Data Structures & Program Design in C

Date No

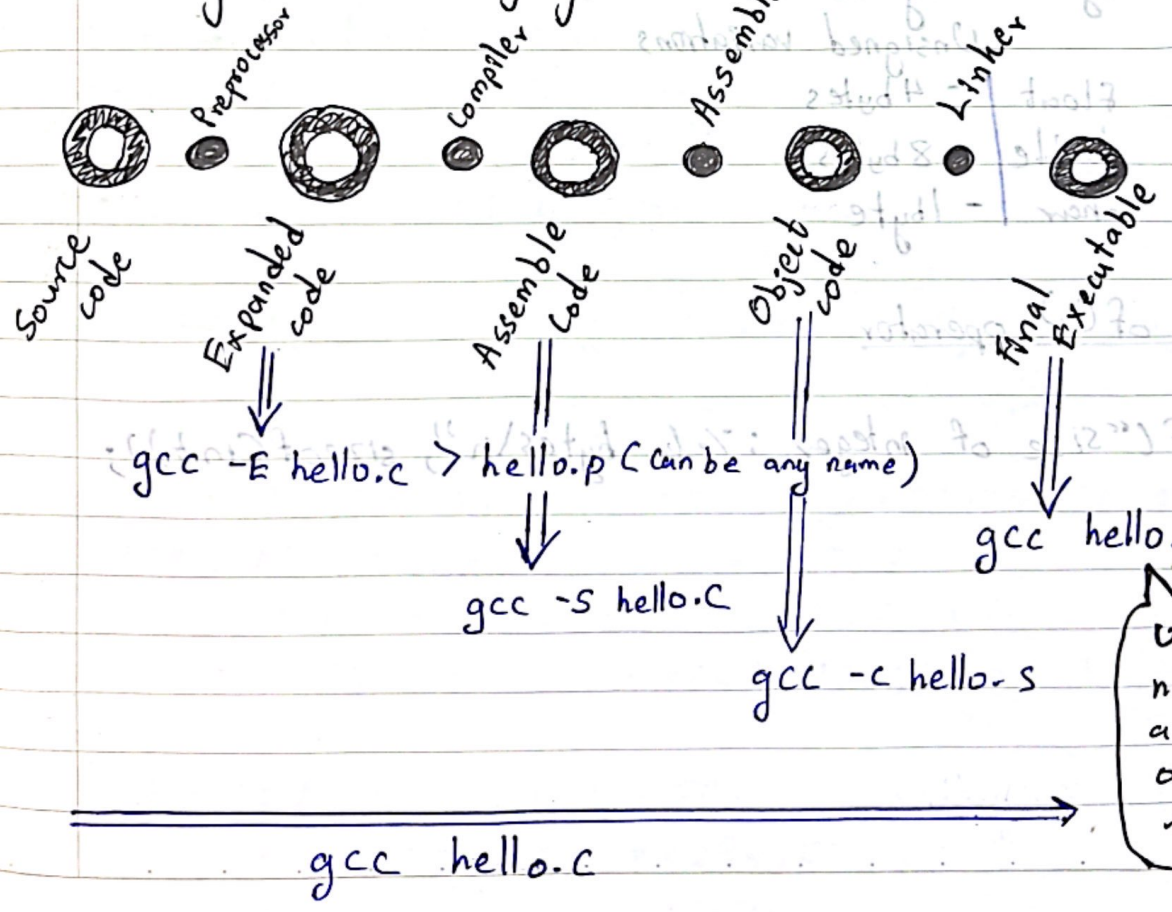
1/ 1st generation languages: Machine language (low level language)

2/ 2nd generation languages :- Contains human readable notation.
- Converted to ML using an assembler

3/ 3rd generation languages :- Contains series of English like words that humans can understand easily.
- Converted to ML using translator or interpreter

4/ 4th generation languages :- Higher level than 3rd gen
- Used in more specific fields such as data and database manipulations.
- More efficient than 3GL

5/ 5th generation languages



Date No

Comments: // This is a comment — * Ignored by the compiler when the executable is being created.

/* This is a comment */

Variables name in C,

- can contain letters, digits & underscores
- must begin with a letter or underscore.
- case sensitive
- no white spaces & special characters
- cannot be a keyword.

Variable types,

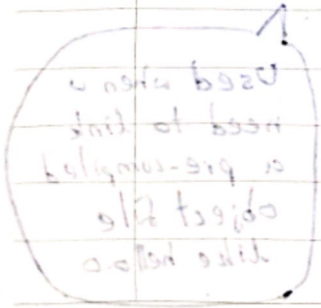
Primitive: int (4 bytes) : short int (- 2 bytes) | Long int (- 8 bytes)

Unsigned variations

float	- 4 bytes
double	- 8 bytes
char	- 1 byte

size of() operator

```
printf("size of integer : %lu bytes\n", sizeof(int));
```



ProMate

%[flags][width][.precision][length]specifier

Date No

COMMON FORMAT SPECIFIERS

1/	%d	int	Signed decimal integer
2/	%i	int	Same a %.d, signed decimal integer
3/	%u	unsigned int	Unsigned decimal integer
4/	%f	float/double	Decimal floating-point (6 digit by default)
5/	%F	float/double	Same as %.f, but capitalized for INF/NAN
6/	%e	float/double	Scientific notation (eg: 1.23e+03)
7/	%E	float/double	Scientific notation with E (eg: 1.23E+03)
8/	%g	float/double	Uses %f or %e depending on value & precision
9/	%G	float/double	Like %.g, but uses %.F/%.E
10/	%x	unsigned int	Hexadecimal (lowercase)
11/	%X	unsigned int	Hexadecimal (uppercase)
12/	%o	unsigned int	Octal
13/	%c	char	Single character
14/	%s	char*	String (null-terminated)
15/	%p	void*	Pointer address

Date No

LENGTH MODIFIERS

1/	hh	signed char/unsigned char	%hhd, %hhu, %hho, %hhx
2/	h	short/unsigned short	%hd, %hu, %hx
3/	l	long/unsigned long	%ld, %lu, %f
4/	ll	long long/Unsigned long	%lld, %llu
5/	L	long double	%LF, %Le, %Lg
6/	z	size_t	%zu, %zd
7/	j	intmax_t/uintmax_t	%jd, %ju
8/	t	ptrdiff_t	%td, %tu

SPECIAL FLAGS (for printf())

1/	-	Left justify in field	42
2/	+	Force (+) for positive numbers	+42
3/	<space>	space before positive numbers	42 -42
4/	0	Pad with zeros	
5/	#	Alternate formatting (prefix, decimal point)	

(Alternate form)

Applies to,

- %o → Prefix with 0 | printf("%#o\n", 10); 012
- %x / %X → Prefix with 0x / 0X | printf("%#x\n", 255); 0xff
- %f, %g, %e → Include decimal always | printf("%#g\n", 3.0); 3.00000

WIDTH & PRECISION

1/ %5d — at least 5 characters wide

2/ %.2f — 2 digits after decimal point

3/ %8.3f — 8 characters wide, 3 after decimal.

4/ %*.*f — width and precision specified at runtime (use with args)

SCANF-SPECIFIC FORMAT SPECIFIERS

1/ %d — Read a signed decimal integer

2/ %i — Read an integer (detects base automatically: 0 for octal, 0x for hex)

3/ %u — Read an unsigned decimal integer

4/ %f — Read a float

5/ %lf — Read a double (matters in scanf, unlike in printf (lf))

6/ %Lf — Read a long double

Date No

7/ %c Read a character (does not skip whitespace)

8/ %s Read a string until whitespace

9/ %[...] Character set - read only characters in the bracket

10/ %[^...] Inverse character set - read until any character in brackets appears.

11/ %p Read a pointer address

12/ %% Match a literal % in input

ESCAPE SEQUENCES (for outputs)

1/ \n : next line
2/ \t : adds tab space
3/ \a : produce a sound or a visible alert
4/ \\ \" : Print \ or double quotes "

Char : • Display one character from ASCII code (128 characters) (7bits)
• Nowadays there are more than 128 due to extended ASCII.

* 2 ways to assign characters,

1/ char a = 'A'; (use single quotes) (double quotes for strings)

2/ char a = 65; (using ASCII decimal value)

OPERATORS

Operator (+, =, -, *, /)
Operands (Thing which the operator works on)

Date No

1/ '=' represents assignment operator

<L.H.S> = <R.H.S>

ex: int a = 5;

2/ "==" used to check equivalence. (Relational operator)
True (1) or False (0)

takes 1 byte

ie: In C all numbers except 0 is true.

Arithmetic Operators

BINARY OPERATORS (Needs 2 operands)

(+), (-), (*), (/), (%)

* % works only for int types. (Modulo)

ex: int a = 17, b = 5 a % b = 2

UNARY OPERATORS (Needs only 1 operand)

Increment (++), Decrement (--), Unary plus (+), Unary Minus (-

a++ ← post setting ex: int a = 5; Output a = 6 b = 5
int b = a++ ie: b = a++ : b = a
a = a + 1

++a ← pre setting ex: int a = 5 Output a = 6 b = 6
int b = ++a ie: b = ++a : a = a + 1
b = a

+ Same for --a & a--

++a++ ✗ ERROR

Integer division: When 2 numbers are divided, only the whole number is displayed.

$17/5$

ex: int = int/int ⇒ 3

double = int/int ⇒ 3.000000

double = double/int = int/double ⇒ 3.400000

double = double/double ⇒ 3.400000

int = double/double ⇒ 3

double c = 17/5 ⇒ 3

double c = 17.0/5.0 ⇒ 3.4

RELATIONAL OPERATORS (True (1) or False (0))

(>), (>=), (<), (<=), (==), (!=)

LOGICAL OPERATORS

AND (&&), OR (||), NOT (!) ← Unary logical operator

Relational Operators ⟹ (Outcome) Logical truth

Logical Operators ⟹ (Works on) Logical truth

i.e: NAND & NOR universal operators. (Functionally Complete)
(can do any logical operation using only NAND or NOR)

BIT MANIPULATION (works only for char & int types)

* Used to ~~manupt~~ manipulate large numbers by breaking them up.

1) & - bitwise AND: int a=6, b=4

```
00110 ← 6
00100 ← 4
-----
00100 → (4)
```

6 & 4 | OUTPUT | 4

2) | - bitwise OR: int a=6, b=4

```
00110 ← 6
00100 ← 4
-----
00110 → (6)
```

6 | 4 | OUTPUT | 6

3) << - leftshift: 6<<2

```
16 8 4 2 1
 0 1 1 0  → 6
11 0 0 0  → 24
```

| OUTPUT | 24

4) >> - rightshift: 6>>2

```
8 4 2 1
0 1 1 0 → 6
0 0 0 1 → 1
```

| OUTPUT | 1

5) ^ - bitwise XOR: int a=6, b=4

```
00110 ← 6
00100 ← 4
-----
00010 → (2)
```

6 ^ 4 | OUTPUT | 2

6) ~ - complement (unary operator): ~6

* If 6 was named unsigned then its 1s complement.

```
00110 → 6
11001 → (-7)
```

| OUTPUT | -7

2s complement due to signed

OPERATOR PRECEDENCE

Date No

		Associativity
1/	Postfix (C) [] → . ++ --)	
2/	Unary (+ - ! ~ ++ -- (type)* & size of) /Prefix	Right to left
3/	Multiplicative (* / %)	
4/	Additive (+ -)	
5/	Shift << >>	
6/	Relational < <= >>=	
7/	Equality == !=	
8/	Bitwise AND &	
9/	Bitwise XOR ^	
10/	Bitwise OR |	
11/	Logical AND &&	
12/	Logical OR ||	
13/	Conditional ?:	Right to left
14/	Assignment	Right to left
15/	Comma	

Example 1: `int x=4, y=5, z=2;`
`int result = x++ * --y + z++ / x;`

= (x++)* (--y) + (z++) / x

(x++): 4 → 5, x = x+1, done after the operation

(--y): 4

(z++): 2

= 4 * 4 + 2 / 5

= 16 + 2 / 5

= 16

Example 2: int a = 3, b = 4, c = 2;
int result = ++a * b-- + c++ - a;

= (++a) * (b--) + (c++) - a
(3+1=4, 4, 2, 4)

= 4 * 4 + 2 - 4
= 16 - 2
= 14

Example 3: int x = 5, y = 10, z = 4;
int result = x++ & y | z++ ^ --x;

= (x++) & y | (z++) ^ (--x)
(5, 10, 4, 5-1=5)

= 5 & 10 | 4 ^ 5
= 0101 & 1010 = 0000 ; 100 ^ 101 = 001

= 0 | 1
= 1

CONSTANTS

- Read only variables. Value cannot be changed once declared.

2 ways to define constants

ie: Variable are by convention named in full capitals.

1/ const <data_type> <var_name> = value;

2/ #define <const_name> <value>

- Defined out side the main method
- Replaces the constant name with the data in the preprocessor step.
- Doesn't have to consider data types.

Date No

- Difference: 2) defines faster
  2) doesn't consider variable type, it just replaces.
  ∴ it sacrifices data type information.

Enumerations (Different from array index) 4 bytes.

- lists that contain constant integer values.

enum <enum_name> {<options>};

- list of values has to be unique within the scope

Example: enum Grade {A, B, C, D};

```
int main()
{
  enum Grade T=A;
  printf("%d\n", T);
}
```

OUTPUT

0 (position of A in the enumerator).

*To print actually "A" you need pointers.

2) if {A=50, B, C, D}
Then B=51, C=52, y=53

3) if {A, B=50, C, D}
Then A=0, C=51, D=52

- You can also set different values to all the elements in the list.

4) if {A=50, B=50, C=50, D}
Then A=50, B=50, C=50, D=51

Date No

TYPEDEF

- Typedef keyword is used to define an alias for existing data types and enumerations.

typedef <data_type/enum> <new_name>;

Example : typedef enum Grade{A, B=50, C=50, D=50}mark;
typedef int VALUE;

```
int main()
{
    VALUE a = 45;
    mark T = C;
    mark X = A;
    printf("%d %d\n", X, T);
}
```

OUTPUT

50 0

TYPE CONVERSIONS (type casting)

- When the operator has operands of different types, they are converted to common types.

2 ways,

1/ Automatic :- Narrow operands converted to wide ones without loosing information.
ex:- int → float, float → double

not prohibited or illegal

2/ Explicit :- Wider operands converted to narrow operands.
<type-name> <expression>
- Generates a compiler warning
- Require explicit type conversion

ex: double a = 68.7;
int b = (int) a;

ex: double c = (double) a/b
a = 3
b = 2
* this happens first.
* So a turns to a double.

Date No

Importance:- Type safety
- Enhancing code readability
- Improved data manipulation

Problems:- Loss of precision
- Overflow or Underflow
- Unexpected behavior

CONTROL FLOW

- Statement: line of code commanding to do a specific task. (terminated by ;)
- Block: Set of statements when executed is syntatically equal to a single statement. (eg: {...}) (No terminating character)

IF-ELSE... (make decisions in the program flow)

```
if (<expression>)
{
  // true
}
else                    * else is optional
{
  // false
}
```

SWITCH

- Multiway decision that tests whether an expression matches one of the defined constant integer variables.

switch (<expression>)
{

Fall through behaviour

* Once one case matches, every other line case below it gets executed.

* Default case is executed when no cases match. (not compulsory to have a default case).
* Default can also be on the top too no strict order.
(Fall through happens through default as well.

JUMP STATEMENTS

1/ break
2/ continue
3/ goto
4/ return

Implicit jump
* Based on a condition you perform a jump

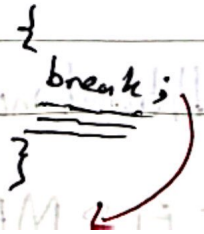
Explicit jump
* Explicitly give the command to jump.

BREAK STATEMENT

- Terminates the immediately enclosing loop block or switch block. (control passes to the statement immediately after the enclosing block)

* If there is no enclosing block, compiler will give an error message.

Basically goes out of the curly brackets.



TERNARY OPERATOR

<condition> ? <if-true> : <if-false>

* Can be easily used to assign values to variables, but the main issue is readability.

```
ex: int a = 26;
    int b = 0;
    if ((b = (a>25) ? 10 : 5) == 5)
    {
      printf("T\n");
    }
    else
    {
      printf("F\n");
    }
```

(b = 5)

Iterative STATEMENTS (loops)

* All loops have 3 components,
1/ Loop condition
2/ Loop counter *defined outside of the loop (exception for FOR LOOPS)
3/ Loop Body

1/ WHILE LOOP

```
while (<condition>)
{ ....
}
```

2/ FOR LOOP

```
for (<initialization>; <condition>; <counter>)
{ ....
}
```

3/ DO-WHILE LOOP

```
do
{ ....
}while (<condition>);
```

* even if the condition is not met, do-while executes at least once regardless.

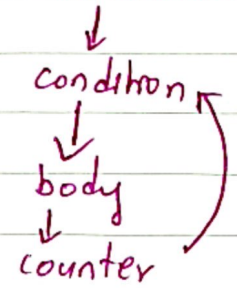
Terminates through a semi colon.

```
ex: for (int a=5; a>=0; a--)
    {
      printf ("A\n");  <= LOGIC
    }
```

① ② ③ ④ ⑤ Counter

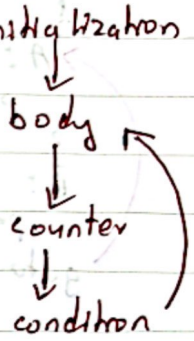
ENTRY CONTROLLED

while loop
for loop



EXIT CONTROLLED

do-while loop. (guaranteed to run once)




```
ex: int a=5;
    int b=10;
    do
    {
      printf("A\n");
      a--;
    } while ((a>0)?10:9)>=10);
```

* can nest but it becomes hard to read.

* Loop runs n times, condition is checked n+1 times.

* If the loop comes across a break statement, it will exit the loop.

CONTINUE STATEMENT (Only for loops) (Not defined for switches and conditions)

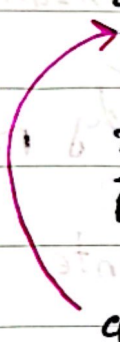
- Takes the program to the next iteration. (Skip the body and go to the next iteration.
* Basically skip a iteration.

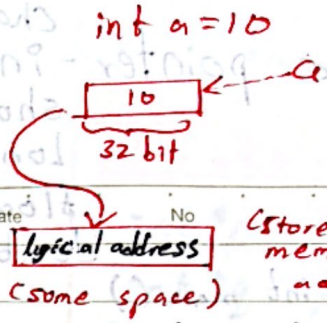
```
continue;
```

GOTO STATEMENT (explicit jump)

goto <label_name>;
<label_name>;

ex! for (int i=0; i<5; i++)
{
A:
printf("%d\n", i);
}
B:
printf("B\n");
goto A;





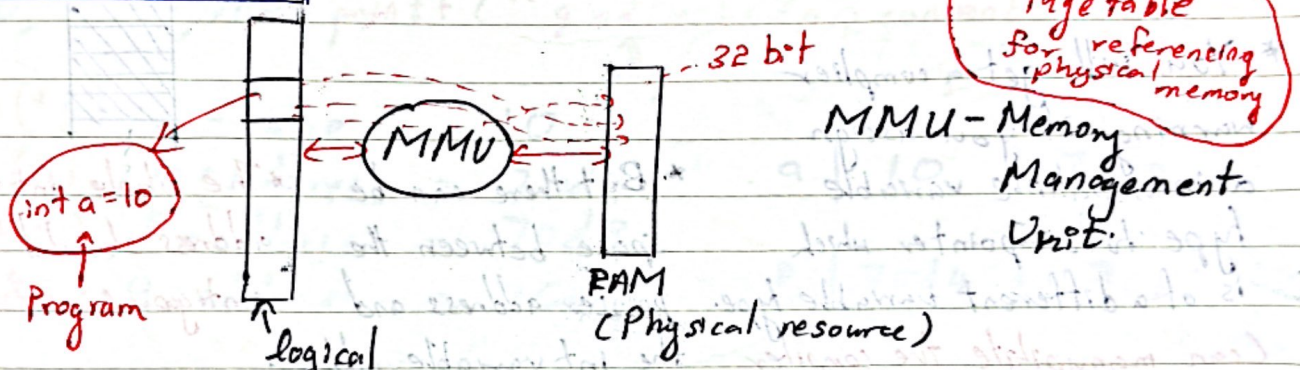
~~Random Variables~~ Pointers

* Holds memory address of a variable.
* Declared using * (indirection or differencing operator)
* Represents a logical memory address.

Memory has 2 forms: 1/ Logical Memory ⇒ memory where information is stored as seen by the computer.

* ← declare pointer
& ← get the address of the variable

2/ Physical Memory ⇒ the actual RAM of the computer.



* Allocation of logical memory to physical memory is done by mmu.

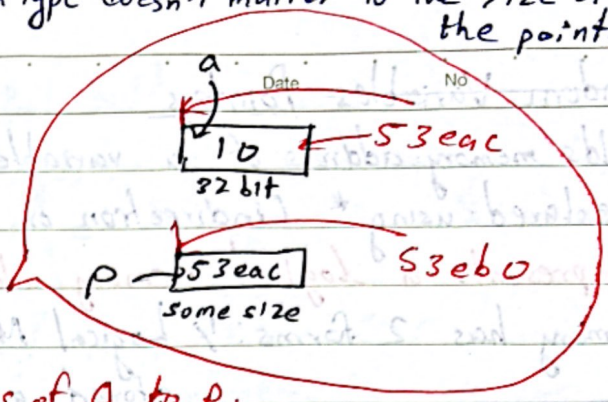
* Program sees memory as consecutively numbered cells.
* Cells can be manipulated individually or as contiguous groups. (groups one after the other without gap in memory)
* Address operator (&) gives the address of an object. (unary operator)

```
int a;
int *p;        <data type> * pointer-name    // to declare
p = &a;        pointer-name = & <object>     // to assign
```

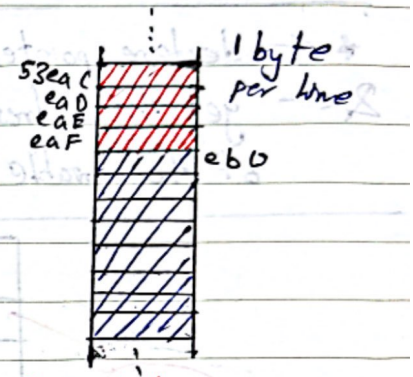
To retrieve and assign values to a memory ~~addr~~ location.

```
<variable> = *ptr    // variable is assigned the value
*ptr = <value>       // store the value
```

size pointer - char, int, short, long, float, double } (8 bytes)
(since it is a 64bit computer)
(if it is a 32 bit computer it will be the size of 4 bytes)
(data type doesn't matter to the size of the pointer)



ex: int main()
{
int a = 10;
int *p = &a;
P holds the address to a integer variable.
→ create pointer, assign address of a to p.
printf("%p\n", &a);
printf("%p\n", &p);
}



* You will get a complier warning if you assign a incompatible variable type to a pointer which is of a different variable type. (can manupilate the compiler to avoid warnings).

* But there can be space between the pointer address and the int variable address.

* The whole int variable address should be contiguous.

* You need data types for pointers to

* Advantage of using pointers is that you can do your operations in the same location where you initially created the variable without copying it several times. Which saves memory.

* Every time you run the program, the OS assigns address to your variables but once it is done executing it is destroyed.
* If you run the program again it will have different address.

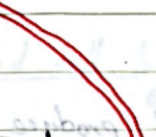
Dereference: printf("Pointed Value = %d \n", *p);
(go to address which p holds and print the value in that address). (Can be used to perform arithmetic operations)

&p - address of p *p - value of the variable
p - contents of p

Date

Alias of a pointer

```
int *p = &a;
int *q = p;
```



* If you change one other will also change

```
int **q
```



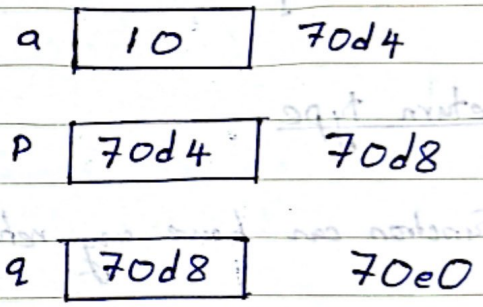
```
int a=10;
int *p = &a;
int **q = &p;
```

```
printf ("%p == value %d\n", p, *p);
printf("%p == value %d\n", q, *q);
```

70d8, 70d4, 10 `printf ("p address = %p, p content = %p, value = %d\n", &p, p, *p)`

70e0, 70d8, 70d4 `printf ("q address = %p, q content = %p, value = %d\n", &q, q, *q)`

(**q) (*p)	a - value	= 10
(*q) (p)	a - address	= 70d4
(q) (&p)	p - address	= 70d8
(&q)	q - address	= 70e0

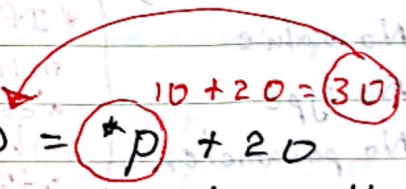


* Pointer to a pointer to a pointer to a double

```
double ***p;
```

Arithmetic operations using dereference

```
int a = 10;
int *p = &a;
int x = *p + 5;
*p = *p + 20
printf("%d\n", x);   15
printf("%d\n", a);   30
```




```
*p = *p + 20
```

step 1 : take value in address in pointer p
step 2 : do the addition (30)
step 3 : assign 30 to the value in address in pointer p

* ∴ you can read and assign values using dereference.
* Dangling pointer is when you got two pointers pointing to the same memory location and you delete one pointer causing the other pointer to point to nothing (garbage value)

Functions

Date No

* Block of functionality which takes inputs & produce an output
  * Not all functions take inputs
  * Not all functions produce outputs.

* Functions allow the reuse of logic within the same program.

```
<return_type> <function_name> ([<params>,*])
{
  //Function body
}
```

Return type

* Function can have any return type (primitive / user defined)

* If it doesn't return a value "void" type is used

* Void can mean,
  1/ No value
  2/ No type
  3/ No parameter.

* If you are trying to return a double it will only return the integer part.

ex:
```
int add(int a, int b)
{
  return a+b
}
int main()
{
  printf("%d\n", add(5,6));
}
```

Parameters

* Inputs to functions
  - Not mandatory to have parameters
  - But parameters cannot be optional
* Arguments are the real functional values passed onto the parameter.

```
int sum (int a, int b)
    return a+b
```

sum (6, 7)
actual parameters

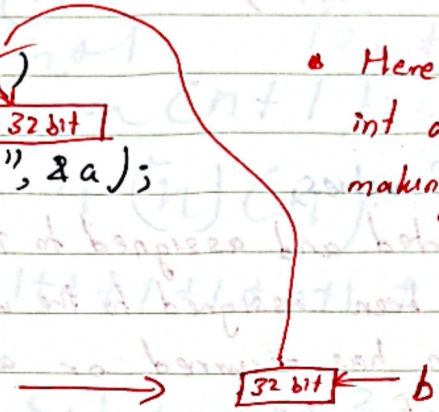
Date No

Call by value and call by reference.

Pass by value

* When parameters are passed using the pass by value methods, the actual parameters are copied to the formal parameters.

```
void test (int a)
{
    printf ("%p\n", &a);
}

int main ()
{
    int b = 45;
    test(b);
    printf ("%p\n", &b);
}
```



* Here the value of b gets assigned to int a of void test function. By making a copy of b in a.

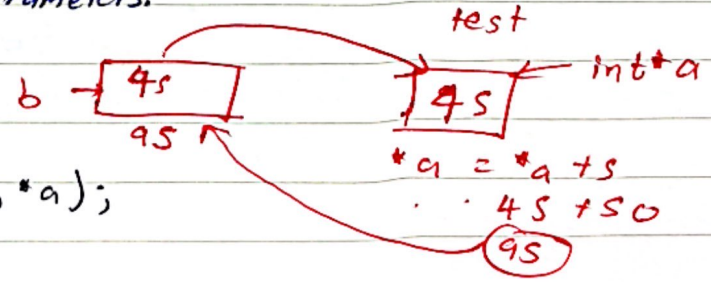
PASS BY VALUE

* Actual variables do not change
* Exists 2 copies of parameters
* Safer than pass by reference.

Pass by reference

* When parameters are passed, the address of the actual parameters are passed as formal parameters.

```
void test (int *a)
{
    *a = *a + 50
    printf ("%p %d \n", a, *a);
}
int main()
{
    int b = 45;
    test (&b);
    printf ("%p %d \n", &b, b);
}
```



* Here the address of b is passed as the parameter to the test function.

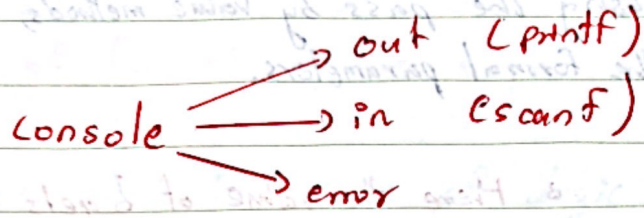
* Actual variables are modified within the function.
* Pointers are used.
* Preffered when large amounts of data is being passed to the function.

Date No

Scanf()

Syntax

int scanf (const char *format, variable_address);



* Scanf returns 3 types of values,
  1/ (>0) values are converted and assigned to the variables
  2/ (=0) no value has been assigned to the variables
  3/ (<0) error of stdin has occurred or and end of file is reached before the assignment was made.

* Supports scanset specifiers which are represented by %.[ ].
  To scan stuff like strings.

* When array element is not defined in modern versions it will give zero, but in old versions it might give garbage values.

* Cannot resize since you cannot guarantee contigous space remaining in memory to resized.

X	50	14	60	90	87	3	24	X
	32	32	32	32	32	32	32	

Date 3/7/25 No

Programming in C

Arrays : Group of elements of the same data type that is stored contiguously in memory.

* AKA homogeneous data structure
* Fixed size data structures.

1) Size either given initially 2) Or determined by the number of elements

```
<data_type> <array_name> [<size>];
<data_type> <array_name> [ ] = {<value 1>,... <value n>};
```

* Has zero based indexing
* To find the length of the array use sizeof()
* Elements of the array can be accessed through pointer arithmetic.

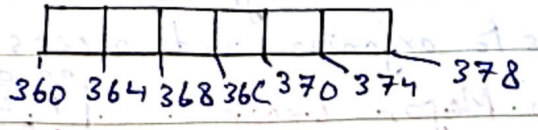
* getting the size of an array: printf("%ld\n ~~%d~~", (sizeof(array_name)/ sizeof(first_element)));

```
int a[ ] = {20, 50, 70, 94, 86};
printf("%p\n", a); // gives the address of a which is the starting point of the array (ie: first element address).
printf("%p\n", &a[2]); // address of 3rd element
printf("%d\n", a[2]); // value of 3rd element.
```

a - address of array &a[x] - address of an element
a[x] - value of an element.

```
for (int i = 0; i < 5; i++)
{
  printf("%p\n", a+i);
}
printf("%d\n", *(a+i));
```

Outputs the addresses of the elements.



* Important: When you ~~can~~ change the data type of ~~of~~ the array the gaps in which the pointers ~~of sign shaqat o ishte~~ are assigned addresses also change according to the data type.

ex: int 4 byte gaps
long 8 byte gaps ∴ type of pointers MATTER.

long
a = a0

a8 a[1] = a + 1(8) (8) *(a+1)
b0 a[2] = a + 2(8) (16) *(a+2)
b8 a[3] = a + 3(8) (24) *(a+3)

$$a[i] = \left(\boxed{\text{Base Address} + \text{index} * (\text{size of data type})} \right)$$

a i 8

* So if you keep increasing the index it will give random values from memory which are outside of the array. Cus pointers doesn't really care about the boundries of the array.

int
a[3]*5 : {*(a + 3*(4))} * 5 : (*(a+3))*5

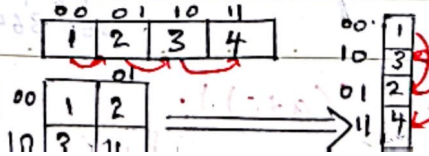
* If its not fixed size (size of data type) variable won't work.
* If its not contigous you cannot use indexing since it will point out to random values.

Multidimensional Array

* Arrays can have more than one subscript or an index

<data-type> <name>[size][size]... [size];

* Two ways to organize and access elements in a multi-dimensional array,
1/ Row Major Order
2/ Column Major Order



start from left side →

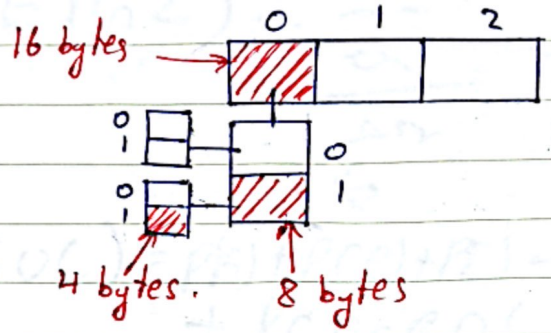
int a [4][3][2][2]

Date No

reading :- there is an array of 4 cells
- inside that array there is an array of 3 cells
- inside that array there is an array of 2 cells
- inside that array there is an array of 2 cells.

Total cells = 4 × 3 × 2 × 2 = 48 cells

ex: a[3][2][2] ⇒ a[0][1][1] = 6



Row Major ~~for~~ Order form Vs Column Major Order

ex: for (int i=0; i<2; i++)
for (int j=0; j<2; j++)

a[i][j]
0 0
0 1
1 0
1 1
(i, j)

~~for~~ (int j=0, j<2; j++)
for (int i=0; i<2; i++)

a[j][i]
0 0
1 0
0 1
1 1
(j, i)

* Main advantage of ~~row~~ row major order is that it is more efficient that column major order.
* Both ways work but row major is more efficient.

Types of pointers

Date No

Void pointers

```
void *<name>;
```

* This pointer can hold any type of variable address, (ie; primitive, user defined and etc). (Basically can be typecasted to any type).

ex:
```
void *p
int a = 10
*p = &a;
float f = 6.0f;
*p = &f;
```

* Void pointers cannot be dereferenced

```
printf("%d\n", *p);
```
✗ gives an error (cannot dereference void pointers)

so to dereference a void pointer, you basically need to define the type of variable it ~~was~~ points to.

```
printf("%d\n", *(int*)p);
```
✓ works. || `*(<data_type>*)<pointername>`

* Standard c doesn't allow for void pointer arithmetic. But gnc allows by

* Enables the implementation of generic functions
* Mainly used in implementing data structures.
* Discard type safety which is a disadvantage.

Function Pointers

Date No

return_type (*ptr_name) (param 1, param 2, ...);

* Enables calling a function using a pointer
* It points out to the start of the executable code.

ex: int add(int a, int b) ⟹ int (*x)(int, int); x = &add;
int Y() ⟹ int (*x)(); x = &Y;

* Useful in implementing call back functions. (can be used to create wrappers around existing functions).

Reading a function pointer: spiral rule is applied clockwise starting from the unknown element.

int *a[20]; : point of array of size twenty with type int

void *f(int, int); : function pointer(f) pointing to two integer parameters ~~or~~ and that points to a void pointer.

void *(*f)(int, double (*t)(int, int)); (f) is a function pointer with parameters int and a double (t) which is a function pointer with ~~para~~ 2 integer parameters which points out to a void pointer.

void* (*f(int, double (*t)(int, int)))(int);

⇓

void* (*x)(int)

writing the whole thing as a function pointer.

Constants and Pointers

Date No

Pointer to a constant

```
const data_type *<pointer_name>;
```

* Data pointed by the pointer can only be read, and cannot be changed by the pointer.

* Ensures that you accidently do not change data pointed by the pointer.

```
ex: int a = 23;
    const int* ptr;
    ptr = &a;
    int b = 50;
    ptr = &b;
    printf("%d\n", *ptr);
```

* But it is possible to change the value by changing the variable the pointer points to, or by directly changing the value of the variable a.

Constant pointers

```
data_type *const <pointer_name>;
```

* Should be assigned at initialization
* Constant pointers will always point to the same memory location and once assigned, cannot be changed.

Summary

	Reassign	Change Value	Initialized
pointer to a constant	✓	X	anytime
constant pointer	X	✓	@decl.
Constant Pointer to a constant	X	X	@ decl

(const <data type>* const <pointer_name>;)

Null Pointers

Date No

- A pointer that doesn't point to any location.
- Obtained by assigning a pointer to a zero(0) or NULL.
- It will result in different behaviours across different machines.

ex: int* a = 0/NULL;

Null Pointer

- Initialize pointers without an initial address
- To check for null before accessing a memory location.
- To pass to functions when there is no valid memory address to provide, indicating that the pointer intentionally points to nothing.

Void Pointer.

- Points to a memory location
- Not all void pointers are the same
- Void pointer is a type.

Wild Pointers.

- These pointers are uninitialized and may point to an arbitary memory location.

ex: int* a;
void* x;

- These pointers makes the program work unpredictably or cause crashes

Strings

- In C, string is a sequence of characters terminated with a null character ('\0').

one dimentional array

char <string_name>[array_size]

ex: Hello

H	E	L	L	O	\0

← array size should be n+1 accounting for the null character.

for: ~~string~~ char string[] ⇐ \0 character should be inserted at the end to show the end of the string.

<string.h> - contains inbuild string manipulation functions.

%s - format specifier

Date No

Scansets: format specifiers used with scanf() in C to read strings that match a specific set or range of characters.

* Scansets do not skip leading whitespaces, so leave a space before the scanset in scanf() if you wanna skip whitespaces.

Structures (better to create outside main(), but it can also be assigned inside main())

* structures are a user defined type in C.

```
struct <name>
{
    <data_type 1> <variable_name 1>;
    <data_type 2> <variable_name 2>;
    ...
} [variable name 1, variable name 2...];
```

* Here there is a fixed number of variables for the structure

* All items under a structure are called 'members' of that structure.

```
struct <name>
{
   <data-type 1><variable_name 1>;
   <data-type 2> <variable_name 2>;
   ...
}
```

* Here you can define any number of variables for the structure.

```
struct <struct_name> <variable_name>
<variable_name ><member-name > ⟸ access a member. = value
```

Pointer to a structure

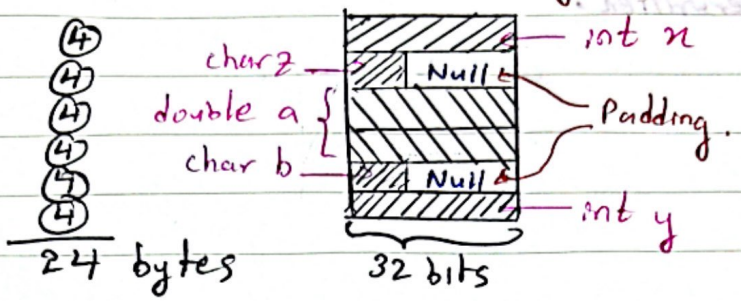
```
struct<struct_name> <variable_name>;
struct <struct_name> *<pointer_name>;

<pointer_name> = &<variable_name>;
<pointer_name> -><member_name> = value;
```

⇑ use arrow operator when accessing members through a pointer to a structure.

Size of structures: int x → 4
int y → 4/8
actually = 8

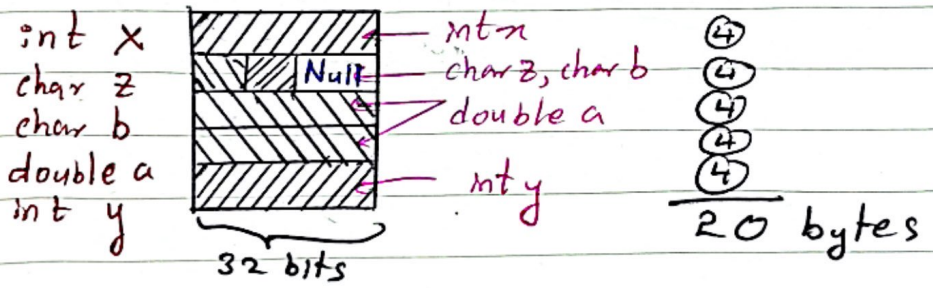
int x → 4
int z → 1
int y → 4/9
actually = 12

int x → 4
char z → 1
double a → 8
char b → 1
int y → 4/18
actually = 24



* So basically padding optimizes the memory usage, cus without it the double value will be there in 3 address and it will waste computational power cus the machine has to read 3 memory addresses.

* You can optimize the above structure by changing the order of which the variables are declared.



* Members of structures are allocated within a contiguous block of memory.

Unions (size of union = size of largest data member in the union)

* User defined data type that allows data types to be stored in the same memory location.

* Since all members are stored at the same memory location, all members cannot be initialized at the same time.

* Basically it will allocate the size of the largest type.

* Primary application is to save memory.
* similar syntax to structures when used as a pointer.

* If multiple members are assigned values @ the same time, the data can be ~~overlaid~~ overwritten.

static Keyword

* Applied to both variables and functions.

* When applied to variables, it has the property of preserving its value even after they are no longer in scope.

```
static <data_type> <var_name> = value;
```

```
ex: int x ()
    {
      static int a = 0;
             a++;
             return a;
    }
    main()
    {
1 ← printf("%d\n"; x());
2 ← printf("%d\n"; x());
    }
```

* Without static the output will be 1 and
* But when you use static the value of a will be preserved and next time the function is called the previous call will be incremented. Output is 1 and 2.

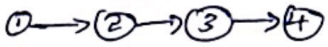
* If you don't use static properly you could waste memory, cus if you aren't using the variable again, it will be stored invain.

* Static variables must not be declared in a structures, but static structures are allowed.

Static functions:

```
static <return_type> <function_name>([params]...)
{
}
```

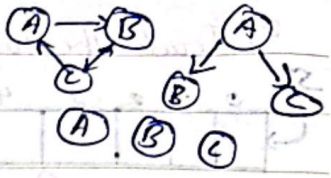
* static functions are restricted to files they are declared in and mainly used



Data Structures

Stacks

- Linear structures : their elements are arranged sequentially, and they are connected to adjacent elements.
- Non-Linear structures



* Only Linear structures are done in this course.

* Doesn't necessarily have to be contigrous but arrays are an example for contigrous linear structures.

ex: Arrays, Linked Lists, Stacks, Queues. Linear structures.

ex: Graphs, Trees. Non-Linear structures.

* To get a non-contigrous sequential linear structure, basically the memory blocks are connected by pointers.

Abstract Data Types (ADT)

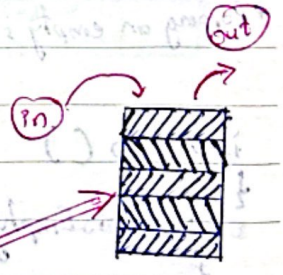
* It is a class of objects whose behavior is defined by a set of values and operations seen by the user.

* Data structures provide how the behaviour mentioned by an ADT is implemented.

(3) fundamentals operations to learn for any data structure,

1) How to insert data
2) How to delete data
3) How to search data

* In addition there would be structure specific operations.



Stack (implemented using arrays or linked lists)

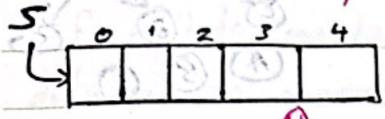
* Follows last in first out policy (LIFO).

* LIFO is not only used in stacks.

* Used in backtracking, functional calls, memory management, syntax checking and etc.

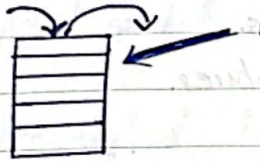
* Good practice to do is ensure that functions do one thing and one thing only.

* Things to keep track of when using arrays as stacks,
  1/ Size of array
  2/ Track the top of stack (TOS) ⇒ basically hold the array index of the top element.



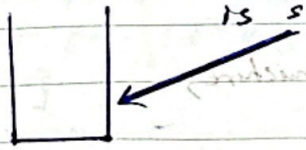
Here TOS is -1.

insert (push) : Worst thing that can happen is stack over flow.



```
void push(int v) ⇐ To push an element into the stack,
{
 if (!isfull())
  {
   S[++TOS] = V;        O(1)
  }
 else
  {
   printf("stack is full");
  }
}
```

delete (pop) : Worst thing that can happen is stack under flow.



* Popping an empty stack

```
int pop()
{
 if (!isEmpty())        O(1)
  {
   return S[TOS--] ⇐ So here you cannot delete from memory since it is an array. What is done is next time an element is pushed, the value will be overwritten.
  }
 else
  {
   return -1;
  }
}
```

```
int isFull() ⇐ check if stack is full
{
 return TOS == size-1;        O(1)
} // guards against stack overflow
int isEmpty() ⇐ check if stack is empty.
{
 return TOS == -1;        O(1)
}
```

peek() : retrives tos without deleting it.

```
int peek()
{
 return tos;        O(1)
}
```

Macros

* It is the name given to a piece of code such that when the preprocessor encounters it in the program, it will be replaced by the code defined.

Object like Macros.

```
#define PI 3.146
```

Not terminated by a semicolon.

Function like Macros

```
#define add(a,b) (a+b)
```

* There are also predefined macros in C like,

__DATE__, __TIME__

Memory Layout of a program

Date No

- C program's memory representation consists of following (5) sections,

Text segment - contains executable instructions, it is shareable so only a single copy is required.

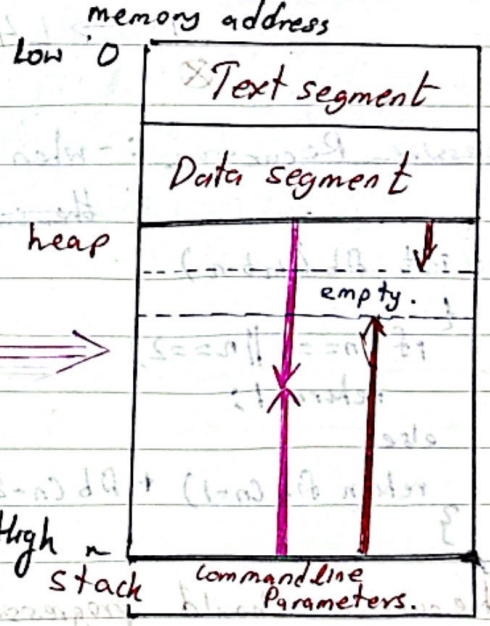
Data segment - Initialized - global and static variables that are initialized
Uninitialized (bss) - for uninitialized global and static variables or ones set to zero.

Heap memory - dynamic memory allocation (basically a collection of memory space where things are stored randomly)
Used by malloc(), free(), realloc() and calloc();

Stack memory - contains program stack

size men (in the terminal)

- Gives the sizes of each section in memory.
  - This way, when stack starts from high memory address and heaps start from low memory address, the amount of space allocated is not limited, it can grow until ~~he~~ each other is reached.

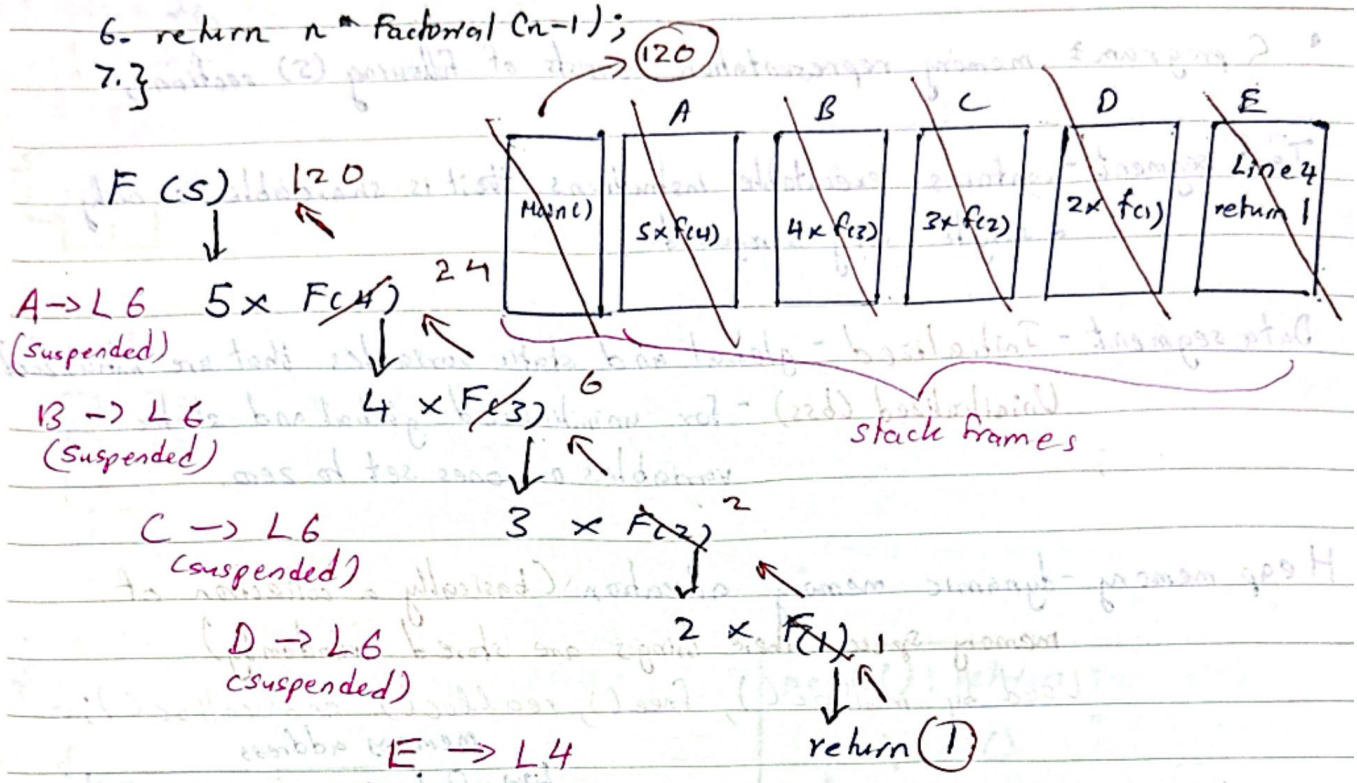


Recursion

- It is "defining a problem in terms of itself" or in programming terms "a function calling another instance of itself".

- Recursion has 2 properties: Base case (anchor point)
  Recursive step

eg: 1. int Factorial (int n)
2. {
3. if (n==1)
4. return 1;
5. else
6. return n * Factorial (n-1);
7. }



Excessive Recursion :- when the same problem is solved repeatedly.

ex: int fib (int n)
{
if (n==1 || n==2)
return 1;
else
return fib (n-1) + fib (n-2);
}

* Here fib(3), fib(4), fib(2) is calculate multiple times.

* Recursion should progressively lead to the base case, i.e. it should reduce the problem to smaller and smaller sizes.

∴ only division and subtraction can lead recursion to its base case.

inserted → D C B A → removed

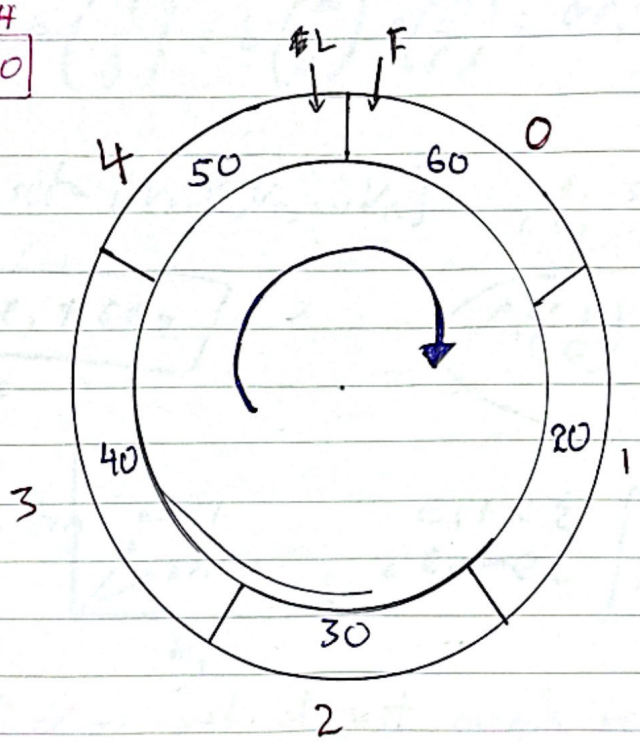
Date No

Queues

- First in first out policy. (FIFO)
- Implemented using arrays or linked lists.
- We need two pointers to keep track of first element & last element.

0	1	2	3	4
60	20	30	40	50

1. Enqueue()
2. Dequeue()
3. Front()/Peek()
4. isFull()
5. isEmpty()



int first, last

↑ keeps track of first element

↑ keeps track of last element

isEmpty

F = L = -1

OR F > L

Is full

⇓

L + 1 = F

Dynamic Memory Allocation and Linked lists

Date No

Dynamic Memory Allocation

* It is the process of assigning memory to a program @ runtime, giving the control to work with data structures of varying sizes.
  - Increases flexibility
  - Increases efficiency.

* Occurs in both stacks and the heap memory.
  For non-function call based it is assigned from the heap.

<stdlib.h> ⇒ malloc(...), calloc(...)

size_t : guarantees to be big enough to handle the largest possible object in the host system. (never negative)
32 bit system ⇒ unsigned int
64 bit system ⇒ unsigned long long

malloc(...) (memory allocation)

* Reserves a block of memory of a specified size from the heap & return a pointer.

```
void *malloc(size_t size);
```

(returns a void pointer cus it can be type casted to any type)

ex: `void *p = malloc(20);`

`int *p = (int*)malloc(20);`

actually the array size is 5 ($\frac{20}{4} = 5$)

32 bits = 4 bytes
∴ 20 bytes is 5 integers

or

```
void *a = malloc(20);
int *p = (int*)a;
```

cast

* Size parameter expects the size of the memory block required in bytes.

* If an value is not initialized at execution to the function, garbage values present at the time of allocation are the initial values of the memory block.

* If no sufficient space is available to allocate, a null pointer is returned. (i.e. check whether it returns a null point)

calloc(...) (contiguous allocation)

Key differences: - Initialises each memory block so that each element is zero
- 2 parameters: total elements (n), size of elements
- Returns a void pointer

```
void *calloc(size_t nelem, size_t elsize);
```

malloc(..) faster than calloc(...).

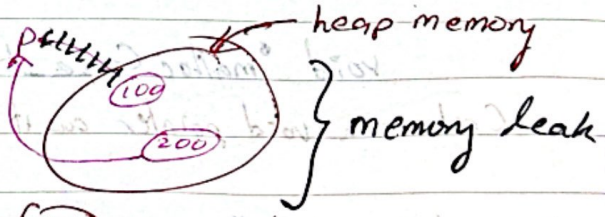
free(...)

* Deallocate memory that was allocated by malloc(..) and calloc(...)

```
void free(void *ptr)
```

* Recommend to free memory that has been allocated.

```
ex: 1. int *p = 0;
    2. p = (int*) malloc(100);
    3. p = (int*) malloc(200);
```



* 100 is no longer assigned to anything, therefore no one can use 100 until program ends executing.

* So if the program keeps on leaking memory, it will eventually run slower and slower and finally crash.

automated tools: sudo apt install valgrind
valgrind --leak-check=full <executable>

realloc(...)

* Reallocates previously allocated memory when the memory allocated by malloc(..) and calloc() is insufficient.

```
void *realloc(void *ptr, size_t new_size);
```

ptr :- pointer previously returned by malloc, calloc or realloc.
- if ptr == NULL, realloc behaves like malloc(new_size)

new_size :- new size in bytes of the memory block.

Register Keyword

```
register <data_type> <var_name> = <value>;
```

* Even though you use register keyword, the compiler ultimately decides whether to put the variable into the register.

Auto keyword

* In modern C, writing auto is redundant because it is the default behavior for local variables.
* It basically means variables are allocated storage when the block (function, loop, etc) is entered. And is deallocated when the block is exited.
  ∴ int a; == auto int a;

Command Line Arguments

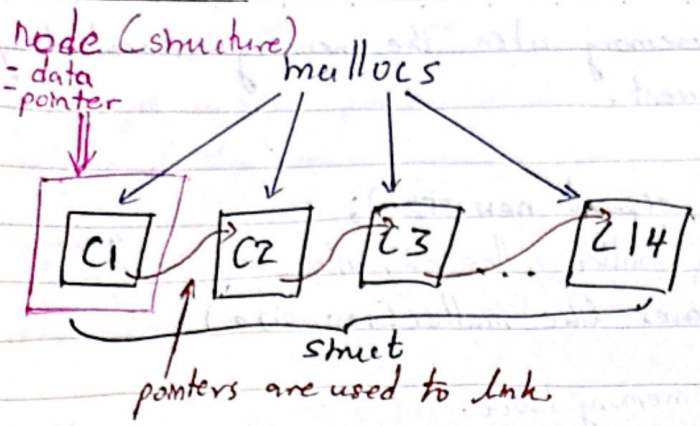
```
int main(int argc, char *argv[]) {
}
```

ex! ./a.out (1) x (2) y (3) z (4)

argc (argument count) - 4
argv[] - argument values

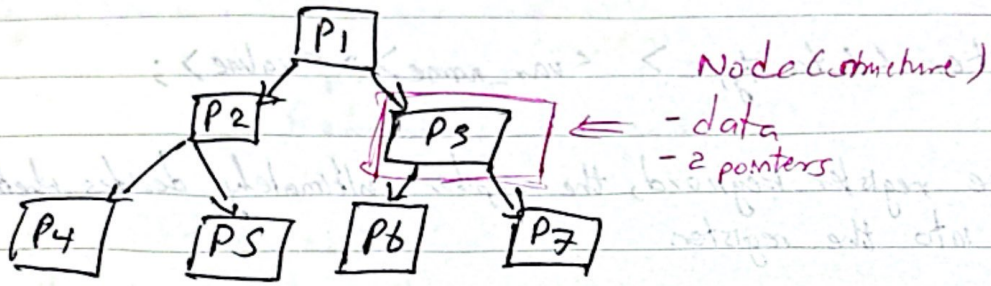
⇒ Basically giving information for a program to run. (NOT USER INPUT)

Linked lists.




```
struct Node {
    int data;
    struct Node * ptr;
};
```

* You can also be more creative,




```
struct Node {
    [<data_type> <member_name>]+;   <- + (1 or more)
    [struct Node * <pointer_name>]+;
}
```

* The nodes can be created by;

```
void *temp = malloc (sizeof (size of (struct Node)));
                    (struct Node *)
struct Node *a = Malloc (size of (struct Node));
```

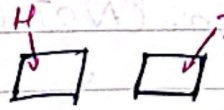
- Add ToHead
- Add To Tail
- DeleteFrom Head
- Delete From Tail.

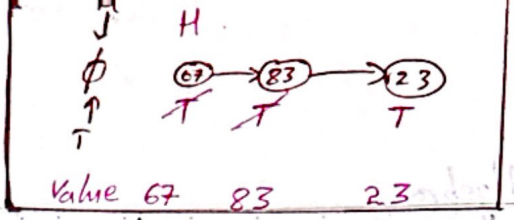
```
a -> data = 25
a -> ptr = &...
```

Singly Linked Lists

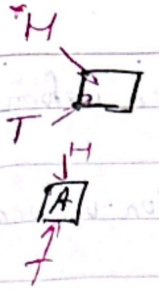
(There is a single link between elements which enables a given node to point to the next node in the list)

1/ H = T = ϕ

2/ 



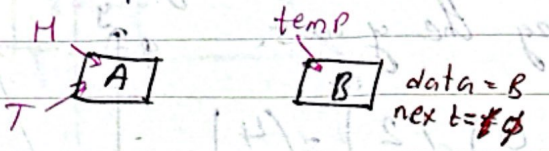
Solution case 1:

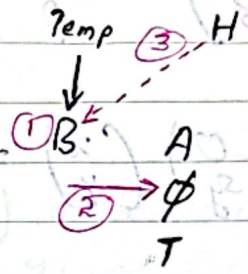


1. temp → malloc
2. H = T = temp

addToHead: ⇒ new element becomes the head

1. temp → malloc(B)

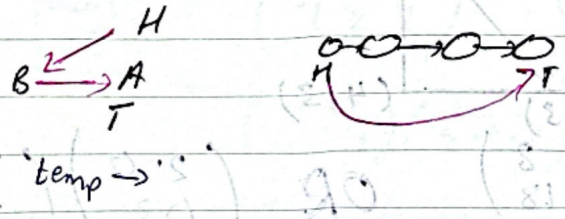




2. temp → next = H;

3. H = temp

This is wrong!

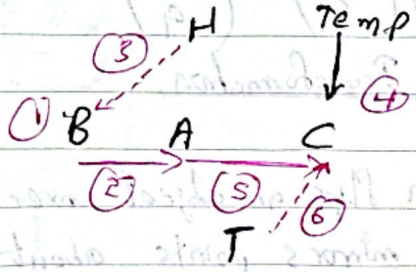


addToTail: ⇒ the element becomes the new tail.

3. Temp = malloc

4. T → next = Temp;

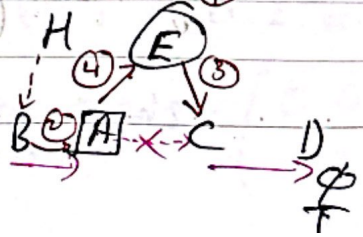
5. T = Temp



Inserting node to the middle of the linked list.

Inserting (E) between A and C

1. temp = malloc(E)
2. find A (start from head and loop until A is found)
3. temp → next = Temp2 → next
4. temp2 → next = temp



C programming

Date No

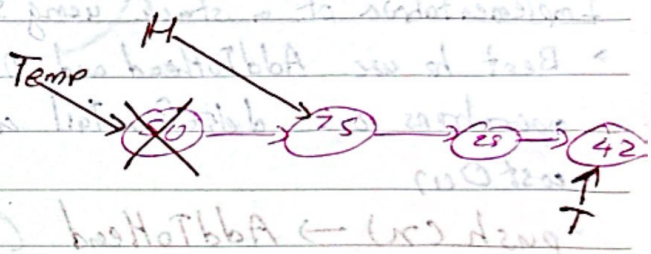
#ifndef YOUR_NAME; #include "your-name.h" ⇐ include in the file.
#define YOUR_NAME;

x x x x x
x x x x x

#end;if;

* You can also delete elements from linked lists in 4 ways,
  1/ delete from head
  2/ delete from tail
  3/ delete an arbitrary node
  4/ delete a single node (H=T)

→ ① Delete from head



Case I - if H=T [H=T=NULL]
Case II - Temp=H
H = H → next
Afterwards free the temp pointer (free (temp);) Big O(1)

② Delete from tail
Case I - if H=T [H=T=NULL]
Case II - loop to find node before tail [temp → next != tail] = temp
free (temp → next)
temp → next = NULL
Big O(n)

for (temp=head; temp → next != tail; temp = temp → next);

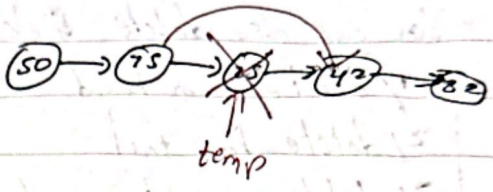
③ Generic Insert
Insert (X, Location, Pre/Post)

ex: Insert (78, 75, Pre) Insert (78, 75, Post)
50 → (78) → 75 50 → 75 → (78)

if location is head
  if Pre → addToHead

if location is tail
  if Post → addToTail

• Delete generically: 75 → next = 25 → next
temp = 25
temp → next = null
free(temp)



Implementation of a stack using Singly Linked List.

* Best to use AddToHead and DeleteFromHead operations than tail operations cus deleteFromTail costs O(n) while head operations only cost O(1)

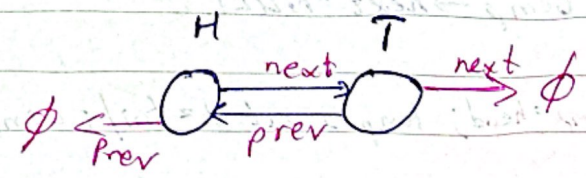
• push(x) → AddToHead O(1) AddToTail O(n)
• pop() → DeleteFromHead O(1) DeleteFromTail O(n)

Implementation of a queue using Singly Linked Lists

* Here also DeleteFromTail is best to be avoided.

enqueue(x) → AddToTail O(1) AddToHead O(1)
dequeue() → DeleteFromHead. O(1) DeleteFromTail O(n)

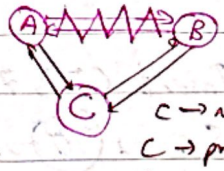
Doubly Linked Lists



* There two pointers are used next and prev. (for the next element & previous element)

```
struct Node {
    [<data_type> <member_name>]+;
    struct Node* next;
    struct Node* prev;
}
```

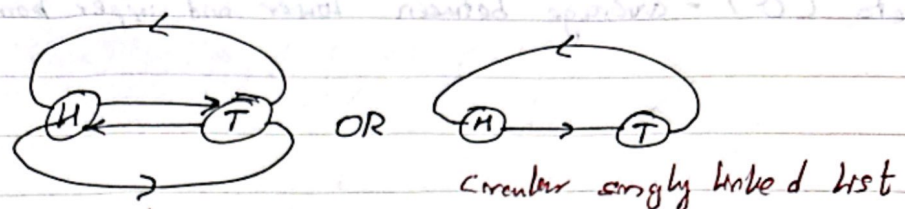
Inserting an element



C → next = B
C → prev = A
A → next = C
B → prev = C

Circular Linked List

- Can be created in a singly or doubly linked list manner.
- Only difference from the previous two is that the head is connected to the tail.



Circular doubly linked list.

Circular singly linked list

Insertion sort

10 | 8 2 1 5 4

sorted | unsorted

sorting in ascending order

8 | 10 2 1 5 4

8 | 10 | 2 1 5 4

8 2 | 10 1 5 4

2 8 | 10 1 5 4

2 8 10 | 1 5 4

sorted | unsorted

Selection Sort

- Find the minimum element in the array a switch it with the first element in the unsorted part.

10 | 8 2 1 5 4

1 | 8 2 10 5 4

1 2 | 8 10 5 4

1 2 4 | 10 5 8

sorted | unsorted

1 2 4 5 | 10 8

1 2 4 5 8 10 (Sorted)

Algorithm Analysis

Date No.

```
Algorithm sum(A,n)
{
  s=0          — 1
  for(i=0;i<n;i++)  — n+1
  {
    s=s+A[i];  — n
  }
  return s;    — 1
}
```

f(n) = 2n+3

O(n)

Space: A = n, n = 1, s = 1, i = 1

S(n) = n+3

O(n)

Algorithm Add(A, B, n) * Add two matrices of size n×n.

```
{
  for(i=0;i<n;i++)  — n+1
  {
    for(j=0;j<n;j++)  — n×(n+1)
    {
      c[i,j]=A[i,j]+B[i,j]  — n×n
    }
  }
}
```

f(n) = 2n² + 2n + 1

O(n²)

Space: A - n², B - n², C - n², n - 1, i - 1, j - 1

S(n) = 3n² + 3

O(n²)

Algorithm Multiply(A, B, n)

```
{
  for(i=0;i<n;i++)  — n+1
  {
    for(j=0;j<n;j++)  — n(n+1)
    {
      c[i,j]=0;
      for(k=0;k<n;k++)  — n×n(n+1)
      {
        c[i,j]=c[i,j]+A[i,k]*B[k,j]  — n×n×n
      }
    }
  }
}
```

f(n) = 2n³ + 3n² + 2n + 1

O(n³)

Space: A = n², B = n², C = n², n = 1, i = 1, j = 1, k = 1

S(n) = 3n² + 2

O(n²)

1/ for(i=0; i<n; i++) — n+1
{ stmt; — n
}
O(n)

2/ for(i=n; i>0; i--) — n+1
{ stmt; — n
}
O(n)

3/ for(i=1; i<n; i=i+20) — n/20
{ stmt; }
O(n)
f(n) = n/20

4/ for(i=0; i<n; i++) — n+1
{
for(j=0; j<n; j++) — n×(n+1)
{ stmt; — n×n
}
}
O(n³)

5/ for(i=0; i<n; i++)
{ for(j=0; j<i; j++)
{ statement; }
}
1+2+3+...+n = n(n+1)/2
f(n) = n²+1/2 O(n²)

i	j runtime
0	0×
1	0✓
	1×
2	0
	1
	2×
n	n

i	P
1	0+1 = 1
2	1+2 = 3
3	3+3 = 6
4	6+4 = 10
k	1+2+3+4+..+k

6/ P=0;
for(i=1; P<=n; i++)
{ P = P+i;
}

∴ P = k(k+1)/2

k(k+1)/2 > n O(√n)
k² > n
k > √n

7/ for(i=1; i<n; i=i*2)
{ statement; }

Assume i >= 0
∵ i = 2^k
∴ 2^k >= n
2^k = n
k = log₂ n O(log₂ n)

i
1
1×2 = 2
2×2 = 2²
2²×2 = 2³
⋮
2^k

if n=8

i	
1	
2	3
4	
8×	

log 8 = 3
log₂ 8 = 3

if n=10

i	
1	
2	4
4	
8	
16×	

log₂ 10 = 3.2

So we take ⌊log n⌋ Proc

8/ for (i=n; i>=1; i=i/2)
{
statement;
}

$i = n$

$\frac{n}{2}$

$\frac{n}{2^2}$

$\frac{n}{2^3}$

$\frac{n}{2^k}$

Assume $i < 1$

$\therefore \frac{n}{2^k} < 1$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 n$

$O(\log_2 n)$

9/ for (i=0; i*i<n; i++)
{
statement;
}

Stops executing when $i * i >= n$

$i^2 = n$

$i = \sqrt{n}$ $O(\sqrt{n})$

10/ for (i=0; i<n; i++)
{ statement; —— n
}
for (j=0; j<n; j++)
{ statement; —— n
}

$2n$ $O(n)$

11/ p=0
for (i=1; i<n; i=i*2)
{ p++; —— $p = \log n$
}
for (j=1; j<p; j=j*2) $O(\log \log n)$
{ statement; —— $\log p$
}

12/ for (i=0; i<n; i++) — n
{
for (j=1; j<n; j=j*2) — $n \log n$
{
statement; — $n \times \log n$
}
}

$2n \log n + n$ $O(n \log n)$

SUMMARY

for(i=0; i<n; i++) — $O(n)$
for (i=0; i<n; i=i+2) — $n/2$ $O(n)$
for (i=n; i>1; i--) — $O(n)$
for (i=1; i<n; i=i*2) — $O(\log_2 n)$
for (i=1; i<n; i=i*3) — $O(\log_3 n)$
for (i=n; i>1; i=i/2) — $O(\log_2 n)$
for (i=0; i*i<n; i++) — $\sqrt{n}$ $O(\sqrt{n})$

File Handling in C

Date No

1/ File Pointer : File *fp;

2/ Open a file : fopen()

Modes of opening files: 1/ open for reading : "r"

2/ Open for writing (creates if it doesn't exists) : "w"

3/ Append to end of file : "a"

4/ Read and write : "r+"

5/ Write and read (truncates) : "w+"

6/ Append and read : "a+"

3/ Write to a file : fprintf() or fputs()

4/ Read from a file : fscanf() or fgets()

5/ End of file : feof(fp)

6/ Error checking : ferror(fp)

7/ Closing a file : fclose(fp) (Alway close. It prevents memory leaks and corruption.