

ET-1

## Function pointer or pointer to function

It stores the ~~addr~~ of function.

It provides runtime binding in C program which resolves many problem.

### How to declare function pointer

function return type (~~\*functionName~~) (Fn Arglist)

R.g:

Void (\*display) (void)



or pointer to fn.

Here display is a function pointer

which takes void argument and returns void.

Void \*display (void)

It is the declaration of fn display which takes void argument and returns void pointer

Note: A function pointer must have the same signature for the fn. that it is pointing to.

### Initialization of function pointer

This is initialized to the add of a fn but the signature of the fn pointer should be the same as the function

The function name also represents the beginning add of the fn.

### Declaration of fn pointer

int add(int, int)

(2)

int (\*pfadd) (int, int)

Initialization of pfadd

pfadd = &add

or

pfadd = add

OR

Declaration and initialization at one go.

int (\*pfadd) (int, int) = add

or

int (\*pfadd) (int, int) = &add

Calling a fn using fn pointer

(\*funcPtnName) (Argument list)

or

(function pointer Name) (Argument list)

e.g. to call add function

int add(int a, int b)

{  
    int c = a+b;

    return c;

}

pointer to add

(int) int (\*pfadd) (int, int) = NULL

pfadd = &add

pfadd(10, 20); or

(\*pfadd)(10, 20);

③ memory allocation and deallocation for fn ptr  
↳ dynamic mem allocation is not safe  
for fn ptr. if we allocate dynamic mem to fn  
ptr. then there's no importance to make fn ptr

\* Note: when we compare two fn ptr. then we  
have to remember that two pointers of the  
some type compare equal if they are both  
null. or both point to the same fn. or  
both represent the same add.

### typedef for function pointer

typedef int (\*pfun

typedef int (\*funptr) (int, int);

passing function pointer as an argument

int add (int a, int b, funptr calculate)

Here calculate is  
a fun pr which takes  
two int argument and  
int and return int)

)

int main()

↳ funptr ptrToSum = &sum;

int result = add(10, 20, ptrToSum);

Return a funptr from the function

typedef int (\*funptr) (int, int);

(4)

## Funptr Asilt Operation (int a)

4

funptr sum = &~~oddSum~~; show;

?  
return sum;

✓ Array of function pointers: we can create an array of function pointer

### Complicated Declaration

Rule

(i) Declaration

(ii) () []

(iii) \*

(i) char \* foo [5]

↑  
foo is an array of 5 pointer to char.

(ii) char (\* foo) [5]

foo is a pointer to an array of 5 characters

(iii) char \* (\* foo) (char)

foo is a pointer to a function which accepts ~~char~~ a pointer to char and return a pointer to char.

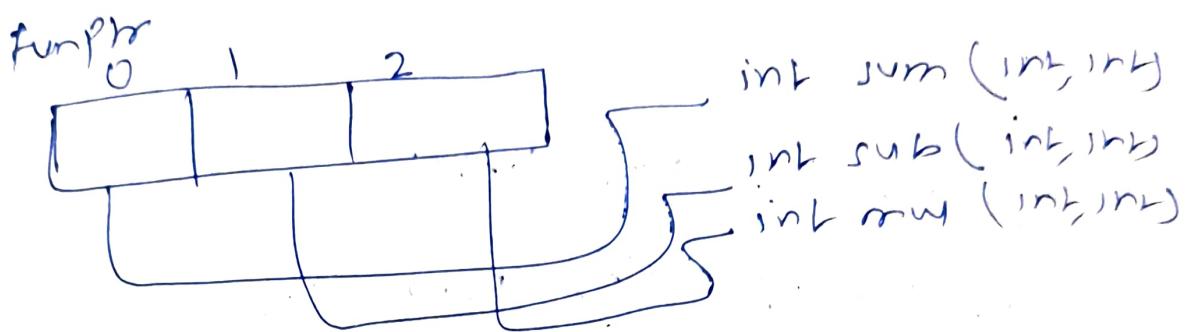
(iv) int (\* funPtr [3]) (int, int);

funptr is an array of 3 pointers to function which accepts 2 integer and return int.

Q) ⑤  $\text{char} * \underset{+}{c} * (\underset{\text{+}}{\ast} \text{foo}[5]) \underset{\text{+}}{(\text{char} *)} [ ];$

foo is an array of 5 pointers to function which takes pointer to char and returns char or pointer to an array of pointers to a char.

ex:  $\boxed{\text{int } (\ast \text{funptr}[3])(\text{int}, \text{int})}$



Initialising an array of function pointers

$\text{int } (\ast \text{ArrOfFunPtr}[3])(\text{int}, \text{int});$

ArrOfFunptr[0] = sum;

ArrOfFunptr[1] = sub;

ArrOfFunptr[2] = mul;

} OR

$\boxed{\text{int } (\ast \text{ArrOfFunPtr}[3])(\text{int}, \text{int}) = \{ \text{sum}, \text{sub}, \text{mul} \};}$

Calling function through array or funptr

$\boxed{\text{int result} = (\ast \cdot \text{ArrOfFunPtr}[0])(10, 20);}$

$\boxed{\text{int result2} = (\ast \text{ArrOfFunptr})(20, 15);}$

int result = (x An often prob [2]) (30, 40); (6)

Typecasting ~~void~~ NULL using function pr

void \* PvHandle = NULL;

~~int (\*pf)(int) = (int (\*)(int)(int))~~

~~int (\*pf)(int) = (int (\*)(int)) pHandle;~~

~~(int (\*)(int)) pHandle~~

using typecast

typedef int (\*pf)(int);

pf Newpf = (pf) pHandle.

function pointer in structure:

\* C does not contain

✓ C structure does not contain fn member  
function like off.. In C long we can not  
create ~~structure~~ function in structure. But  
using fun ptr. we can provide these features  
and these function ptr will treat like  
structure and can also support poly-  
morphism in C.

Adv: ✓ Fun pointer provides runtime binding.  
using function ptr we can create generic fun.  
that perform the opr as per user choice.

(7) ~~typedef~~ → It reduces the complexity  
~~It defines a new name for existing type and does not introduce a new type~~  
↓  
~~increase code readability and portability.~~

### Syntax

typedef oldtype Newtype

typedef unsigned int UnsignedInt  
it creates a synonym for or new name for existing types

- why we use typedef: when size of int is not specified
- (i) code portability:
  - (ii) code Readability:
  - (iii) code clarity:

scope for typedef: → block scope, file scope

typedef with structures:

```
struct stud
{
    int Roll;
    int Class;
    char name[20];
};
```

struct stud s1; ⇒ variable s1 is of type struct.

Now:

(i) of typedef struct stud Student

Now Student s1 ⇒ some or above

(8)

OR

typedef struct stud

{ int

OR

typedef struct stud

```

    {
        int Roll;
        int class;
        char name[20];
    } Student;

```

(2)

Now student s1;

Declaration of pointer for struct stud:typedef struct stud \* S PTR;OR

(3)

typedef struct

```

    {
        int Roll;
        int class;
        char name[20];
    } Student;

```

typedef with structure definition

Now student s1;

S†

~~Note~~ we should use structure tag at the time of structure declaration. bcoz if we do not use a structure tag with structure then we will get a compiler error when structure tries to reference itself.

(7) typedet struct

h. studentinfo into array student list  
int roll;  
int class;  
} studentinfo;

with structure tag

typedet struct stud;

h. char name[20];  
struct stud \*sinfo;  
studentinfo \*sinfo;  
int roll;  
int class;  
} studentinfo;

typedet with structure pointer:

typed struct stud

h. char name[20];  
int roll;  
int class;  
struct stud \*plist;  
} StudInfo, \*pStudent;

StudInfo, \*pStudent;

Now StudInfo Ram.  $\Rightarrow$  some on

char stud Ram.

pStudent Mori  $\Rightarrow$  some on

struct stud \* Mori;

⑩ typedef with array ~~int~~ increases the readability

typedef int Brick\_Price[3];

Brick-Price price  
price is an array of 3 int

Brick-Price \*pBrick-Price  $\Rightarrow$  is same as

int (\*pBrick-Price)[3]; // pointer to an array  
of 3 int

typedef with 2D array

typedef int iodata[2][2];

idata ip; // Here ip is a 2D array of  
size 2x2

idata \*pIp // is a pointer to 2D array

typedef with array of function pointer

typedef int (\*ArrOfFunPrt[3])(int, int)

ArrOfFunPrt atp = {add, sub, mult};

✓ typedef for typecast

void \*pHandle = NULL;

int (\*px)(int) = (int (\*) (int)).pHandle  
using typecast

(11) `typedef int(PF)(int);`

`PF arg = (PF) pHandle.`

typedef with enum:

```
typedef enum
{
    NoError = 0,
    ReadError,
    WriteError,
    LogError
} errorList;
```

`errorList flagState; // creating enum variable`

Advantage of typedef over #define:

✓ It follows scope rule

File scope, block scope.

Compiler

✓ Give new symbolic names to existing type

Does not follow scope.

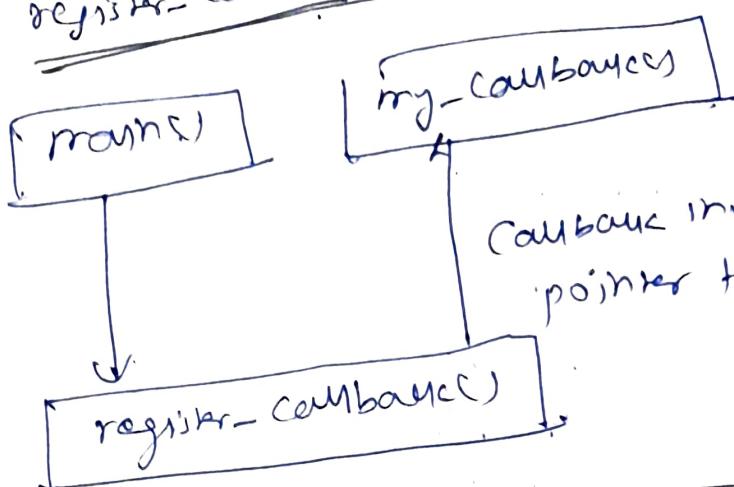
Preprocessor

Used to create alias of any type and value

Callback in C:

\* A callback is any executable code that you pass as an argument to other code which is expected to callback (execute) the argument at given time. i.e not of a function is passed as an argument to another function to call it, then it will be called as callback function.

(12) Register a callback which is just pointing at  
pointer as an argument to some other function.  
Eg: register\_callback.



Ex: callback.c typedef void (\*callback\_ptr)();  
~~#include <reg\_callback.h>~~  
typedef  
void my\_callback(void);

{  
 printf("my\_callback");  
}

int main(void)

{  
 my\_callback \*ptr\_my\_callback = my\_callback;  
 callback\_ptr ptr\_to\_my\_callback = my\_callback;

~~Register our callback function~~

register\_callback(ptr\_my\_callback);

register\_callback.h

typedef void (\*callback\_ptr)();

void register\_callback(callback\_ptr, ptr\_my\_callback);

// reg\_callback.h (Registration goes here)

(13) void register\_callback( callback\_ptr ptr\_reg\_callback)

\*ptr-reg-callback();

use of callback: callback can be used to create a library that will be called from an upper layer pgm and in turns the library will call user defined code on the occurrence of some event.

enum in C

✓ user defined datatype and it consists of set of named constants integer: i.e. mapping of integer value to some name (readability).

enum follows the scope rule and the compiler automatically assigns value to its mem constants.

Note: variable of enum type stores one of the values of the enumeration list defined by that type.

Syntax: enum Enumeration- Tag { Enumeration - List; }

enumeration type more

Ex, separated named constants

enum FLASH\_ERROR { DEF\_ERROR, BUS\_ERROR}

- (14) Note: (i) If we do not initialize the member of the enum list then the compiler automatically assigns the value to each member of the list in increasing order.
- (ii) By default compiler always assigns 0 to the first const. memb. of the enumeration list.

example:

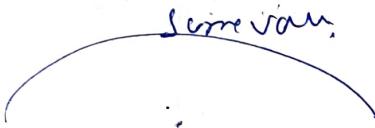
enum Days { mon, Tue, wed, Thu, Fri, Sat, Sun }  
0 1 2 3 4 5 6

enum Days eDay = Tue

Note: (iii) the set of enumeration constant may contain no duplicate value i.e. two constant member may have the same value.

Ex:

enum Days { sun=0, mon=2, Tue=0 }



(iv) enum follows the scope rule: i.e. in two different block we can define enum with the same name and even with same name content also.

(v) we can also declare an unnamed enumerator data type; which means that we can omit the enumeration tag.

enum { sun=0, mon=2 } Days;

Days = sun

(15)  In enumeration, every identifier of the list or variable so should have unique Name within the scope.

enum Days1 { Sun, mon, Tue, y; }  
enum Days2 { Sun, wed, Thurs };  
invalid in the same scope.

An enumeration tag also follows the scope rules. So enumeration tag should be different from the tag of structure, union or the enumeration type.

like: enum ERROR\_LIST { LOW, ERROR, FLAG }  
struct ERROR\_LIST  
  
{ } ; ————— IT invalid.

we can assign the value to the enumeration constant in any order; unassigned constant get the value from the previous constant + 1

enum Days { mon=5, sun=0, Tue, wed,  
Thurs=55, fri=60 };

so mon=5 ← Thurs=55  
sun=0  
Tue=1  
wed=2  
fri=60

(b) The value assigned to the enum member should be integral constant and within the range of integer.

### Important points

- ① enum in C ~~define new type~~, but the macro does not define new type.
- ② enum follow scope rule.
- ③ enum in C type is an integer but the macro type can be any type.

### typedef with enum

typedef enum Days

Sun,  
Mon,  
Tue,  
Wed,  
Thu,  
Fri,  
Sat

Weekdays

OR

Sun,  
Mon, Tue, Wed,  
Thu, Fri, Sat

Weekdays

### Storage class in C

- ↓
  - Decides the extent(lifetime) and scope(visiblity) of the variable in the pgm.
  - Helps us to trace the existence of a specific variable during the running of the pgm.

for storage class

(17) auto: default storage class, local variable (non static)

scope:

lifetime:

storag.

default value.

static: It makes a variable or function private for the file in which it declares.

scope: if we use static keyword with no ~~global~~ variable then the scope of the global variable is limited to the file in which it declares.

~~\* static variable preserves its previous value and it is initialized at compile time when mem is allocated. not initialize static variable get initialized by compiler to 0.~~

~~\* If we have initialized static variable then it will be created in data segment (dd) otherwise it will created in (bss) (block started by symbol of the program~~

Ex. Calc.c

static int count=0 // stored in (ds) and scope to Calc.c

\* void add()

{ static int l=0; // ds and scope to add fn

static int k // bss and scope to add fn

?

scope:

(18)

~~lifetime: till the whole execution of pgm.~~

storage:

Default value: 0

~~extern: → It only declares a variable and is used by giving reference of the global variable that is accessed by files of pgm.~~

It says to compiler that the variable is defined elsewhere in the pgm, if only points to the already defined variable in pgm.

File1.c

```
int g Data;
```

File2.c

```
printf("gData") ; Error
```

File3.c

```
extern int g Data;
```

```
printf(" gData"); ✓
```

scope:

~~lifetime: global variable lifetime through out the execution of pgm~~

storage: It initialized here in (eds)

otherwise loss

Default value: 0

⑯ register: we cannot use (& and \*) with register variable.

register int data;

Memory layout of C pgm

### ① Stack

⑩ stack frame will be created in the stack where fr is called.

⑪ each fn has its own stack frame

⑫ sp (stack pointer)  
Registers

### ⑬ Heap

malloc, calloc,  
realloc

→ stored by all  
shared libraries and

dynamically loaded modules in process

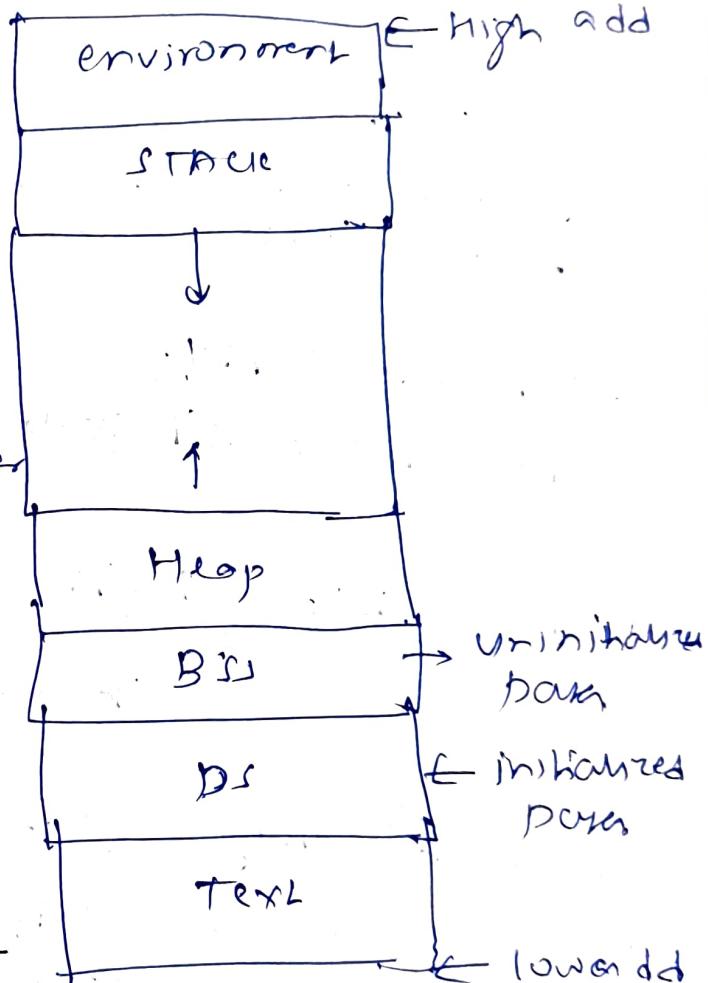
### ⑭ BSS (Block started by symbol)

→ contains all uninitialized global and static variable

→ all variable in this seg. initialized to 0 and pointer by NULL

→ pgm loader allocates mem for

BSS



IV) Data segment (20)  
contain explicitly initialized & global  
and static variable.

V) Text: contains binary of the compiled program.  
Executable.

classified in two types

- ✓ initialized read-only area.
- ✗ initialized read-write

#include < stdio.h >

char str[] = "Rajkumar"; // read/write  
Data segm.

char str = "Keyboard"  
library string

\* constant string i.e. literal goes to read-only  
data segment. this is a type of constant global  
data goes to this segment.

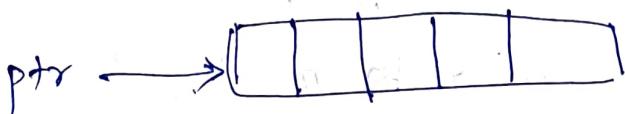
Dynamic memory allocation  
malloc, calloc, realloc

not destroyed by compiler

Note: A.T. C standard, if the size for the space  
required is 0. The behaviour is implementation  
defined. error in null pointer is reported or  
behaviour is as if the size were non-zero  
value except from the returned pointer  
should not be used to access an object.

② ① malloc  
↓  
memory alloc  
char \*piBuffer = malloc(5 \* sizeof(char));  
or

char \*ptr = malloc(5);



free(ptr) // Here ptr will become dangling.

if: ptr=NULL;

② calloc  
↓  
c for continuous alloc  
void \* calloc(size\_t noOfblocks,  
size\_t objectSize)

int \*ptr = (int \*) calloc(4, sizeof(int));

↓ Here space is initialized to 0.

③ re-alloc:  
↓  
void \* realloc(void \*ptr,  
size\_t size)

dynamically re allocate

int \*ptr = (int \*) calloc(5, sizeof(int));

ptr = realloc(ptr, 10 \* sizeof(int));

\* Disadv: mem fragmentation and overhead.

→ if ptr=NULL then  
realloc behaves like malloc.

(17) void free(void \*ptr)

If ptr is NULL

(22)

free tree doesn't

do anything

\* callloc slower than malloc.

Is it better to use malloc() or callloc();

✓ callloc initializes the mem. with 0.

callloc is similar to malloc + memset.

ptr = callloc(m, sizeof(int));

ptr = malloc(n \* sizeof(int));

memset(ptr, 0, (n \* sizeof(int)));

Ques What's the return value of malloc(0)?

✓ It is implementation defined. It could be null pointer or it shows the behaviour like that size is some non-zero value.

### Structure in C

~~struct~~

#### Declaration

struct stud

{ int Roll;

int age;

char name[20];

}

struct stud s1, s2; // definition

or

struct stud

{ int Roll;

int age;

(23)

char name[20];

? s1, s2; // s1, and s2 are definition  
s1 and s2 are variable or struct

struct type

OR

Struct

h. int roll;

int age;

char name[20];

? s1, s2; // this is also defn of s1 and s2

~~Note:~~ A structure does not contain a mem with incomplete or function type (except the flexible array) that is why at the time of structure declaration, it cannot contain the instance of itself but contains a pointer to itself.

Struct myData

h. int a;

Struct myData; // illegal

? data;

Struct myData

h. int a;

Struct myData \*ptr; // valid

? data

Struct myData

{ int a;

}; int b; }

// valid in C99 and C11

struct myData

(24)

{ int a=0 // 1st element }

} { b; }

Note: gcc permits a structure to have no members.

struct myData

{ }

} ;

Note: we can not initialize the members of the structure at the time of structure declaration  
beoz there is no memory allocated to the member at the time of declaration

struct myData

{ }

int a=0 // 1st element

{ }

int b=0 // 2nd element

struct myData

{ }

int a;

{ }

int b; // 2nd element

Initialization of structures

struct myData info={2,3};

or struct myData info;

info.a=2;

info.b=3

Designated Initialization: means we can

initialize the members of the structure in any order using dot(.) and member name

(.member-name);

(25)

struct myData data = { .b=30, .a=.20 }

or

struct myData data = { .a=30, .b=40 };

Accessing structure member with variable and pointer:

struct Emp

```
L struct Emp
{ int eid;
  int salary;
};
```

or

struct myData data = { b: 2, a: 3 };

struct Emp e1, \*ep;

e1.eid = 20

e1.salary = 20000

ep = (struct Emp\*) malloc ( sizeof( struct Emp ) );

ep->eid = 30

ep->salary = 40000

or

(\*ep).salary

How to calculate size of structure in C

An using sizeof operator or own created macro or using function.

Anonymous Structure in C

A structure or union with no tag.

introduced in C11 but not in C99