Static with local variables

```
static
int add (int x, int y){
return x+y;
static int Counter= 0;
Counter++;
}
main (){
int result= add (5,3);
add (5,4);
```

this file is stored in the 2nd Call because it's already defined

→ Counter after function executed is not deleted because of static

So Counter = 2

Automatic:

local variable → destroy the end of the function

Static local variable → allocated end of the function

Scope and extern example

```
int sum (int, int);
int main ()
{
int result;
Cout << result << endl;
result = add(5,3);
Cout << result;
return 0;
}
int add (int x, int y)
{ return x+y;
```

→ here it's garbage value because no value assigned to result yet

→ here result = 8 because it's after the function call

ALADIB

Scanned with CamScanner

- function by default is extern So if we move add to another file the program will run.

Static variable is allocated to the memory as long as the program is running

Static int counter = 0; → excuted only once inside the function
global variable has an initial value of = Zero 0.

int counter; → definition of global variable = 0 allocated in the memory
extern int counter; → declaration of the variable in another file

Global can be accessed in any scope of the program unlike static variable which can be only accessed inside its scope.
Static variable can't be used in any on the files

Local variable ___ auto,
global variable ___ scope project ___ share resource
___ static with global ___ scope ___ file-Comm

global variable ___ scope project ___ extern
___ variable declaration in the other file
extern int count;
extern with function → default → extern

Scanned with CamScanner

```
unsigned char x;        /* a regular global variable */
void sub (void) {
  x = 1;
{ unsigned char x;      /* a local variable */
  x = 2;
{ unsigned char x;      /* a local variable */
  x = 3;
  PORTA = x; }          <- local
  PORTA = x; }          <- global
  PORTA = x; }          <- global
}
```

Note: if there's a local and global variable with the same name the local should hold the value

global note

- file.h
  extern int global_variable /* Declaration of variable */

* file.c
  #include <file.h>
  int global_variable /* Definition of variable */

file 2.c:
  #include <file.h>
void use(void){
  global_variable++; /* Use of variable
  }

file3.c
static int global_variable; → same name but not the same variable in 1, 2

ALADIN
Scanned with CamScanner

- C's scoping rules specify that the scope
- of an identifier begins at its point of declaration rather than the top of the block in which it is defined.

float i = 5.f; → Global

{ int j = i;
i = i + 5; // global
int i = 0; // beginning of local identification
float → another local variable
j i = 10 → for the local variable not the global because a local variable is identified
}

Pointers :

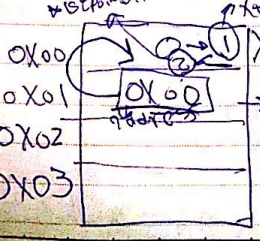
A pointer is a ~~data~~ variable that contains the address of a variable

the pointer data type represents values used indirectly

All data stored in Computer memory as a series of ones and zeros.

When the computer evaluates a pointer value, it reads the ones and zeros as a memory address

C data types acts as filters which interpret these ones and zeros



unsigned char x = 0

char *isPointer = &x

isPointer → Char Pointer → Pointer name

&x → address of x

Pointer to Character

To change ex: X++; divece points to → *isPointer ++;

Scanned with CamScanner

int* ptr; float* ptr; char* ptr;

1st Pointer is an address

* usages:

access operators

1- Declaration of pointer.
2- to access the value to adjust it (item) using the address.

Note: The pointer should the same type of the value which it points to.
int*ptr=&x;
int

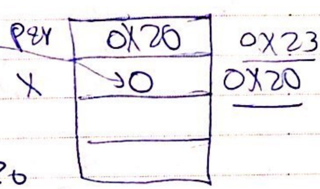
1st Pointer

lcdout(&1stpointer) → 0X01 → address of the pointer itself as it is shown in the last figure

(1st pointer) → 0X0B
lcdout(1stpointer) → address of the value

& return address of the variable
ex = &X

X++; — X=1
ptr++;
address



ptr=21
ptr--; ptr=20

x=9; direct
*ptr=10; indirect

*ptr=7 → *(&x)
→ x=7
lcd_out(ptr) → 0x20
lcd_out(*ptr) → 7
lcd_out(X) → 7
lcd_out(&x) → 0X20
lcd_out(*ptr) → 7

* lcdout(&ptr) → 0X23

Type casting :-
changing the type of a variable

float x = 12,3 ;
char y = x ; 12 is ok so we need type casting

type casting: cast float

int x = 12; Note:- float accepts
float y = x; integer value
y = 12,00 so we don't need casting

but when we change from float to integer casting happens.

Note: Compiler by default is responsible for casting

ex. float x = 12,3;
int y = (int) x ;

Compiler does that not typed by the user

Note:-

char x = 53;
int y = x;
cout << x << endl; → new line command in C++
cout << y;

output: 5 → x is characted
53 → 53 represent 5 in ASCII set
53 → integer value y is integer

cout << (int) x; here x is casted as an integer

cout << (char) y; so we get 53

5

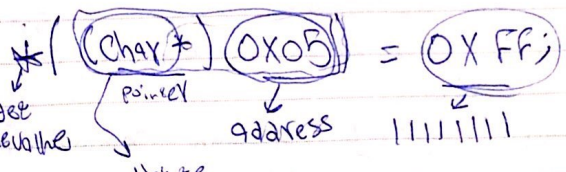
Datasheet of any microcontroller shows register description with name and address

→ 0XFF all set to 1

↓ 8 ones

hex number 08h → we type it → 0X05 → 1 byte

Note to cast it



*((*char*) 0X05) = 100;

PORTA =

* Note we need to write keyword Volatile before Char

ex *((volatile char*) 0X05) = 0XFF;

20621

(int*) 0X20 → 2 bytes ? not for all machines

(long*) 0X20 → 4 bytes

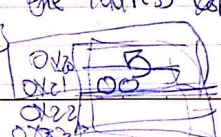
2621622623

0X20 = 5 → Compile error

((Char) 0X20) = 5;

↓ the address of a Char

((int) 0X20) = 5;



← 2 bytes

Scanned with CamScanner

*((long *) 0X20) = 5

Casting *((char*) 0X21) = address → 0

0X20	5
0X21	00
0X22	00
0X23	00

* Volatile keyword

The compiler always tries to optimize the code of the user so it doesn't take a lot of memory.

```
int main(){
  int x=100;
  int y =0;
  while(1){
    while (x<200){
      y++; }
  }
```

→ here the condition is always true because the value of x doesn't change. So the compiler to optimize the code

So the compiler assumes that X is always constant

```
int x=0;
int main(){
  while (1){
    x=PortA
    while (x!=1)
    { y++; }
  }
}
```

→ here PortA always carry x value so compiler assume PortA is constant PORTA carries first value

The compiler takes it one of the loop so it doesn't have to check it again

* to make compiler understand that PORTA and X might change we use Volatile (value)

so we use it with address casting because each hardware component might deliver ... used

/ /

8 means the value can change because of the hardware not software — ((Volatile Char*) 0X0B)
So the compiler checks the value every time

Tip: using directives
#define (PORTA) *((volatile char*) 0X0B)
Text replacement

Note: GPIO.txt has all the address definitions

Defined type
int → size differs from a compiler to another
2, 4, 8
char 1 Byte, 2 Bytes

Note
Compiler
8 bit microcontroller → char → 1 Byte
int → 2 Byte
32 " " → char → 2 Byte
int → 4 byte

8 bit
2 byte ← int x = 100000; // sensor → int = 2 bytes — x = 64.000 2 bytes — limit for int
short int y = 10000; // sensor
int z = x + y;
(100000 X) not allowed can carry less value more than those
on it

32 bit
int x = 100.000; // sensor → x = 100000
int y = 100.00; // sensor
int z = x + y;
4 bytes

difference of int size in each compiler
Can lead to an error.

So we use new data types. :)
using keyword typedef

typedef unsigned char Uint8_t

typedef made by programmer or user

number of bits

main ()
{
unsigned char → unit8_t x = 10;

for 8bit Compiler
{ typedef signed char sint8_t
typedef unsigned int unit16_t
typedef unsigned long int unit32_t

unit32_t x=1000;
unit16_t y=10000
unit32_t z=x+y;

if we transfer the code to 32bit compiler nothing happens because it replace unit32_t with int which is 4 bytes size in the other compiler so no error occurs

Compiler 32bit:
typedef unsigned char unit16_t
typedef unsigned short int unit8_t
typedef unsigned int unit32_t
* Compilers are independant.

* GPIO definition
like we said before Compiler optimizes the codes before turning it into an assembly

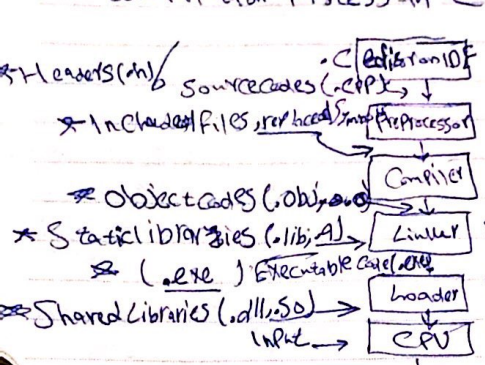
* Note:-
input register is not a constant nor a variable and can be changed by user
volatile prevents optimization of the code
we type PORTA
* define *(volatile char*) 0x05)

Scanned with CamScanner

Compilation Process and Linker Script File :-

IDE : integrated Development environment

Compilation Process in C :-

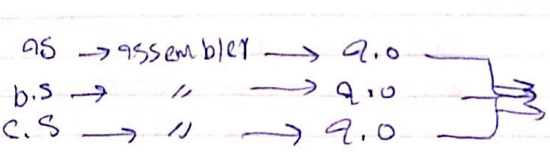


- IDE ~~the~~ has the code written by user output
- #include #define and headers
  ~~that~~ other than that is called source code
- Preprocessor replaces headers with its content
  So we get only source code. [.c]
- Compiler ~~take~~ receives the C file then turn it into Assembly
  => (.obj, .o)
- Assembler take the assembly file then turns it into binary code
  Now see.
  Compiler mainly include both the Compiler and Assembler
- Linker searches for ~~a variable which is~~ an external variable or a function that has been called.
  i.e Links up all the files
- Loader directes the code ~~to~~ over to CPU

Scanned with CamScanner

Linker:-

While writing a multi-file program, each file is assembled individually into object files. The linker combines these files to form the file executable.



Code and data have different runtime requirements. For example code can be placed in read-only memory so that it doesn't change and data require read-write memory it would be convenient might if code and data is not interleaved

- For this purpose, programs are divided into Sections.

Most programs have at least two sections .text for code and .data for data

Assembler directive .text and .data are used to switch back and forth between the two sections.

Sections:-

When the assembler hits a section directive, it puts the code/data following the directive in the selected bucket. Thus the code/data that belong to particular section appear in contiguous locations.

Linker Script file:-

Section merging and replacement is done by the linker. The Programmer can control how the Sections are merged, and at what locations

ALKADIB
Scanned with CamScanner

they are placed in memory through a linker script file.
A very simple linker script file ~~ass~~

all data of files are collected together by ~~linker~~ linker script type
and the same goes for all text
but all data and text must be used in the ~~file~~ combined file. if functions aren't called in ~~main~~ file
Linker script file won't collect them
ex

* Basic Linker Script
Syntax:

```
Sections {①
    . = 0X00000000 ; ②
    .text : { ③
        file → abc.o (.text) ;
        def.o (.text) ;
    }
} ④
```

The beginning adress of the code
wants linker to collect all text in the file

* more pohuy way
wildCard in linker Scripts

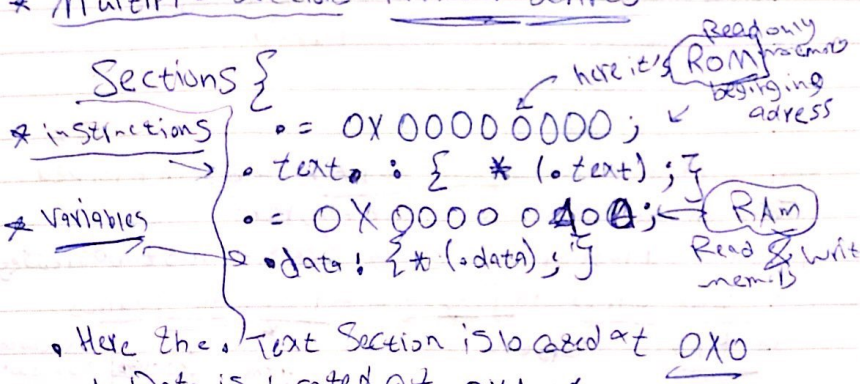
```
Sections {
    . = 0X00000000;
    .text : { * (.text) ;}
}
```

all the text of the file compiled

ALADDIN
Scanned with CamScanner

• if the Program Contains both .text and .data Sections the .Data section merging and location can be specified as shown below

* Multiple Sections in linker Scripts



• Here the .Text Section is located at 0X0 and .Data is located at 0X4000

LOE does the function of the linker Script file

Linker Script File Part 2:

RAM is volatile So it loses the values stored in it once the power is off then after that we will find garbage values instead.

The Code must be stored in non-Volatile memory to save it.

Code is loaded into the flash memory but our Variables Should be stored in RAM So that they can be easily modified

Scanned with CamScanner

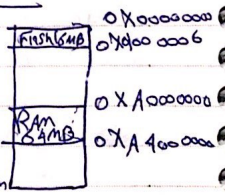
~~All Code and data~~

RAM is volatile memory, and hence it is not possible to directly make the data ~~is~~ available in RAM, on power up

All Code and data should be stored in ROM flash memory before power up.

On power up ~~the~~ a Start up Code is supposed to Copy the data from flash to RAM and then proceed with the execution of the program

, so the program's .data Section has two addresses a load address in flash and a run-time address in RAM ↓ copy only



Save Change linker script file to specify ~~both~~ the load address and the run-time ~~and~~ address for the .data Section

A small piece of Code should Copy the .data Section from flash (load address) to RAM (run-time address)

Syntax Scripting language

```
SECTIONS {
    . = 0X00000000;
    text : { * (.text); } → ROM
    etext = .;
    . = 0XA0000000;
    .data : AT (etext) { * (.data); }
}
```

variable in Copy it Script Language

ROM

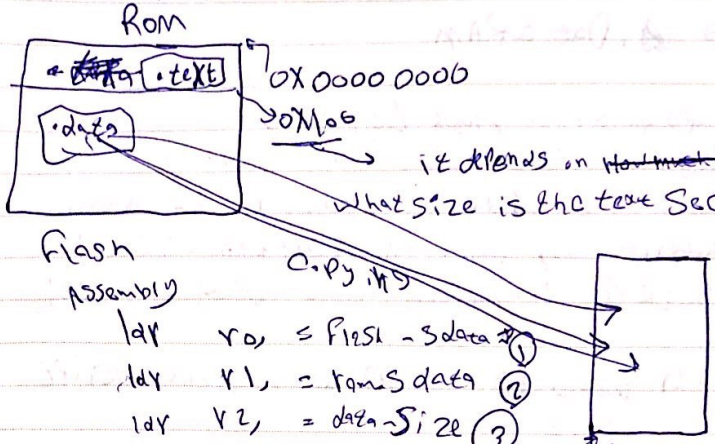
Load address

① address

~~Last~~ First available address after text Section

ROM ← address available

Scanned with CamScanner

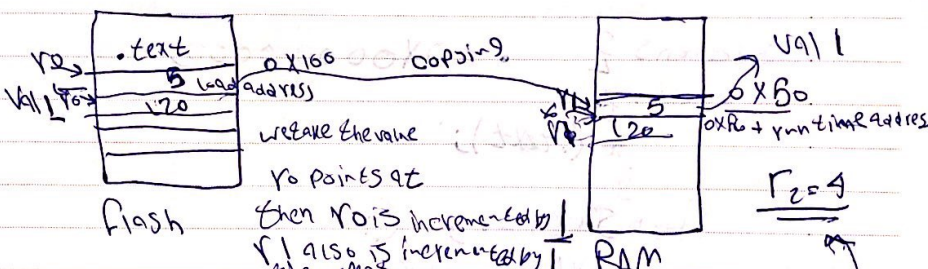


Assembly

```
ldr   r0, = Flash - Sdata   ①
ldr   r1, = ramS data       ②
ldr   r2, = data - Size     ③
```

Copy:

```
ldrb  r4, [r0], #1   address of source  increment address by 1
strb  r4, [r1], #1   store
subs  r2, r2, #1
bne copy
```



Note: variable is accessed in RAM Not ROM.

the process continue till $R_2 = 0$

So the data size is done

then we have Create the Program

~~Compiled~~

Scanned with CamScanner

/ /

Copying Data to RAM

- To copy data from Flash to RAM, the following information is required.
  1- Address of data in flash (Flash_sdata)
  2- Address of data in RAM (ram_sdata)
  3- Size of the data section (data_ssize)

Note: Compiler does that by using Linker script file

- The Linker script can be slightly modified to provide these information.

Linker script with section copy symbols

```
sections {      . = 0x00000000;
  .text : {
      * (.text);
  }
  flash_sdata = .;        (1)
  . = 0xA0000000;
  ram_sdata = .;          (2)
  .data : AT(flash_sdata) {
      * (.data);
  };
  ram_edata = .;          (3)
  data_size = ram_edata - ram_sdata;   (4)
}
```

data end section → ram_edata

ram_sdata ← data start section

Solution: the 3 requirements variables are obtained by Linker

(26) $T_{st} = 100$ N. m   ① tapping = 40%

tapping ② = 80% $X = .8$

$T \propto X^2$ $T \propto I^2$

T_1 100 → $(.4)^2$

T_2 ? → $(.8)^2$

$T_2 = 400$ N. m

• Exception handling and C startup code:-

In the previous code there was a bug starting .text at address 0X0000 0000. because 0X0000 0000 is actually preserved for interrupt service or interrupt table or exception handle in ARM since exception handling is used to deal with unexcepted error

	Exception	Address
• First 8 words in the memory map are reseved for the exception vectors.	Reset → jump to	0X00
• When the exception occurs the control is transferred to one of these 8 locations	Undefined instruction	0X04
	Software Interrupt (SWI)	0X08
	perfect Abort	0X0C
	Data Abort	0X10
	Reserved, not used	0X14
	IRQ	0X18
	FIQ	0X1C

These locations are supposed to contain a branch that will transfer control to the appropriate exception handler

Scanned with CamScanner

Exception handling :-
We add a third Section called VectorsSection

So we add

- text : {
  - *(.Vectors);
  - *(.text);

C-start up

it is not possible to direcely execute C code, when the Processor comes out of reset. since unlike assembly language, C programs needs some basic pre-requisites to be satisfied

Before transferring Control to C code, the following have to be setup Correctly.

1- Stack 2- Global variables:
local variable *initialized
3- Read-only data *uninitialized
Strings & consT

1- Stack :-
String ~~data~~ local Variables, passing function ~~arguments~~, String ~~get~~ return address

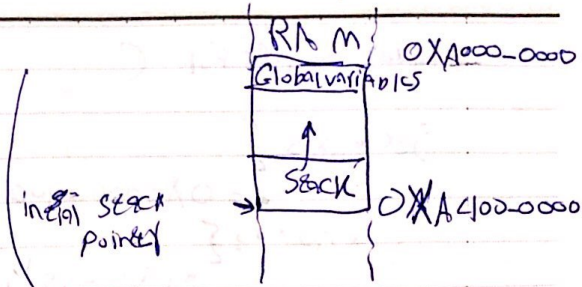
So itmust be Setup Correctly before transferring Control to C code
we must initialize the Stack Pointer to point at the bottom of the Stack

I dMSP = 0XA 400 0000

Scanned with CamScanner

2- Global variables

Compiler stores initialized global variables in data sections and copied from flash to RAM.



in RAM

uninitialized variables are stored in a separate section called .bss is used for uninitialized variables and they will be initialized to zero, in order to save memory.

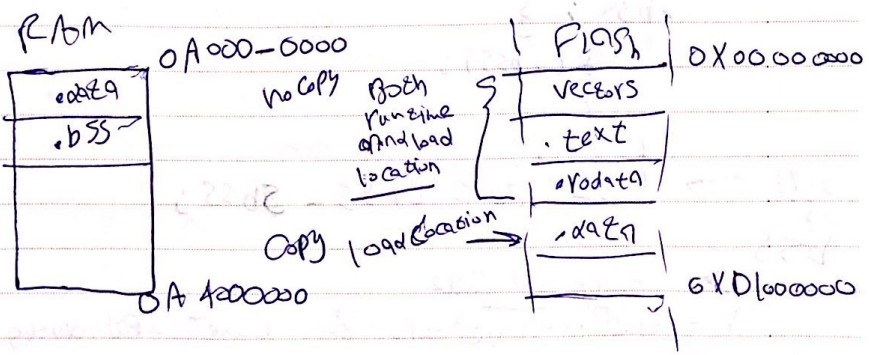
.bss

Read-only Data ..

- Gcc places global variables marked as Const in a separate section called .rodata which is also used to store string Constants.

Flash

- Since Contents of .rodata Section will not be modified, they can be placed in Flash.



Scanned with CamScanner

Linker Script For C

```
Sections {
        . = 0X00000000
    .text : {
            *(vectors);
            *(.text);
    }
    .rodata : {
        *(.rodata);
    }
    flash_sdata = .;
    . = 0XA000 0000;
    ram_sdata = .;
    .data . : AT( flash_sdata) {
        *(.data);
    }
    ram_edata = .;
size of RAM  data_size = ram_edata - ram_sdata;
    sbss = .;
    .bss : {
        *(.bss);
    }
    ebss = .;
size of bss  bss_size = ebss - sbss;
```

- Start up Code
- exception vectors
- Code to copy .data from FLash to RAM
- Code to zero out the .bss
- Code to Setup the Stack Pointer
- Branch to main

Scanned with CamScanner

C typedef

we use it to give a type a new name

```
typedef unsigned char BYTE;

BYTE b1, b2;
```

By convention use _t in the end of the defined type to make the user know that, it's a defined type

ex:-

```
typedef unsigned char Uint8_t
```

Why typedef in C

(1) code portability

Sizeof int is not especified by the C standard so int is not the same for all the machines

thus we use typedef

```
#if def AVR32
    typedef int int_32
# else
    typedef long int_32
# end if
```

(2) Code Readability

here pf is a function returning int (*(*pf())[4])();

a pointer to an array whose size is 4

and contains pointers to function returning an int

using typedef

```
typedef int (*fun)();
typedef int (*fun)();
typedef fun (*PAR fun)[4];
```

Scanned with CamScanner

/ /

fun arr [4] = { test1, t2, t3, t4 };

~~PArr fun PF ()~~

PArr fun PF () pf is function which return PArr fun

{

. Array

PArrfun PFArr = &arr;

return (PFArr);

} ← address

3. Code readability

it give a new name to a complex declaration

~~for a stu~~

for struct and union it helps to avoid

struct key word

struct Student information

{

}; → typedef struct

{

} Student information;

Application of typedef in C

1- use of typedef with Pointers

typedef int* int Ptr; int Ptr Ptr

② for structure and typedef char* char Ptr ;

union

③ ~~for~~ with function point

typedef int (* arrayarithmatics) [4] (int, int);

→ arr of pointers = { Add → ——, —— }

ALADIB

④ With Structure Pointer

typedef struct studentsinfo functions Studentinfo;

Studentinfo * ptr;

⑤ With array

typedef int price [3] → create elements

Price Array [3]

any name

Difference between #define and typedef

typedef: is limited to giving symbol names to types only.

#define can be used to define alias for values aswell (as one, 3.14 as Pi.

typedef: interpretation is preformed by the Compiler

#define statements are Processed by the Pre-Processor

#define ; should not be terminated with a semicolon

typedef: should be terminated with Semicolon.

#define : will just cpy and Paste

typedef: it defines a new type

Scanned with CamScanner

typedef: follows scope rule

#define: no scope rule is followed

void fun()
{
#define one 1;
}
typedef int integer;

the main()
{
printf("%d", ONE); Print 1
integer 0; → error
}

Some types you can only define using typedef only and not #define
typedef unsigned int U_INT32;

Note
typedef char* PTR
PTR a, b, c; char* a, *b, *c;

#define PTR char*
PTR x, y, z; → char *x, y, z;

pointers

int a = 10; at 80f

value can be accessed by using the name or the address

location → value 10 (80f) → name of location

80fp address 80f

types works as filters.

& var → Address

pointers store addresses rather than values

Declaring a Pointer

we should decide the type of data it can point to

the Computer uses data type to determine the memory block the Pointer points to

int a; int takes up memory in 8 bit blocks.

int *myIntPtr

& address operator to get the address of element data type

* access operator (dereference)

its used with any Pointer.

we can use & to get the address of the Pointer.

which is called handle

using *dereference * operator

1- define the pointer

2- assign value to the address

the pointer can

int *ptr

ptr = &a

*ptr = 5;

ALADIB

Scanned with CamScanner

int X = 1000;

int *P = X → 1000 → address not value

int *q = (int*) X ; q is pointer that

int *ptr = &X address no. 1000

~~Why we don't~~

Common mistake when working with pointers

int C, *PC;

PC is address but C is not.

PC = C; // error

*PC = &C; // error &C is address but PC is not

PC = &C; correct

*PC = C correct

~~Why~~ we don't get an error when using int *P = &C;?

Because it's declaration of pointer and we assign value to it.

int *P = &C;

it's better if we do the 2 step method

int *P;

P = &C;

Benefits of Pointer

1- more efficient in handling Arrays and structures

2- they allow references to function and thereby helps in passing of function as arguments to other functions.

3- it reduces length of the program and execution time as well

4 allows C to support dynamic memory management

- you can perform arithmetic operations on pointers just as you can on a numeric value

like increment and decrement

when you ++ or -- a pointer it will always increment by the number of bytes occupied by the type it points to. char or int or double

if p is pointer to some element of array then p++ increments p to point to the next element

or p+n to -

four arithmetic operations possible on pointers

++ -- + - and -

note: if we have pointer to int, adding 2 to the pointer will add 2*2 = 4 to the address it contains

ALADIB

Scanned with CamScanner

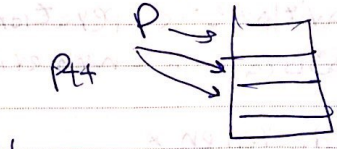
this is because int variables occupy 2 bytes.

This operation will move the pointer to the next memory location without impacting the actual value at the memory location.

This mechanism makes pointer arithmetic simple and uniform for all types, and hides all details of type-size from the programmer.

---

eg
pointer to a float → 4 bytes



(+) size of data type

Loops & Jumps increment by the size of data type

comparisions
pointers may be compared
such as ==, <, and >

if the two pointers P1 & P2 point to variables that are related to each other such as the same array
P1, P2 can be meaningfully compared

ALADIN
Scanned with CamScanner

Pointer to Struct

```
Struct Point { int x;
               int y;
             };
```

```
struct Point *p;
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

p->x : access of pointer

Pointer to function

ex

```
doub (*pf) (double, int);
```

we can pass function as an argument

The declaration of a pointer must specify the number and types of the function arguments and the function return type.

Note

→ `int (*pf)(char)` pointer to function in which char output int

→ `int *f(char)` function in which char, return pointer to int

`fun`

`add = &fun`

`int (*ptr) (int, int)`

we can use `ptr` = not necessarily `&ptr`

Note:- like array's function's names are automatically converted to pointers without using the address of operator &.

Scanned with CamScanner

Pointer to Struct

```
struct point { int x;
               int y;
             };
```

```
struct point *p;
printf("x is %d\n", (*p).x);
printf("x is %d\n", p->x);
```

p->x : access operator

Pointer to function

ex

```
doub (*pf) (double, int);
```

we can pass function as an argument

The declaration of a pointer must specify the number and type of the function arguments and the function return type.

note

→ `int (*pf)(char)` pointer to function in which char and int

→ `int *f(char)` function in which return pointer to int

```
fun
add = &fun
```

`int (*ptr) (int, int)`

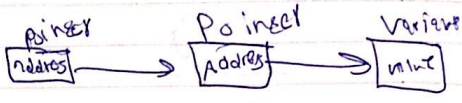
we can use `ptr` = not necessarily `&ptr`

note... like array's functions names are automatically converted to pointers without using the address of operator &.

Scanned with CamScanner

Pointer to Pointer

* It's a chain of pointers or multiple indirection



A variable that is pointer to pointer must be declared it's done by placing an additional asterik infront of its name

ex. int ** var;

```
int var;
int *ptr;
int **pptr;
var = 300
ptr = &var;
pptr = &ptr;
```

to get var printed
printf ("%d", var);
// , *ptr);
// , **pptr);

* pointers to const and const pointers to const values

declares ptr a pointer to const int type.

const int * ptr; → int is const
const ↓ value is const

you can modify ptr itself but the object pointed by ptr shall not be modified

int * const ptr → points at the same place const

~~that int can be changed~~

Scanned with CamScanner

~~int const ptr~~ int * const ptr
declare *ptr a const pointer to int type

you are not allowed to modify ptr right the object pointed to by ptr can be modified

ex

```
Const int a = 10;
Const int * ptr = & a;
* ptr = 5 ; // wrong
ptr++; // right
```

value a is const.

```
int a = 10
int* const ptr = & a;
*ptr = 5 ; // right
ptr++; // wrong
```

→ int const * ptr // ptr is a pointer to constant int

— int * const ptr // ptr is a constant pointer to int

Call by Refrence using value passing the address as arguments

fun(&x, &y)

```
fun (int* ptr1, int *ptr2)
{
}
```

ALADDIN

Scanned with CamScanner