(ET-1)

function pointer or pointer to function

↓

It stores the add of function.

It provides run time binding in C program which resolves many problem.

How to declare function pointer

```
function.returntype (*functionPtrName) (fnArgList);
```

e.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 return void pointer

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

Initialization of function pointer

↓↓

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

Function name also represents the beginning add of the fn.

Declaration of fn pointer

```
int add(int, int);
```

```
int (*pfadd)(int, int);
```

eg.

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

```
(*funcPtrName)(ArgList);
```

or

```
(functionpointerName)(Argument_list)
```

e.g. to call add function

```
int add(int a, int b)
{
    int c = a+b;
    return c;
}
```

pointer to add

```
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 useful for fn ptr. if we allocate dynamic mem to fn ptr. then there is no importance to create fn ptr.

* Note: when we compare two fn ptr. then we have to remember that two pointers of the same type compare equal if they are both null. or both points 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 ~~same~~

```
int add (int a, int b, funptr Calculate)
```

Here Calculate is a fun ptr which takes two int argument and as 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);
```

funptr ArithOperation(int o)

{

funptr sum = &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 a pointer to char and return a pointer to char.

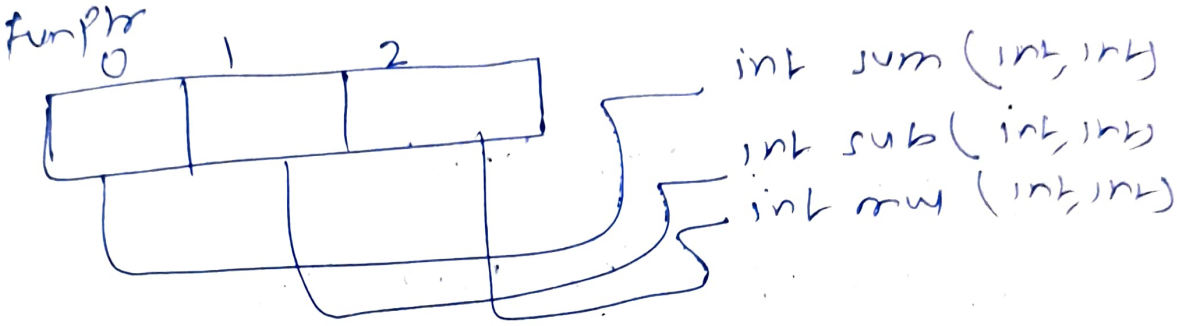
(IV) int (*funPtr[3])(int, int);

funPtr is an array of 3 pointers to function which accepts 2 integer and returns int.

(N) (5) char *(*(*foo[5])(char*))[];

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

ex: int (*funPtr[3])(int, int);



Initializing an array of function pointers

int (*ArrOfFunPtr[3])(int, int);

ArrOfFunPtr[0] = sum;
ArrOfFunPtr[1] = sub;
ArrOfFunPtr[2] = mul;

— OR

int (*ArrOfFunPtr[3])(int, int) = {add, sub, mul};

Calling function through array of FunPtr

int result = (*ArrOfFunPtr[0])(10, 20);
int result2 = (*ArrOfFunPtr)(20, 15);

int result2 = (*ArrOfFun ptr[2])(30, 40); ⑥

Typecasting void NULL using function ptr

```
void *pvHandle = NULL;
```

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

int (*pf)(int) = (int (*)(int)) pvHandle

(int (*)(int)) pvHandle.

using typeCast

```
typedef int (*pf)(int);
```

```
pf Newpf = (pf) pvHandle.
```

function pointer in structure:

- ~~C does not contain~~
- C structure does not contain the member function like C++. In C lang we cannot 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 polymorphism in C.

Adv: ① Func pointer provides runtime binding using function ptr we can create generic fun. that perform the opn as per user choice.

(7)

typedef → It reduces the complexity

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 Unsigned;

it creates a synonym for or new name for existing types

why we use typedef

(i) code portability: when size of int is not specified

(ii) code Readibility:

(iii) code clarity:

scope for typedef: → block scope, File scope

typedef with structure:

```
struct stude
{
    int Roll;
    int class;
    char name[20];
};
```

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

Now:

(1) typedef struct stud Student

Now Student s1 ⇒ same as above

OR

typedef struct stud

{ int

OR

(2)

```
typedef struct stud
{   int Roll;
    int class;
    char name[20];
} Student;
```

Now Student S1;

Declaration of point for struct stud:

typedef struct stud * SPTR;

OR

(3)

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

→ typedef with structure definition

~~typedef with~~

Now Student S1;

S1

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) typedef struct

{
char name[20];
struct studentinfo *info; ✗ error shows incomplete
int roll;
int class; no structure tag
} studentinfo;

with structure tag

typedef struct stud

{
char name[20];
struct stud *sinfo; ✓
~~Studentinfo~~ *sinfo;
int roll;
int class;
} studentinfo;

typedef with structure pointer:

typed struct stud

{
char name[20];
int roll;
int class;
struct stud *pslist;
} StudInfo, *pStudInfo;

Now StudInfo Ram. => some on

struct stud Ram;

pStudInfo Hari => some on

struct stud *Hari;

(10) typedef with arrays increases the readability

```
typedef int Brick_Price[3];
```

Brick_Price price

price is an array of 3 int

Brick_price *paPrice ⇒ is same as

```
int (*paPrice)[3]; //pointer to an array of 3 int
```

typedef with 2D array

```
typedef int iaData[2][2];
```

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

iaData *pIp // is a pointer to 2D array

typedef with array of function pointer

```
typedef int (*ArrOfFunPtr[3])(int, int)
```

ArrOfFunPtr afp = { add, sub, null };

typedef for typecast

```
void *pvHandle = NULL;
int (*pF)(int) = (int (*)(int)) pvHandle
```

using typecast

(11) typedef int(*pf)(int);

pf arp = (pf) pvhandle.

typedef with enum:

```
typedef enum
{
    NoError = 00,
    ReadError,
    WriteError,
    LogError
} errorList;
```

errorList flagState // creating enum variable

Advantage of typedef over #define:

- follow scope rule
- File scope, block scope.
- compiler
- give new symbolic name to existing type

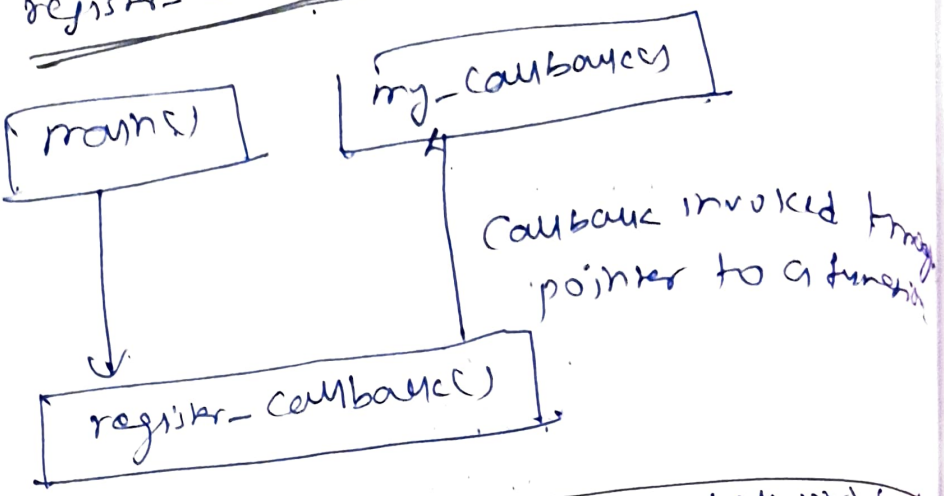
- Do not follow scope.
- preprocessor
- used to create alias of any type and value

Callback in C.

* A callback is any executable code that is passed as an argument to other code which is expected to call back (execute) the argument at given time. i.e a ref 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 passing a function 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)

{

callback_ptr = 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)

(23) 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 turn the library will call user defined code on the occurrence of some event.

enum in C

- a user defined data type 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 assign 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 Name

{}, separated named constants

enum FLASH_ERROR { DEF_ERROR, BUS_ERROR }

(14) Note: (i) If we donot initialize the members of the enum list then the compiler automatically assign 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, Thrys, Fri, Sat };
            0   1    2    3      4    5
enum Days eDay = Tue
```

Note:: (i) The set of enumeration constant may contain a duplicate value. i,e two constants member may have the same value.

ex.

```
enum Days { Sun=0, Mon=2, Tue=0 };
```
(Sun=0 and Tue=0 have same value)

(ii) enum follows the scope rule: i,e in two different block we can define enum tag with the same name and even with same name constant also

(iii) we can also declare an unnamed enumerator data type, which means that we can omit the enumration tag.

```
enum { Sun=0, Mon=20, } Days;
```

Days = Sun

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

```
{
    enum Day { Sun, Mon, Tue};
    enum Day2 { Sun, Wed, Thrus};
}
```

Sun invalid in the same scope.

(v) 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
{ . .                    // Invalid.
  . .
};
```

(vi) we can assign the value to the enumeration constant in any order, unassigned constant get the value from the previous constant +1

```
enum Day { mon = 5, sun = 0, Tue, wed,
           Thrus = 55, fri = 60};
```

So mon = 5
sun = 0
Tue = 1
wed = 2
Thrus = 55
Fri = 60

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

Important points

(i) enum in C define new type. but the macro does not define new type.

(ii) enum follow scope rule.

(iii) 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,
    Thurs,
    Fri,
    Sat
} WeekDays;
```

OR

```
typedef enum
{
    Sun,
    Mon, Tue, Wed,
    Thurs, Fri, Sat
} WeekDays;
```

Storage class in C

↓

decides the extent(lifetime) and scope(visibility) of the variable in the pgm.

helps us to trace the existence of a specific variable during the runtime of the pgm

(17) Four storage class

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

scope:
lifetime:
storage:
default value.

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

scope:

→ if we use static keyword with the global variable then the scope of the global variable is limited to the file in which it declared

* 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 (ds) 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 i=0; // .ds and scope
                    to add fn
    static int k // .bss and
                 scope to add fn
}
```

scope:

lifetime: till the whole execution of pgm

storage:

Default value: 0

extern: → It only declare a variable and is used to give the reference of the global variable that is accessed by files of pgm.

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

File1.c

```
int gdata;
```

File2.c

```
printf("gdata") //error
```

File3.c

```
extern int gdata;
printf("gdata");
```

scope:

lifetime: global variable lifetime through out the execution of pgm

storage: It initialized then in (ds)

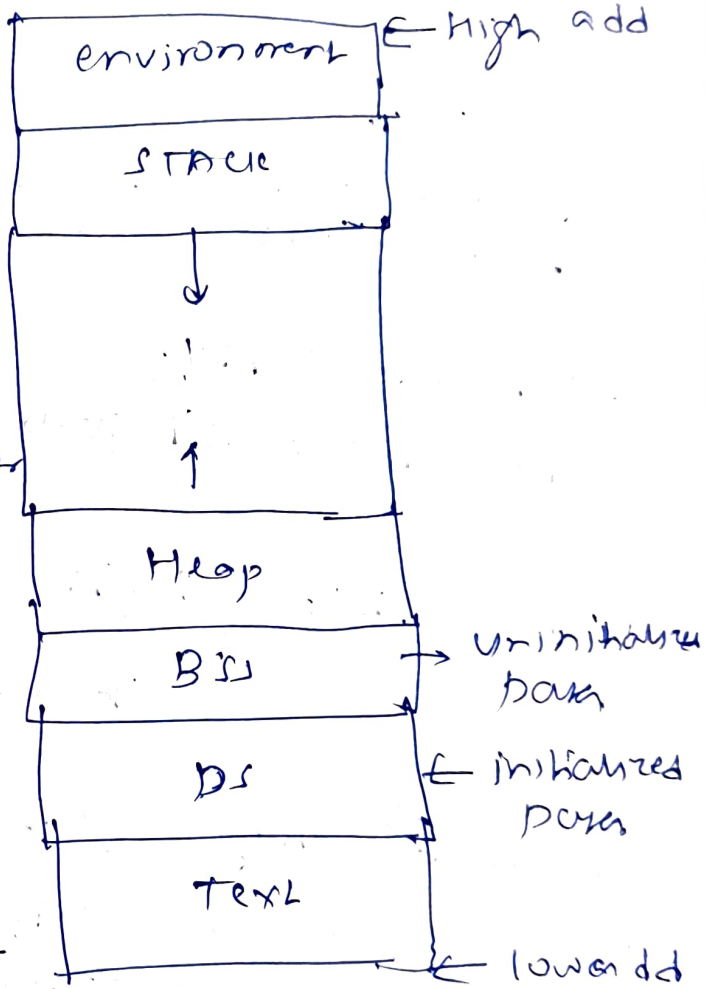
otherwise in bss

Default value: 0

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

register int data;

Memory layout of C pgm



(I) Stack

(i) Stack frame will be created in the stack when fn is called.

(ii) each fn has its own stack frame

(iii) sp (stack pointer Register

(II) Heap:

malloc, calloc, realloc

stored by all stored libraries and dynamically loaded modules in process

(III) 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:
- contain explicitly initialized global and static variable.

(v) Text: contain Binary of the compiled pgm.
- storable.

classified in two types
- initialized read only area.
- initialized read-write "

```
#include <stdio.h>
char str[] = "Rajkalash"; // read write
                          Data segment
```

~~char~~ *str1 = "Rajbhar";
literal string

* constant string i/e literal goes to read only data segment. also all types of constant global data go to this segment.

Dynamic memory allocation
malloc, calloc, realloc ||
Not destroyed by compiler

Note: A.T. C standard, if the size of the space requested is 0. The behaviour is implementation defined. either a null pointer is returned or behaviour is as if the size were non-zero value except that the returned pointer shall not be used to access an object.

(2) (i) malloc

memory alloc

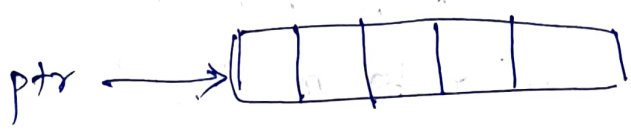
void * malloc(size_t size);

char *piBuffer = malloc(5 * sizeof(char));

(ii)

or

char * ptr = malloc(5);



free(ptr) // Here ptr will become dangling

ptr = NULL;

(ii) calloc

c for contiguous alloc

void * calloc(size_t noOfBlocks, size_t Object_size)

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

↓ Here space is initialized to 0.

(iii) re-alloc:

↓

dynamically re allocate

void * realloc(void *ptr, size_t size)

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

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

* Disadv: mem fragmentation and mem leak.

if ptr = NULL then realloc behave like malloc.

(iv) void free(void *ptr)

if ptr is NULL then free does nothing

* calloc is slower than malloc.

Is it better to use malloc() or calloc()?

✓ calloc initializes the mem. with 0.

calloc is similar to malloc + memset.

```
ptr = calloc(n, sizeof(int));

ptr = malloc(n*sizeof(int));
memset(ptr, 0, (n*sizeof(int)));
```

Qu 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;
```

char name[20];

} s1, s2; // s1 and s2 are definition
s1 and s2 are variable of struct
stud type

OR

struct
{ 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 contain a pointer to itself.

struct myData
{ int a;
struct myData; // illegal
} data;

struct myData
{ int a;
struct myData *ptr; // valid
} data

struct myData
{ int a;
int b[]; // valid in cag and C11
};

```
struct mypara
{
    int b; // illegal
} d;
```

Note: gcc permits a structure to have no members

```
struct mypara
{
};
```

Note: we cannot initialize the members of the structure at the time of structure declaration because there is no memory allocated to the member at the time of declaration

```
struct mypara
{
    int a=0; // illegal
    int b=0;
}
```

```
struct mypara
{
    int a;
    int b; // valid
}
```

Initialization of structure:

```
struct mypara into={2,3};
```

```
struct mypara into;
into.a=2;
into.b=3
```

Designated initialization: means we can initialize the member of the structure in any order using dot(.) and member name (.member_name);

(25) struct mypoint data = {.b=30, .a=20}

or

struct mypoint data1 = {.a=30 .b=40};

Accessing structure member with variable and pointer:

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

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

OR

```
struct mypoint data =
{2, 3};
```

How to calculate sizeof structure in C

By 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