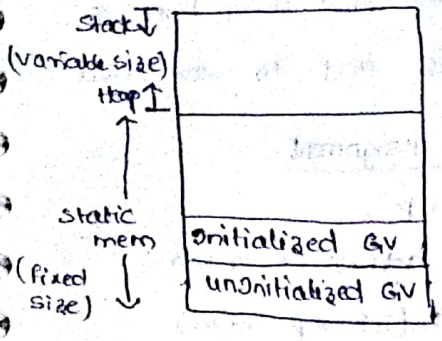
Programming languages

Memory organization of C-program:



Stack memory
- → local var (auto)
- → formal parameters
- → function calls
- → Access links (static links)
- → Temporaries
- → Return address.

*) Stack allocation will be done at runtime

Static memory
- → Static variables
- → String Constants
- → scalars (constants)
- → function code
- → Global var are stored in static mem

*) static allocation will be done at Compile time

Heap memory

*) Allocated during runtime

*) Allocated through malloc(), calloc(), realloc()

*) deallocated by free()

*) For all dynamic data structures, Heap memory is allocated.

Storage Classes: It describes storage area, default value, scope and lifetime

SDSL

Automatic storage classes

Auto
- → Stack memory
- → default value is Garbage
- → function scope
- → function life time.

*) usage of keyword is optional

*) Every local var is treated as auto variable

Ex:- (auto) int i;

Register
- → register memory
- → default value is Garbage
- → function scope
- → function lifetime

*) We cannot Access addr of reg var, so pointers are not used

*) The keyword register request compiler to store the var in Reg rather than RAM if available

Ex:- register int i;

Static storage classes

Static
- → static memory
- → default value is 0
- → function scope
- → program life time

Ex:- static int i;

Global (External var)
- → static memory
- → default value is 0
- → program scope
- → program life time.

*) Declared outside of fn blocks

*) If there is any name conflict between local and Global then compiler will give preference to local variable

Extern: (dummy stmt)
- → No physical memory is allocated at run time
- → Used at Compile time to resolve unknown var
- → It can be declared as Global.
- → It may lead to linking error.
- → If Extern var initialized with a value then Physical memory is allocated.

Ex:- Extern int a = 5;

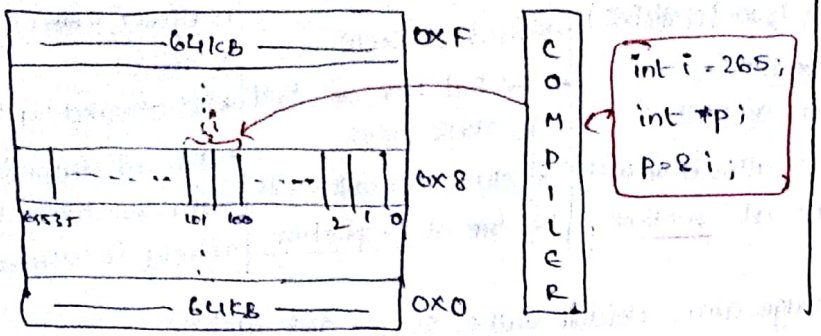
A)
```
void main()
{
register int i;
scanf("%d", &i);
}
```
Compile time error (CTE)
Segmentation fault

```
void main()
{
register int i;
scanf("%d", i);
}
```
CTE
Segmentation fault

```
void main()
{
int i=10;
register int j=i;
}
```
CTE
~~Segmentation fault~~
We can't copy -

Pointers : (datatype *pointer_var)

→ In Turbo C 3.0 by default every pgm occupies 1MB of memory which is a collection of 16 segments and each segment of size 64KB and every byte is identified by a unique location num known as its address and to store that address we need pointer variable.

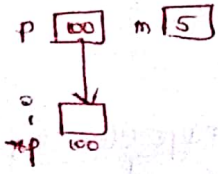


Pointer assignment

q = p ;

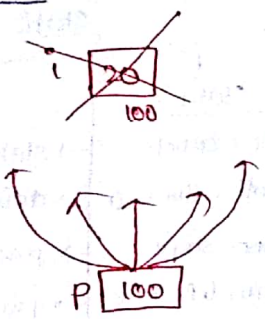
Passing address of a var :

```
void fn(int *p, int m)
{
}
main()
{
  fn(&i, 5)
```



Dangling pointer problem :

```
int *g()
{
  int i=20
  return &i;
}
void main()
{
  int *p;
  p=g();
  pf("%d", *p);
}
```



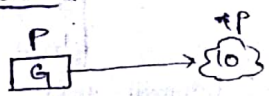
*) The pointer p is initially pointing to some mem location where object i is available but after some time the object is removed but still pointer is pointing to same mem loc

↓ sol

- Increasing life time by making var as GV
- Placing static keyword (or) bcoz they have pgm lifetime.

Uninitialized pointer problem :

```
Void main()
{
  int *p;
  *p = 10;
}
```

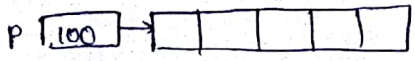


*) In this prob pointer p is not initialized with any valid address

Dynamic memory Allocation :

malloc()

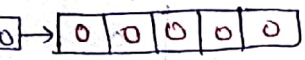
→ int *p ;
P = (int*) malloc (5 * sizeof (int))



→ Initialized with Garbage value
→ Return type is (void *)
→ Created in heap memory

Calloc()

→ int *p
P = (int*) calloc (5, sizeof(int));



→ Initialized with zero's.
→ Return type is (void *)
→ Created in heap memory

*) int *g()
{
int *px;
px = (int *) malloc (sizeof(int));
*px = 10;
return px;
}
void main()
{
int *p
for (i=1; i<=n; i++)
{
p = g();
Pf("%d", *p);
free(p);
}
}

*) On this pgm the memory allocated by malloc which is no longer used is not released

↓

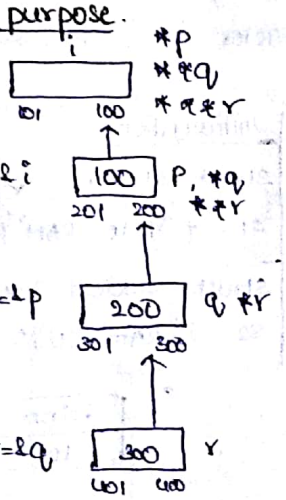
Memory leakage problem

↓ sol

Use free(p) for every value of i

Pointer to pointer :

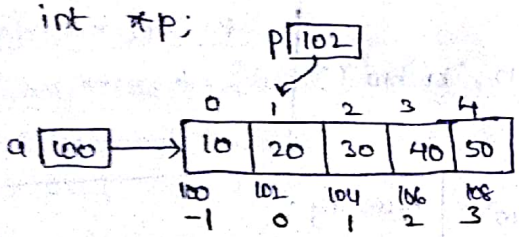
Declaration	Datatype	purpose
int i	int	*p, **q, ***r
int *p	int *	p = &i
int **q	int **	q = &p
int ***r	int ***	r = &q



Pointer to Array :

Array name is a constant pointer which always contains base address of the array and we cannot change the content of the array

*) int a[5] = {10, 20, 30, 40, 50};
int *p;



a = a+1 ⇒ Compile time Error
L-value reqd.

p = a+1
⇒ 100 + (1 * size of DT)
→ 100 + 1*2 → 102
P = 102.

*) Pf("%u %u", a, &a[0]);
100 100

*) subscript rule.
a[i] ⇒ *(a+i)
⇒ *(a+(i*size of DT))

*) a[i] = *(a+i)
= *(i+a)
= i[a]

Points to remember :

$x[i][j] = *(x+i)[j] = *(*(x+i)+j)$

(i) $*(*(x+i)+j)$ ⇒ value of i^{th} ID, j^{th} index

(ii) $*(x+i)+j$ ⇒ address of i^{th} ID, j^{th} index

(iii) $*(x+i)$ ⇒ address of i^{th} ID, 0^{th} index.

(iv) $x+i$ ⇒ Base addr of i^{th} ID

(v) x ⇒ Base addr of Array / Base addr of 0^{th} ID / Base addr of 0^{th} ID, 0^{th} index.

*) int *p : p is a pointer to an int DT

*) int (*p)[3] (Pointer to Array) :- p is a pointer to the array of 3 elements of int DT

*) int *p[3] (Array of pointers) :- p is a Array of 3 pointers of int DT.

Pointer to string :

String Constant : Group of chars enclosed in double quotes is called string constant

→ Every string constant is stored in static memory

→ It always returns base address of the string constant

→ It is always terminated by NULL character

Pointer to structures :

Structure declaration:

```
struct student
{
  int idno;
  char s[6];
};
```

No physical mem is allocated

Structure variable:

struct student s1; (student : DT)

Physical mem is allocated.

Initialization :

struct student s1 = {10, "RAM"}; ✓

struct student s2 = {"RAM", 10}; ✗

Accessing

s1.idno ⇒ 10

s1.s ⇒ RAM

s1

idno	s
10	'R' 'A' 'M' '\0' '\0' '\0'

Alternate way of defining structure

```
struct student
{
  int idno;
  char s[6];
} s1 = {10, "Ram"};
```

Array of structure :

- struct student a[3] = {10, "Gudipati", 11, "sai", 12, "krishna"};
- struct student *p;

*) P = a+1

P = a + (1 * size of stud)

= 100 + 1*11

= 111

Student

	10 "Gudipati"	11 "sai"	12 "krishna"
a 100 →	idno s	idno s	idno s
	100	111	

p 111

Accessing :

a[0].idno → 10

a[0].s → Gudipati

P → idno ⇒ 11

(*p).idno ⇒ 11,

Nesting of structure :

```
struct date
{
  int dd,mm,yy;
};
struct emp
{
  int idno;
  struct date dob;
  char s[5];
} e1 = {10, 04, 12, 1991, "sai"};
```

e1	10	04 12 1991	"sai"
	idno	dd mm yy dob	s

Accessing :

- e1.idno → 10
- e1.dob.dd → 04
- e1.dob.mm → 12
- e1.dob.yy → 1991
- e1.s → sai

Points to remember :

1) *++p → c : (1), (2)
2) (*p)++ → c : (1) → (2)
3) (p++) → c : (1) p → c (2) ++(1)
4) p++ → c : (1) p → c (2) ++(1)

Pointer to function : (To implement call back value efficiently)

Declaration : ret type (*pointer_var) (list of args);

Ex: void (*p)(); → p is a pointer to a function where function takes no args and return type is void.

void (*p)(int,int); → p is a pointer to a function where function takes 2 integer args and return type is void.

Assignment :

pointer_var = fun_name;
(or)
pointer_var = &fun_name;

Accessing :

pointer_var (list of args);
(or)
(*pointer_var)(list of args);

Array of function pointers :

Declaration : int (*p[5])(int,int);

Assignment : p[0] = &fun_name (or) p[0] = fun_name;

Accessing : p[0](int,int) (or) (*p[0])(int,int);

Static scope and Dynamic scope :

Static scope (local + visible Ancestor)

Referencing Environment of a stmt in a statically scoped language is the collection of all local var + all other ancestor variables which are visible in the stmt.

Dynamic scope : (Local + Active sub pgm)

Referencing Environment of a stmt in a dynamically scoped language is the collection of all local var + all other active sub program variables which are visible in the stmt.

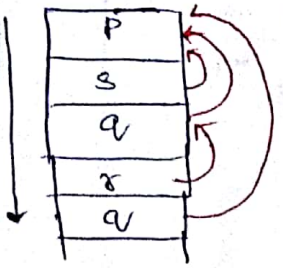
Active sub program :

A sub pgm is called active if its execution has started but not yet terminated.

Rules for creating access link :

Every Activation record in the stack must have Access link to the most recent activation record which lexically contains it.

- P lexically contains q, r, s
  Q lexically contains r



Why Access link

→ To Access variables.

Parameter passing Techniques :

(i) Call by value :

- Actual parameters carry their value and copied into formal parameters

→ later formal parameters can be modified but their modification are not reflected in calling function variables.

(ii) Call by Reference :

- Actual parameters are carrying their references and formal parameters are acting as alias to their corresponding actual parameter.

→ If formal parameters are modified then it reflects in calling function variables

Note :

Possible Combinations : Static scope + Call by value
Dynamic scope + Call by reference

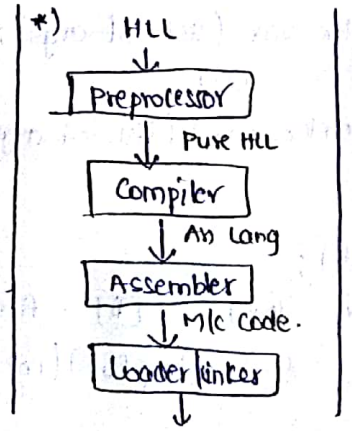
Points to remember :

To Execute a pgm in linux :

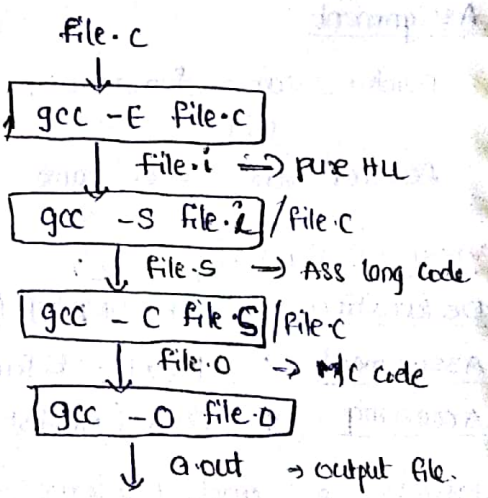
```
$ vi file.c
$ gcc file.c
$ ./a.out
```
} Commands

$ hello world. → output

*)



*



Format specifiers :

%.6.2f ⇒ Print as floating point, atleast 6 chars wide and 2 after decimal point

%.p ⇒ pointer address
%.x ⇒ int unsigned hex value
%.u → int unsigned decimal
%.o → int unsigned octal value.

Enumerator : ∵ readability ↑

enum boolean { NO, YES }
(0, 1)

enum months { Jan=1, feb, mar, apr - - - }
(1, 2, 3, 4)

Constants :

Const float pi = 3.14 ⇒ Pi value doesn't changes throughout the Execution of the program

left and right shift bitwise operator:

A (60) : 0011 1100

A<<2 = 1111 0000 (240) → $60 * 2^2$

A>>2 = 0000 1111 (15) → $60 * 2^{-2}$

Precedence table: (BÜAS RBLTİSC)

	Operators			Associativity
①	( ) , [ ] , → , .		(Brackets)	L to R
②	! ~ ++ -- + - * & (type)		size of (unary)	Right to left
③	* / %	}	Arithmetic	L to R
④	+ -	}		L to R
⑤	<<, >>		- shift	L to R
⑥	<, <= > >=	}	Relational operator Comparision.	L to R
⑦	== !=	}		L to R
⑧	&	AND	Address	L to R
⑨	^	XOR	Bit wise operator	L to R
⑩	|	OR	}	L to R
⑪	&&	Logical AND		L to R
⑫	||	Logical OR		L to R
⑬	? :	Ternary		Right to left
⑭	= += -= *= /= %= &= ^= |= <<= >>=		Short hand operator	Right to left
⑮	,	Comma.		L to R

- Brackets
- Unary operator
- Arithmetic
- shift
- Relational
- Bitwise
- logical
- Ternary
- shorthand
- Comma.

Realloc : (void *realloc (void *ptr, size_t size)
20B

20B → avail → allocate extra 10B
20B → not avail → Copy the 10B content and store it in another 20B memory

× —— The End × ——×