer (int \*\*p) accessed by \*\*.
* func(int \*ptr) is the declaration while passing an pointer to a function. ( This pointer can point to the various elements of an array using ptr++[Note that ptr is not the identifier of the array] )
* arr points to the first element, so func(arr); can be used to pass the array pointer that points to first element but this pointer cannot be incremented or decremented ( error is lvalue required, maybe it can be incremented or decremented too in printf (“%d”, \*(ptr+i)); ) (but string pointer can be incremented or decremented).
* int \*ptr[size]; is used to declare array using pointer.
* \*\*ptr is equivalent to \*ptr[0] and arr[0]. We cannot use ptr++ here since it is used to point to the array’s first element (It should not be modified). (This is not for string pointer given below)
  + char \*ptr = “Hello World”; can be used to declare string using pointer. \* is not required to access the string ( printf(“%s %p %c”, ptr, ptr,\*ptr); )
  + ptr++ gives ello World.
* To create an list of strings using pointer:-

char \*names[] = {

"Zara Ali",

"Hina Ali",

"Nuha Ali",

"Sara Ali"

};

* Here,

Value of names[0] = Zara Ali ( When stdin(gets or fgets (exclude scanf %s)) to 2D array, the first string sometimes becomes “\n”. Use scanf(“\n”); before stdin as a solution to this )

Value of names[1] = Hina Ali

Value of names[2] = Nuha Ali

Value of names[3] = Sara Ali

* {“%s”, names[i]} gives the entire string at ith row.
* {“%s”,\*names} gives the first string.
* You cannot scanf() these strings using %s because it is similar to reading a pointer directly using scanf(). (Segmentation fault)
* Instead allocate memory for it using malloc() OR store the string in a normal char array and then copy it to the string using strdup().

**CALL** **BY** **VALUE**

#include <stdio.h>

void swapx(int x, int y);

int main()

{

int a = 10, b = 20;

swapx(a, b);

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

return 0;

}

void swapx(int x, int y)

{

int t;

t = x;

x = y;

y = t;

printf("x=%d y=%d\n", x, y);

}

**Output** :

x=20 y=10

a=10 b=20

**CALL BY REFERENCE**

void swapx(int\*, int\*);

int main()

{

int a = 10, b = 20;

swapx(&a, &b);

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

return 0;

}

void swapx(int\* x, int\* y)

{

int t;

t = \*x;

\*x = \*y;

\*y = t;

printf("x=%d y=%d\n", \*x, \*y);

}

**Output** :

x=20 y=10

a=20 b=10

# DYNAMIC MEMORY ALLOCATION

When array size has to be increased or decreased. Declared using pointers. Header <stdlib.h> has to be used.

* malloc() - memory allocation, A single large block of specified size is initialised with garbage values by default (can also be considered as non-initialised). { ptr = (cast-type\*) malloc(byte-size) }
  + ptr = (char\*) malloc(5\*sizeof(char));
* calloc() - contiguous allocation, individual blocks of specified size are intialised with default 0 values. { ptr = (cast-type\*) calloc(n, element-size); }
  + ptr = (int\*) calloc(5, sizeof(int));
* free() - frees allocated memory. { free(ptr); }
  + free(ptr);
* realloc() - Re-allocates allocated memory and new blocks are initialised with garbage values. { ptr = realloc(ptr, newSize); }
  + ptr = realloc(ptr, 10\*sizeof(int));

TO REFER !

* Isalpha(), abs(), atoi(), etc...
* Large size handling
* Diff for input-output and processes
* GCC can’t store very large values other than input or output.
* %1000000007
* ((m%n)^2)%n = (m^2)%n (Check validity)
* The reason of taking Mod is to prevent integer overflows. The largest integer data type in C/C++ is unsigned long long int which is of 64 bit and can handle integer from 0 to (2^64 – 1). But in some problems where the growth rate of output is very high, this high range of unsigned long long may be insufficient.
* \_\_int64\_t
* typedef long long int int64\_t
  + from library stdint

typedef signed char int8\_t;

typedef short int int16\_t;

typedef int int32\_t;

# if \_\_WORDSIZE == 64

typedef long int int64\_t;

# else

\_extension\_

typedef long long int int64\_t;

# endif

#endif

/\* Unsigned. \*/

typedef unsigned char uint8\_t;

typedef unsigned short int uint16\_t;

#ifndef \_\_uint32\_t\_defined

typedef unsigned int uint32\_t;

# define \_\_uint32\_t\_defined

#endif

#if \_\_WORDSIZE == 64

typedef unsigned long int uint64\_t;

#else

\_extension\_

typedef unsigned long long int uint64\_t;

#endif

/\* Small types. \*/

/\* Signed. \*/

typedef signed char int\_least8\_t;

typedef short int int\_least16\_t;

typedef int int\_least32\_t;

#if \_\_WORDSIZE == 64

typedef long int int\_least64\_t;

#else

\_extension\_

typedef long long int int\_least64\_t;

#endif

/\* Unsigned. \*/

typedef unsigned char uint\_least8\_t;

typedef unsigned short int uint\_least16\_t;

typedef unsigned int uint\_least32\_t;

#if \_\_WORDSIZE == 64

typedef unsigned long int uint\_least64\_t;

#else

\_extension\_

typedef unsigned long long int uint\_least64\_t;

#endif

/\* Fast types. \*/

/\* Signed. \*/

typedef signed char int\_fast8\_t;

#if \_\_WORDSIZE == 64

typedef long int int\_fast16\_t;

typedef long int int\_fast32\_t;

typedef long int int\_fast64\_t;

#else

typedef int int\_fast16\_t;

typedef int int\_fast32\_t;

\_extension\_

typedef long long int int\_fast64\_t;

#endif

/\* Unsigned. \*/

typedef unsigned char uint\_fast8\_t;

#if \_\_WORDSIZE == 64

typedef unsigned long int uint\_fast16\_t;

typedef unsigned long int uint\_fast32\_t;

typedef unsigned long int uint\_fast64\_t;

#else

typedef unsigned int uint\_fast16\_t;

typedef unsigned int uint\_fast32\_t;

\_extension\_

typedef unsigned long long int uint\_fast64\_t;

#endif

/\* Types for `void \*' pointers. \*/

#if \_\_WORDSIZE == 64

# ifndef \_\_intptr\_t\_defined

typedef long int intptr\_t;

# define \_\_intptr\_t\_defined

# endif

typedef unsigned long int uintptr\_t;

#else

# ifndef \_\_intptr\_t\_defined

typedef int intptr\_t;

# define \_\_intptr\_t\_defined

# endif

typedef unsigned int uintptr\_t;

#endif

/\* Largest integral types. \*/

#if \_\_WORDSIZE == 64

typedef long int intmax\_t;

typedef unsigned long int uintmax\_t;

#else

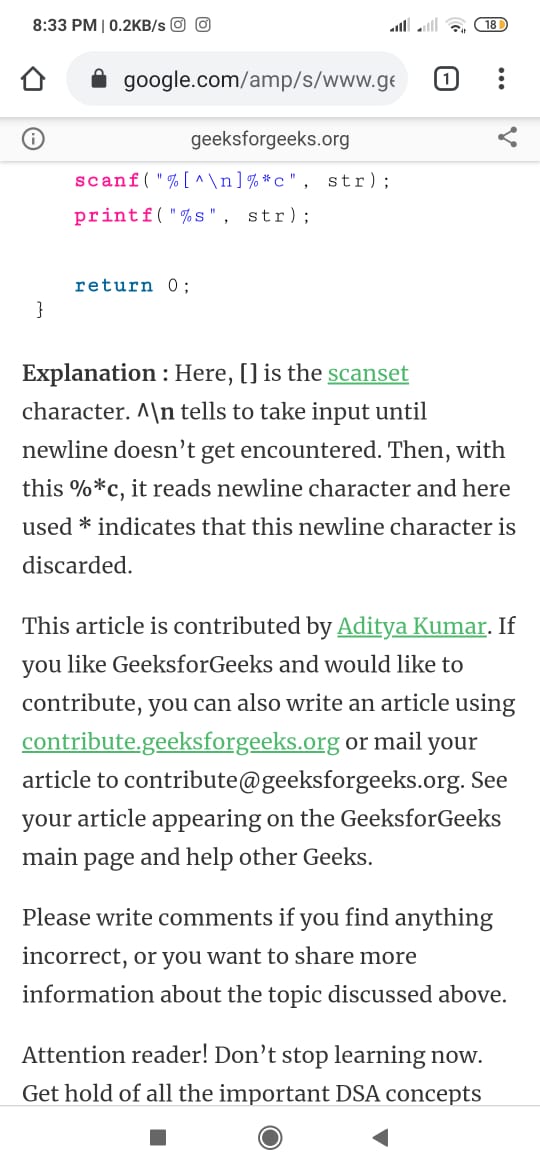
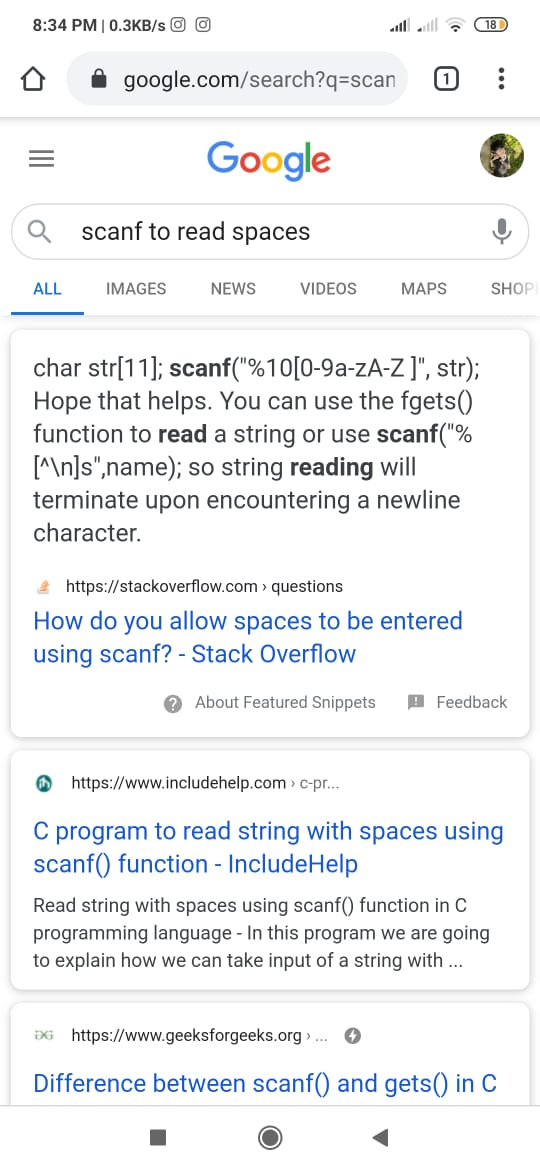
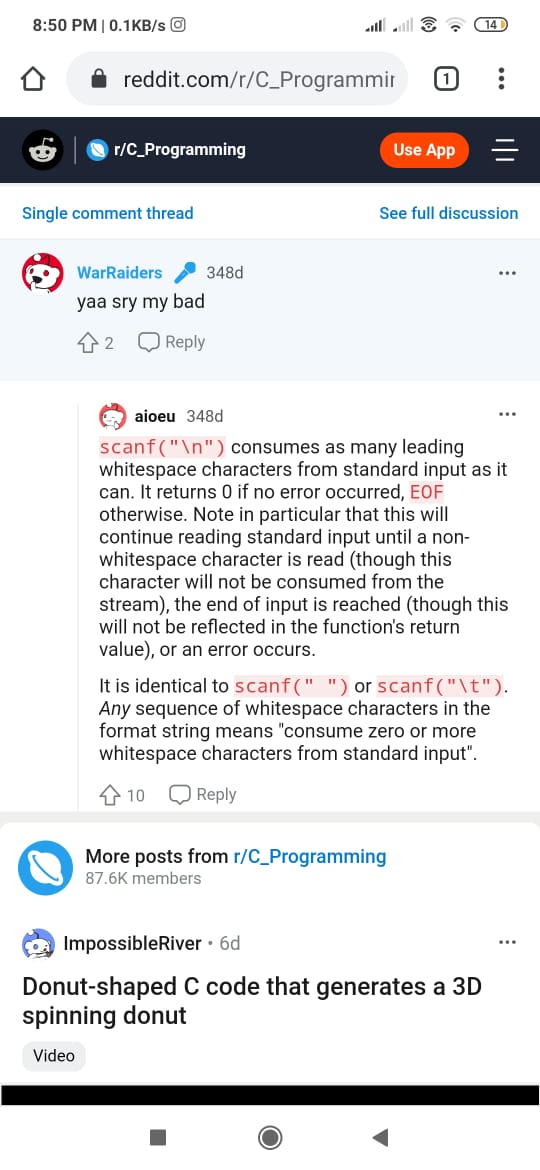
\_extension\_

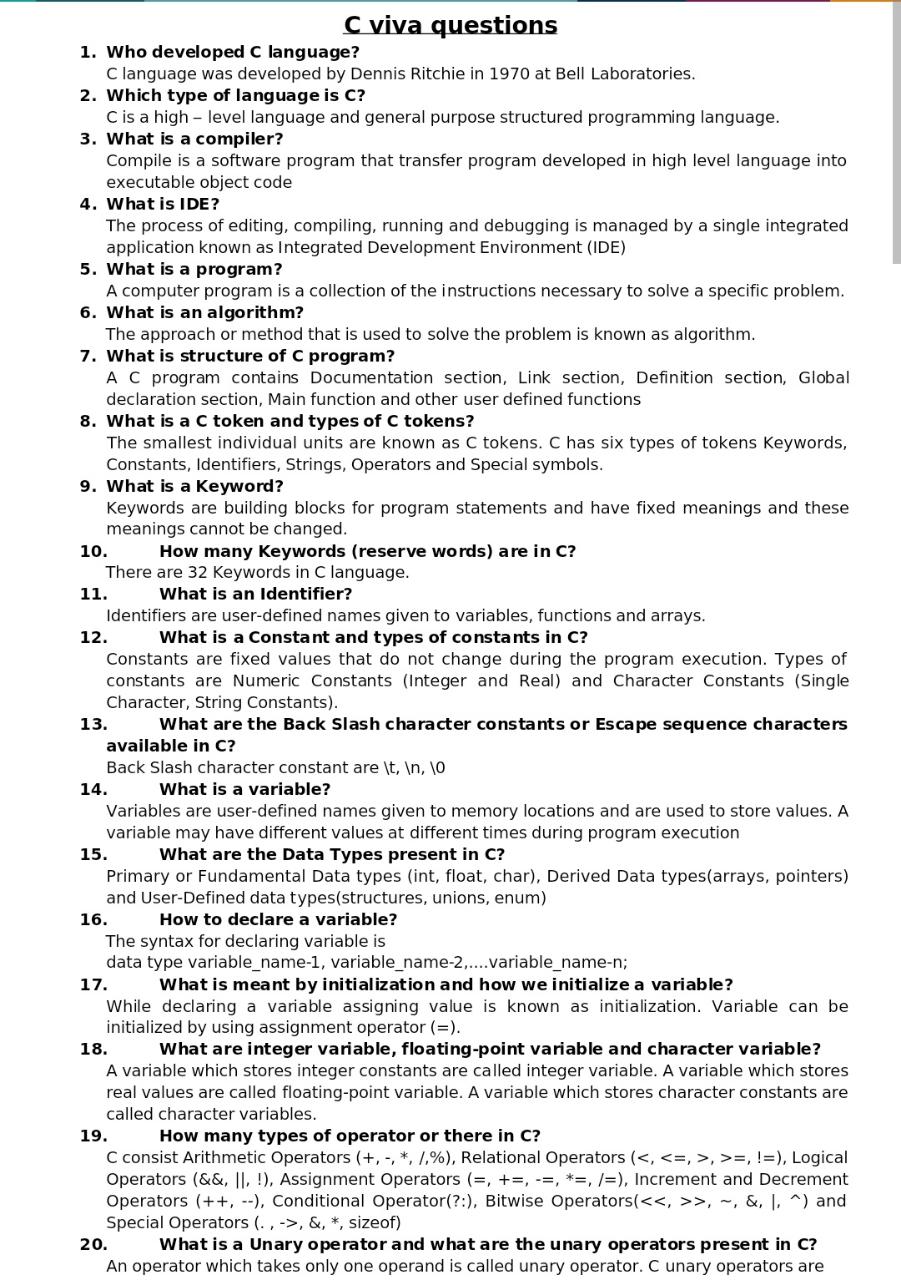
typedef long long int intmax\_t;

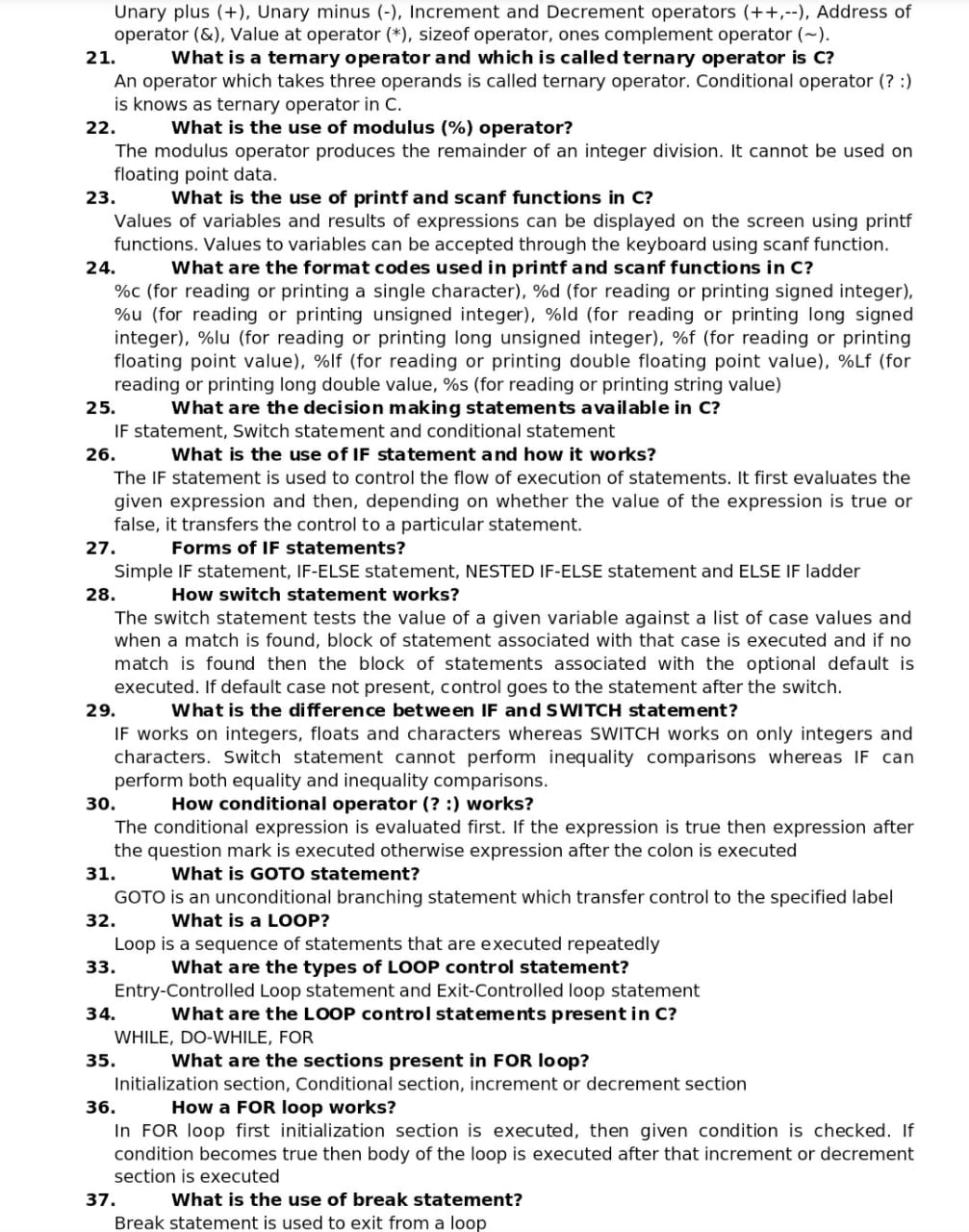
\_extension\_

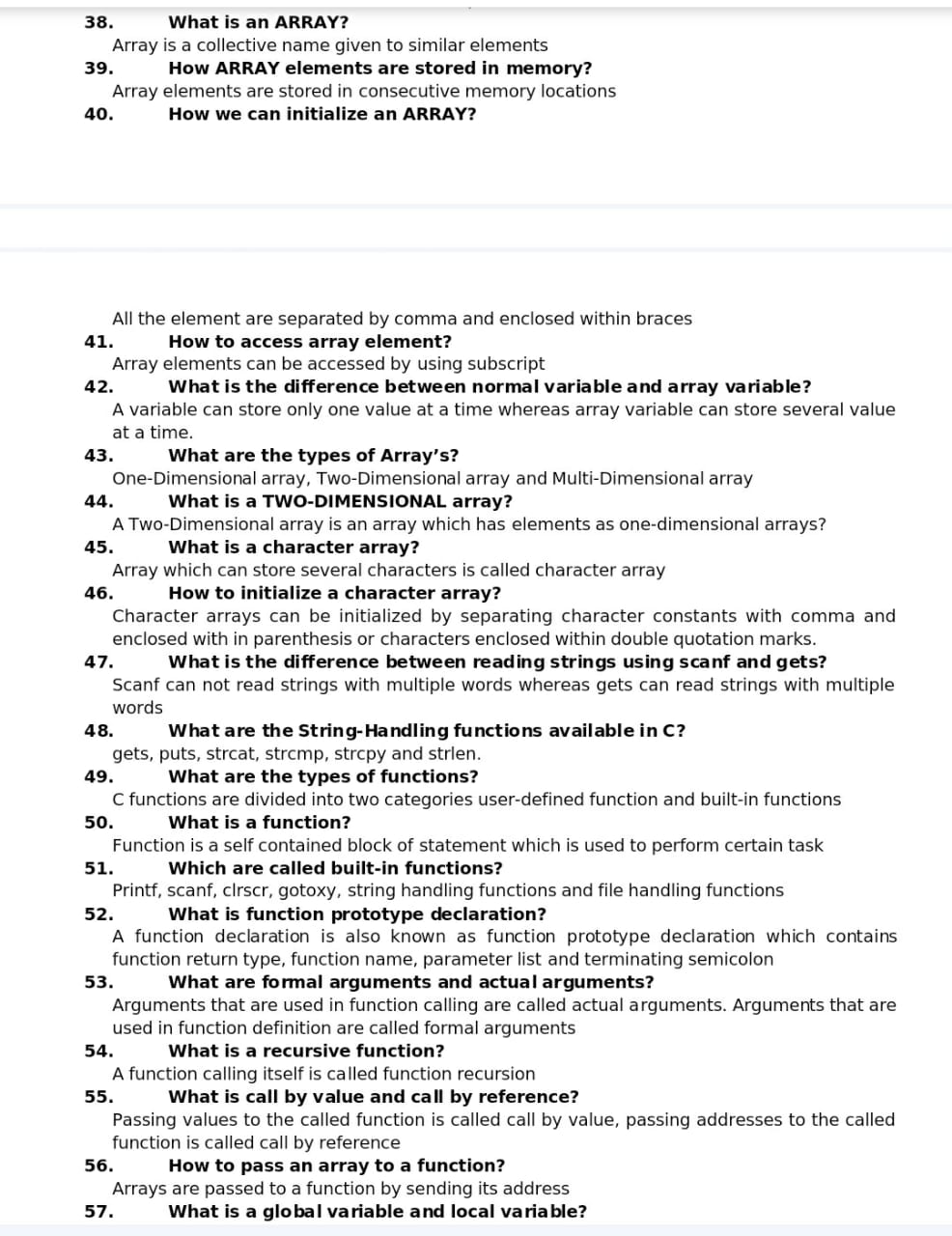
typedef unsigned long long int uintmax\_t;

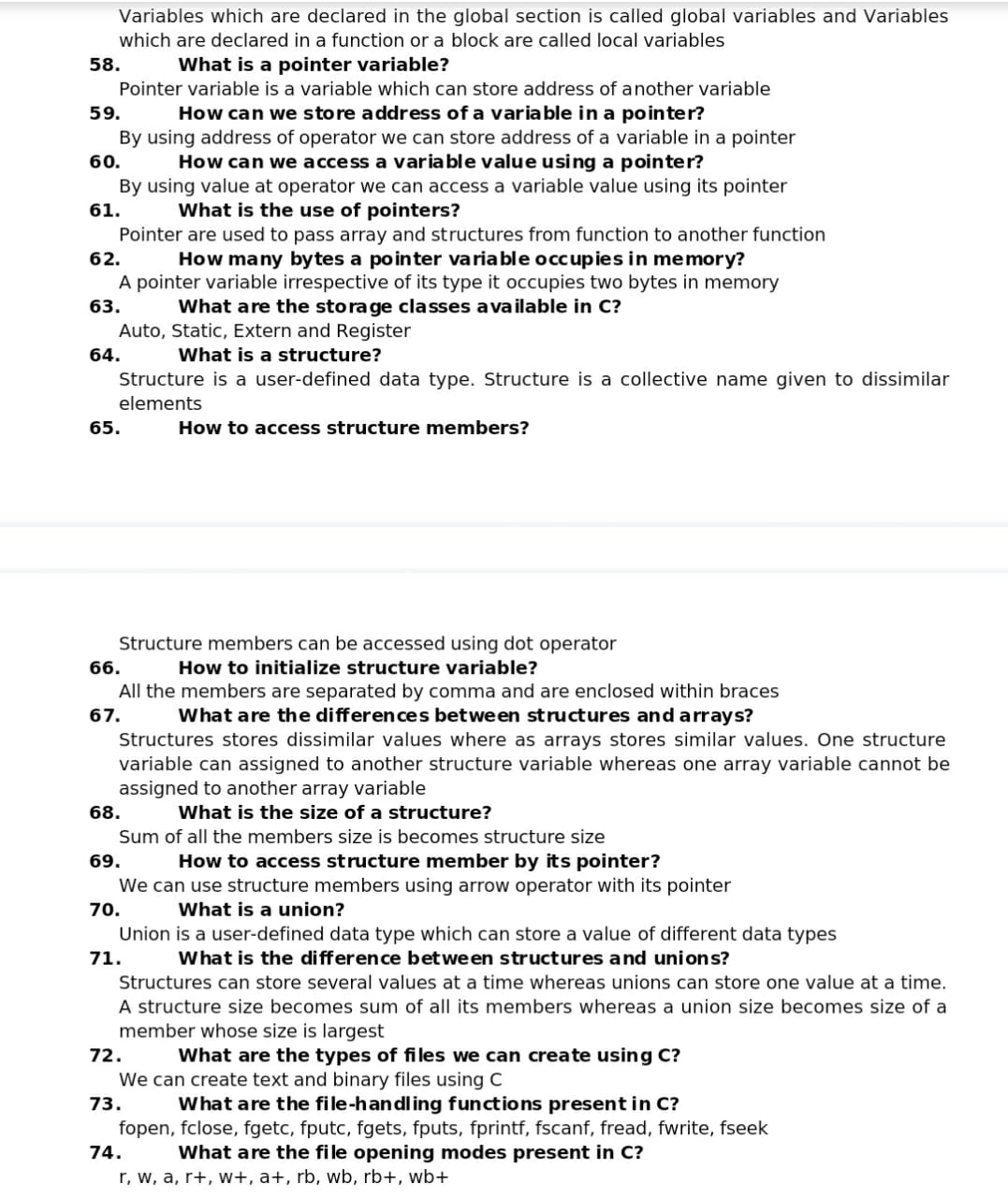
* https://sites.uclouvain.be/SystInfo/usr/include/stdint.h.html
  + Nice library must read
* Argument passing to function args, argv etc...
* -> operator
* OOP keywords in C (static, abstract, etc...)
* Function pointer
* Abstraction in C
* Vectors in C/CPP











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