25.07.2022 Monday

PRINCIP OF programming Languages.

Date: / / Page no.

(#) printf( ) {
return no. of char.
L to R }

Ex. printf("%d", (printf("computer")));

Sol: 8 computer

Q) for (printf("Car"); printf("bike"); printf("ABC")) 3
i=3 { printf("Z");
}

Sol: ABZC BZC BZC BZC

PPL Sumeet Gupta Sir

High level language — Compiler/Interpreter → low level lang

(#) Reasons for studying programming languages :-

① Increase capacity to express ideas.

② People can easily express their ideas clearly in any ~~idea~~ language only when they have clear understanding of natural language.

③ Similarly, if programmers want to simulate the features of languages in another language they should have some ideas regarding the concepts in other language as well.

(#)

Scanned with CamScanner

26.07.2022
Tuesday.

Date: / / Page no:

~~Improved~~

② Improved background for choosing appropriate language.

(i) Some professional programmers have had little formal education in CS. They have developed their programming skills independently or through in-house training programmes. Such training programmes often limits the instructions to one or 2 languages that are directly relevant to current project of the organisation.

(ii) If the language they learned are no longer widely used and many features now available in programming were not commonly known at the time, such programmers when given a choice of languages for a new project continue to use language with which they are most familiar even if it is poorly suited for the project.

[ex. C++ has OOPS concepts easily
But C, has OOPs concepts but little difficult to implement]

(iii) If these programmers were familiar with a wider range of languages they would be better able to choose the language that includes the features that best address the characteristics of the problem.

③ Increased ability to learn new language.

In software development, continuous learning is essential. The process of learning new programming languages can be lengthy and difficult specially for

Date / / Page no:

someone who is comfortable with 1 or 2 languages.

Once a true understanding of the fundamentals concepts of languages is acquired, it becomes ~~quite~~ far easier to see how these concepts are ~~imparted~~ incorporated into the design of language being learned.

(4) Better understanding the significance of implementation

An understanding of implementation issue lead to an understanding of why languages are designed, the way they ~~there~~ are. This knowledge in turn leads to the ability to use a language more intelligently as it was designed to use.

By studying the concepts of programming languages programmers can learn about previously unknown ~~or~~ & unused part of languages which they already used and began to use that feature.

For ex.:- Programmers who knows little about the complexity of implementation of sub programmes calls often do not realize that a small sub program that is frequently called can be highly inefficient designing choice.

BD

Time complexity of * > +
2*a > a+a
O/P is same

Scanned with CamScanner

Date: / / Page no:

(5) Better use of languages that are already known.
Most programming languages are large and complex. Accordingly, it is uncommon for a programmer to be familiar with and use all of a feature of a language. By studying the concepts of programming language, programmers can learn about previously unknown or unused part of the languages they already and begin to use that feature.

(6) Overall advancement of computing.
There is a global view of computing that can justify the study of programming language concepts the most popular language ~~is~~ are not always the best available.

ESCAPE SEQUENCE

Imp

(3)

	i/p	o/p
(1)	MANIT BPL	BPLNIT
(2)	MANIT BPL	BPL MANIT
(3)	MANIT BPL	MANIBPL
(4)	~~MANIT BPL~~ NAME SURNAME	SURNAME NAME
(5)	Hello User Welcome to C/C++	Hello User Welcome To C/C++
(6)	Your Name	"Your Name"
(7)		Back Slash \ / Front Slash \ / \ / \ /

Scanned with CamScanner

Date: / / Page no:

(8) \n : is newline
(9) \t : is Tab
(10)

(1) printf(" MA\b\bNIT \r BPL");

(2) printf(" MANIT \r BPL\t");

(3) printf(" MANIT \b\b BPL");

(4) printf(" NAME \r SURNAME");

(5) printf(" Hello \n\t User \n \t\t Welcome \n \t To \n C/C++");

(6) printf(" \" Your Name \" ");

(7) printf(" \\ \/ \\ \/ \\ \/ \\ \/ ");

(8) printf(" \\n : is newline");

(9) printf(" \\t : is Tab");

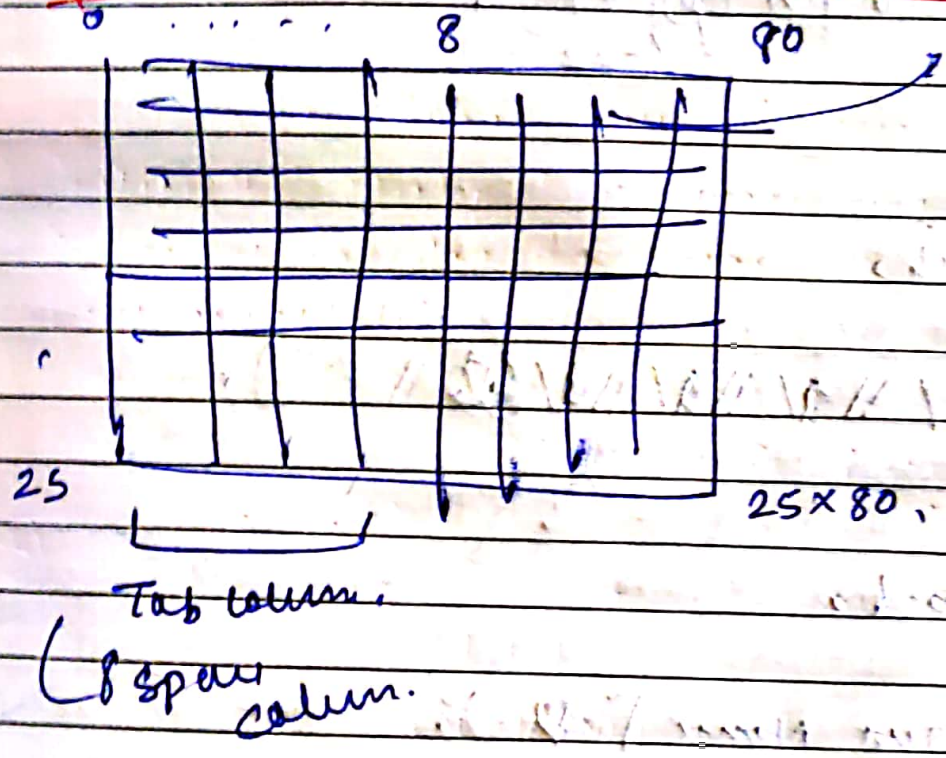
(1)
```
#include <stdio.h>
int main () {
  printf(" MA\b\bNIT \r BPL ");
return 0;
}.
```

Scanned with CamScanner

Symbols
① "
② \ } use backslash \ to print

Tab: collection of 8 spaces

\n → newline
\t → Tab character → jump cursor 8 space column
\b → backspace character → just like backspace but it ~~delete~~ does not delete character but overwrites it
\r → carriage return
~~Cur~~ moving cursor first time of row



~~MANIT \b\b~~

② ~~#~~include <stdio.h>
int main ( ) {
printf ("MANIT\rBPL ");
return 0;
}

③ ~~#~~ include <stdio.h>
int main ( ) { printf("~~MA~~NIT \b\b\bBPL");
return 0;
}

Scanned with CamScanner

Date: / / Page no:

(4)
```
#include<stdio.h>
int main(){
    printf("NAME \r SURNAME ");
    return 0;
}
```

(5)
```
#include<stdio.h>
int main(){
    printf("Hello \n\t your \n\t\t Welcome \n\t
            to \n C/C++");
    return 0;
}
```

(6) (1)
```
#include<stdio.h>
int main(){
    printf("\\a/ \\a/ \\a/ \\a/ \\ /");
    return 0;
}
```

(7)
```
#include<stdio.h>
int main(){
    printf("\"Your Name\"");
    return 0;
}
```

(8)
```
#include<stdio.h>
int main(){
    printf("\\b n ; is a newline");
    return 0;}
```

(9)
```
#include<stdio.h>
int main(){
    printf("\\t ; is a Tab");
    return 0; }
```

Scanned with CamScanner

01.08.2022
Monday

Date: / / Page no:

Q) printf("\");

Sol: o/p: error

printf("\n");

starting

print

but no end "

correct:

printf("\n");

Q) printf("\\\");

Sol: ~~com~~ error

no end "

Correct:

printf("\\\\");

PROGRAMMING DOMAINS

① Scientific Applications :- The first digital comp. in early 50's were invented and used for scientific and Engineering applications.

The scientific Application of that time used relatively simple data structure and required large number of floating pt- arithmetic computation.

floating pt. Number.

The most common data structure were arrays and matrices.

The most common control structures were counting loops and control statements.

↳ for loop type.

used to store homogeneous data type in contiguous memory locations

The early high level programming languages invented for scientific applications were

Scanned with CamScanner

Date / / Page no.

designed to provide for those needs where competition was Assembly language, efficiency was a primary concern

↳ symbolic representation of Machine language.

language nearer to Machine language

⇒ takes less time to compile.

T(High level language) > T(Assembly language)

The first language for Scientific application was FORTRAN (1957)

② Business Applications :-

(i) The use of computers for Business Applications began in 1950s

(ii) Business languages are characterised by facilities for producing reports, precise way of describing and storing decimal Numbers and character data, ability to specify decimal ~~Ability to data~~ arithmetic operation.

Spreadsheets +, -, \, *, %.

(iii) special Computers were developed for these purpose along with the special languages.

(iv) The first successful high level language for Business was COBOL (1959)

It is still most commonly used language for these application

→ egg

→ pan (OS)

→ stove

→ data entry language

③ System Programming Applications :-

Fortran kept user & machine

(i) The operating system and the programming support tools of a computer system are collectively known as its System Software.

OS + Programming ST = SS

System Software is used almost continuously and so it must be efficient.

Time efficient

Space efficient

Scanned with CamScanner

Date / / Page no.

(ii) In the 1960s and 1970s some computer manufacturers such as IBM developed special machine oriented high level languages for System Software on their machines.

→ For IBM mainframe computers, the language was PL/S

→ However most system software is now written in more general programming languages such as C and C++

→ The Unix OS is written almost entirely in C which has made it relatively easy to code or move to different machines.

(4) Artificial Intelligence (AI) :-

Not numbers ; statements type No calculation required

(i) AI is broad area of computer application characterisitcs by the use of symbolic rather than numeric computation.

(ii) Symbolic computation means that symbols consisting of names rather than numbers are manipulated.

(iii) Symbolic computation is more conveniently done with linked list of data rather than arrays.

(iv) The first widely used PL developed for AI application was the functional language LISP (1959)

List Processing

Alexa, deep learning, ML, NLP

(5) Web Softwares :-

JAVASCRIPT or PHP

(i) The World wide Web is supported by wide collection of languages ranging from markup languages ranging from markup languages such as HTML to general purpose PL such as Java. (functionality) (page design

Because of the need for dynamic web content some computation capability is often included in the technology of content preparation

(ii) This functionality can be provided by embedding programming part in an HTML document such part is often in the form of scripting lang. such as

Scanned with CamScanner

02.08.2022
Tuesday

Date / / Page no.

Different Types of Programming Languages :-

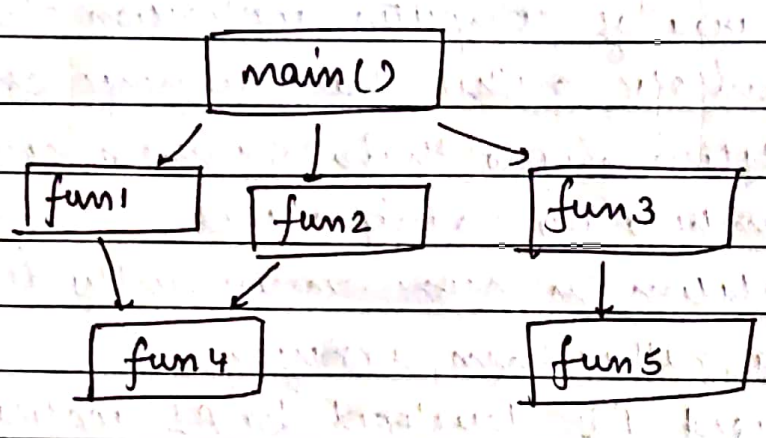
(I) Procedural Oriented Programming :- (POP)

(i) In procedural oriented programming, large program is divided into smaller programs or procedures.

(ii) In POP, a program is written as sequence of functions. Each procedures contains a series of instructions for performing a specific tasks.

(iii) During the program execution, each procedures can be called by other procedures.

To call a procedure, we have to write procedure name only.



Top Down Approach

Stack wise (FILO) or (LIFO)

(iv) The major emphasis of these languages is on Procedure and not on the data.

(v) POPL allows the data to move freely around the system.

(vi) Top Down Program design approach is used by POP.

(vii) Examples :- COBOL, FORTRAN, C, ALGOL, BASIC, etc

(viii) For sharing of data among multiple functions many important data items are placed as global. This makes data moves openly around the system from function to function.

Scanned with CamScanner

Date: / / Page no:

Scope: where we can access variable

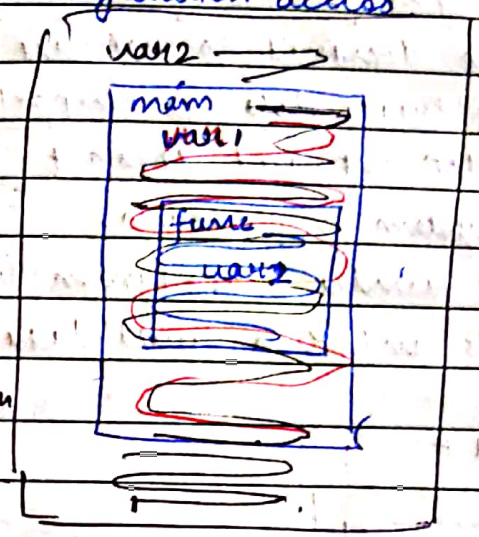
local variable - inside the function access

Lifetime:

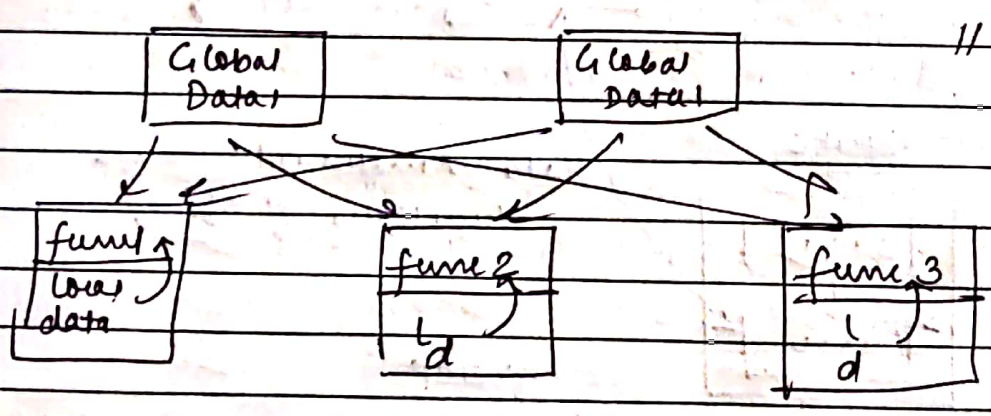
Global - throughout the program

Static - throughout the function

Local - duration function



Function may transform data from one form to another



// Can change value if not required

// Security threat

Disadvantages of POP :-

(i) In POP, Global data can be accessed and changed by any procedure or function. There is no data security. Incase if we want to change type of data of global data then we also ~~then~~ need to revise all functions that access the data. due to this it may happen that some error.

(ii) POP does not model real world problems very well because functions are action oriented and do not correspond to elements of a real ~~value~~ world problem.

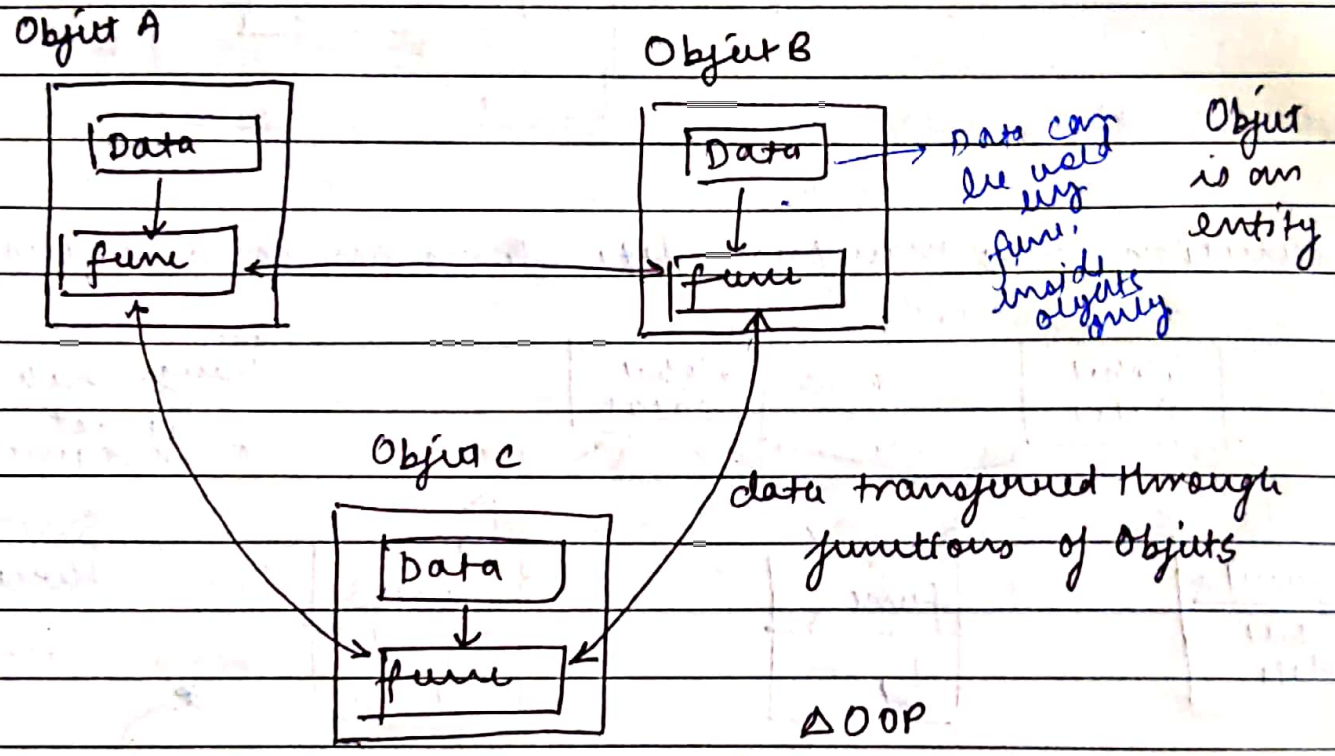
Scanned with CamScanner

(II) Object Oriented Language :- (OOL)

(i) It is designed to remove some drawbacks of POP.

(ii) OOP treats data as important elements in the program development and doesn't allow it to flow freely around the system or program. (emphasis on data)

(iii) OOP divides a program into a number of entities called objects and then build data and functions around the objects.



Use pictorial representation in theory whenever possible.

(iv) Data of objects can be accessed only by the function associated with the object.

(v) Function of one object can access function of other object
Communication through functions

(vi) OOP ties data and functions together and protect it from accidental modification from outside function.

(vii) In OOP, data is hidden or secured and cannot be accessed by external functions.

Scanned with CamScanner

(viii) In OOPs, objects communicate with each other through functions.

(ix) In OOPs, emphasis is on data rather than procedure

(x) OOP follows Bottom Up program design approach
(functions first & then create main)

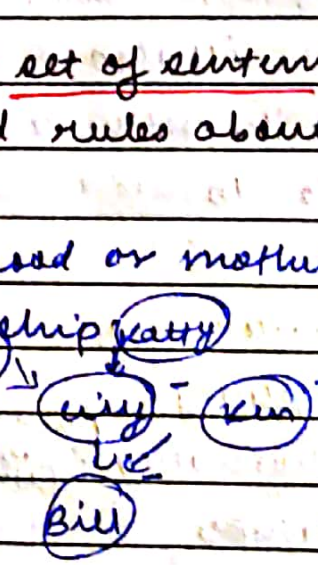
(III) Logic Programming Language (LPL) :-

(i) Logic is the study of formal reasoning based upon statements or propositions.

(ii) Any program written in LPL is a set of sentence in a logical form expressing facts and rules about some problem domain.

Exm. Giving information about fatherhood or motherhood determines grandparent relationship

John is father of Lily.
Katty is mother of Lily
Lily is mother of Bill.
Ken is father of Bill.



Q) Who is the grand mother of Bill
Sol: Katty

(iii) LPL is an example of Rules based language
In Rules Based language, rules are specified in no particular order.

(iv) The Languge Implementation system must choose an order in which the rules are used to produce the desired results.

(v) This approach to software development is totally different from both need with the other 2 category of Langiige and

Scanned with CamScanner

clearly requires a completely different type of languages.

(vi) The most commonly used LPL is

PROLOG

Language Evaluation Criteria :-

int / float

basic building block

(i) The purpose of this subject to examine carefully the underlying concepts of the various constructs and capabilities of programming languages.

(ii) We will also evaluate these features focusing on their impact on the software development process including maintainence.

(iii) To do this we need the set of evaluation criteria.

CRITERIA → cost cost (Security)

Characteristics	Readability	Writability	Reliability
Simplicity	✓	✓	✓
Orthogonality (min. primitive data type)	✓	✓	✓
Data types	✓	✓	✓
Syntax design	✓	✓	✓
Support for abstraction		✓	✓
Expressivity		✓	✓
Type checking			✓
Exception Handling			✓
Restricted aliasing			✓

cost has nothing to say about Characteristics

Readability

(i) One of the most imp. criteria for judging the programming language is the ease with the which programs can be read and

17.08.2022 Wednesday

Chapter-1 ← PPL by Robert Sebesta → 1
C → K&R C C++ Balaguruswami

Date: / / Page no:

understood.

(ii) ⊙ Before 1970s, software development was largely thought of in terms of writing code.

⊙ The primary the characteristics of programming languages was efficiency.

⊙ language constructs were designed more from the point of view of the computer rather than of the computer users.

⊙ After the software lifecycle concept was developed coding was relegated to a much smaller role and maintainence was recognised ~~to~~ as the major part of the cycle, particularly in term of cost.

(iii) Because ease of maintainence is determined in large part of readability of programmes, Readability become an important major in the quality of programs and programing languages.

Characteristics that contributes to the Readability of Programming languages.

(i) Simplicity :- The overall simplicity of Programming language strongly affects it readability

(i) The language with large no. of basic constructs as more difficult to learn than one with smaller no. basic building block

(ii) Programmers who must

2nd complicating feature → Features multiplicity

more than 1 way to accomplish a particular operation

Date / / Page no.

ex: count = count + 1 ;
count + = 1 ;
count ++ ;
++ count ;

(iii) 3rd Potential Problem → Operator Overloading

single operator symbol has more than 1 meaning. Although it is useful but it can lead to reduce readability if user are allowed to create their own overloading and do not do it sensibly.

② Orthogonality :-

Relatively small set of ~~repetive~~ primitive constructs can be combined in a relatively small number of ways to build control & data structures of language.

Furthermore, every possible comb^n of primitives is legal & meaningful.

data type

Primitive Inbuilt (int, float)

Non-Primitive User defined

Linear (linked list, arrays, list)

Non-Linear (tree, graph)

NOTE: We cannot use "void" in structures.

Scanned with CamScanner

Date: / / Page no:

Orthogonality is related to simplicity.
The more Orthogonal the design of the language, fewer no. of exceptions language rules are required.

Fewer exceptions ⇒ Higher degree of regularity in the design which makes the language easier to learn, read & understand.

③ Data Types : The presence of adequate facilities for defining datatypes and data structures in a language is another significant aid to Readability.

for ex. Suppose Numeric type is used for an indicator flag because there is no boolean type in the language.

In C, there is no bool

```
flag = 1 ;    // no diff whether bool or number
```

In such language we might have assignment
flag = 1 ;
meaning of this statement is unclear.
Whereas the language that includes boolean type there would be

```
flag = true ;
```

making it more clear.

Date / / Page no.

(ii) Syntax design : The syntax of the elements of the language has the significant effect on the readability of programme. Program readability are strongly influenced by the forms of language's special words

(for, while, . . .

32 Keywords in C)

specially imp. is the method of forming compound statements or statement groups primarily in control constructs.
Some langige have used matching pairs of special words or symbols to form groups.
indentation when brackets use

FORTRAN (

) end loop

( ) end if

Characteristics that contribute to the writability of programming languages :-

Writability is a ~~major~~ measure of how easily a language can be used to create program for a choosen problem domain.
Most of the language characteristics that affect readability, also affects writability.
~~This follows~~

Scanned with CamScanner

24.08.2022 Wednesday

HW

Date: / / Page no:

[Read 1st Chapter Sicuesta]

(5) Support for abstraction → hiding

function calling by f^n name not any structure of function

(6) Expressivity

program should be readable & understandable.

ex. (i) i = i + 1;

i++; ✓ More comfortable

(ii) for while do while

↓

more comfortable to read

Reliability : program should be executed properly without errors

(7) Type Checking

int var-name

integer ✗

int ✓

Properties of var-name (during compile time)

(8) Exception Handling

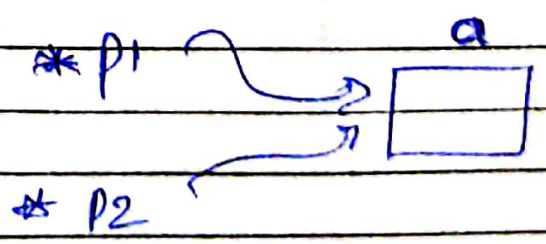
Exception → divide by 0 (known at run time error)

C → does not support this

C++/Java → try and catch

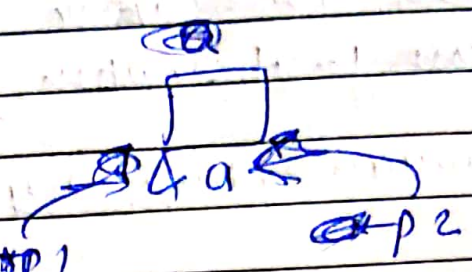
(9) Restricted aliasing :

diff. name for same



Scanned with CamScanner

int *p1 = &a;
int *p2 = &a;



Update in p1
can affect p2
→ & mix - mess
∴ Affect Reliability

Cost (Drawbacks)

① Training for employees.
② Execution of platform
③ Compilation
④ Hardware

Compiler Interpreters.

Scanned with CamScanner

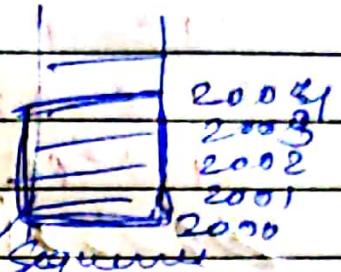
Date. __/__/__ Page no. _____

Scanned with CamScanner

Date ___/___/___ Page no: _____

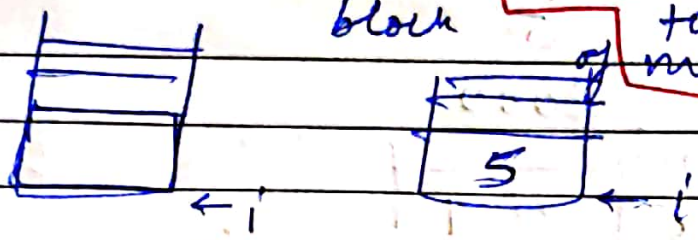
int i; //Reserves 4 blocks of memory

int i=5; //declaration & initialization

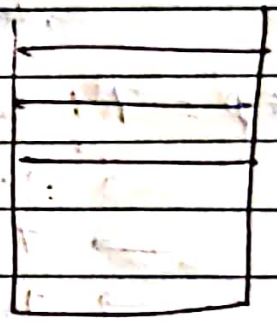


int i = 5; //stores 5 in that block

i will point to first byte of memory = 2000



int i = 6;



defines three things

address value name

POINTER

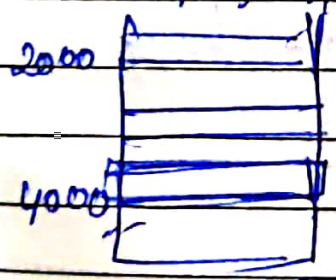
↳ is a variable which stores the address.

(1) * dereferencing operator

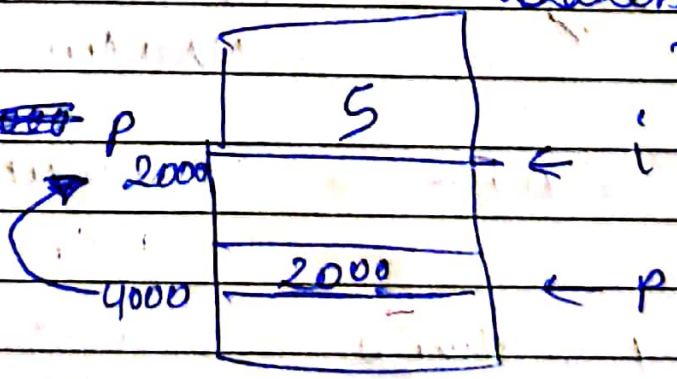
& address of operator

int *p; //declaring ptr

(2) & ⟷ starting address from where to store



int *p = &i;



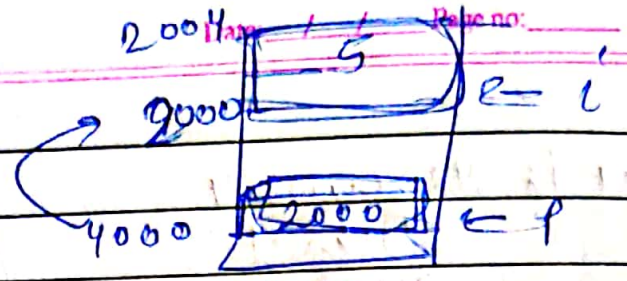
(3) int *p;

↳ points towards int variable

eg) char *p;
int x; p = x; Error

Scanned with CamScanner

int * p;
int * p;
int *p;
} all are correct



*p = &i;

Q) printf("%d", p); → ~~3000~~ 2000 (address of i)

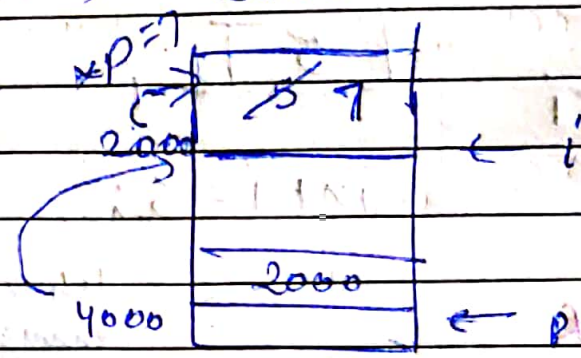
printf("%d", *p); → 5

printf("%d", i); → 5

printf("%d", &i); → 2000

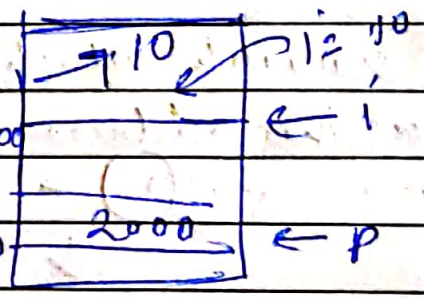
↓ address of

*p = 7;



Q) → 2000
→ 7
→ 7
→ 2000

Q) i = 10 → 2000
→ 10
→ 10
→ 2000



Restricted aliasing → ptr and i can change value both.

int *p;
p = 0 ✓ p = 100 ✗
NULL

sizeof(p) = 4 ; always

struct abc {
};

abc* p;
abc obj;
p = &obj;

27.08.2022
Saturday

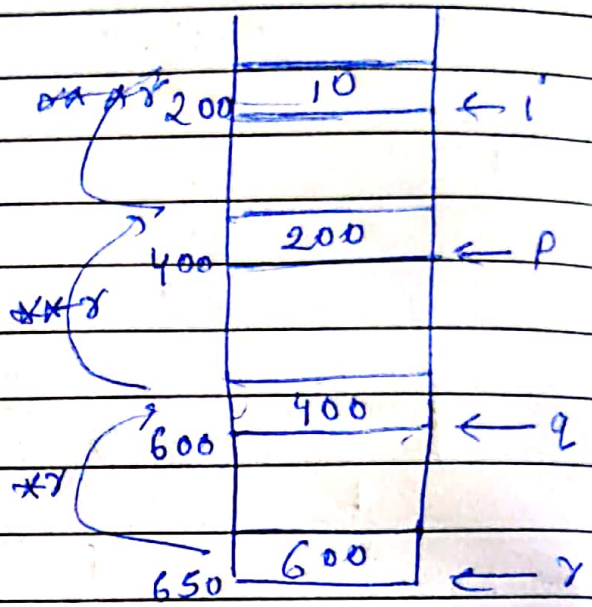
Date: / / Page no:

Pointer of Pointer

```
int i = 10;
int *p = &i;
int **q = &p;
// Storing addr of pointer in pointer

int ***r = &q;
```

if I said I don't want to go to value and put one * then it will give error



NOTE: * * * must be upto depth

printf(... i)	→	10	q →	400
&i	→	200	*q →	200
*p	→	10	***r →	10
*q	→	200		
**r	→	200		
**q	→	10		
p	→	200		

#Using pointer as a function argument :-

```
int main(){
    int a = 10
    increment(a);
    printf("%d", a);
}
```

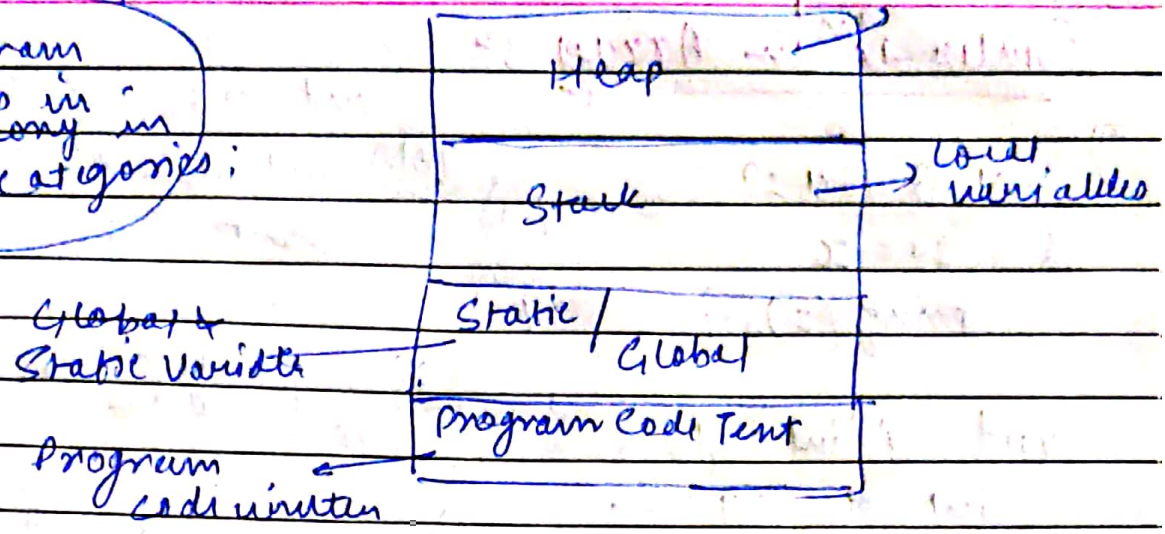
```
increment (int a)
{ a++; }
```

o/p → 10

(malloc, calloc, realloc, free)

Date / / Page no

Program resides in memory in 4 categories:



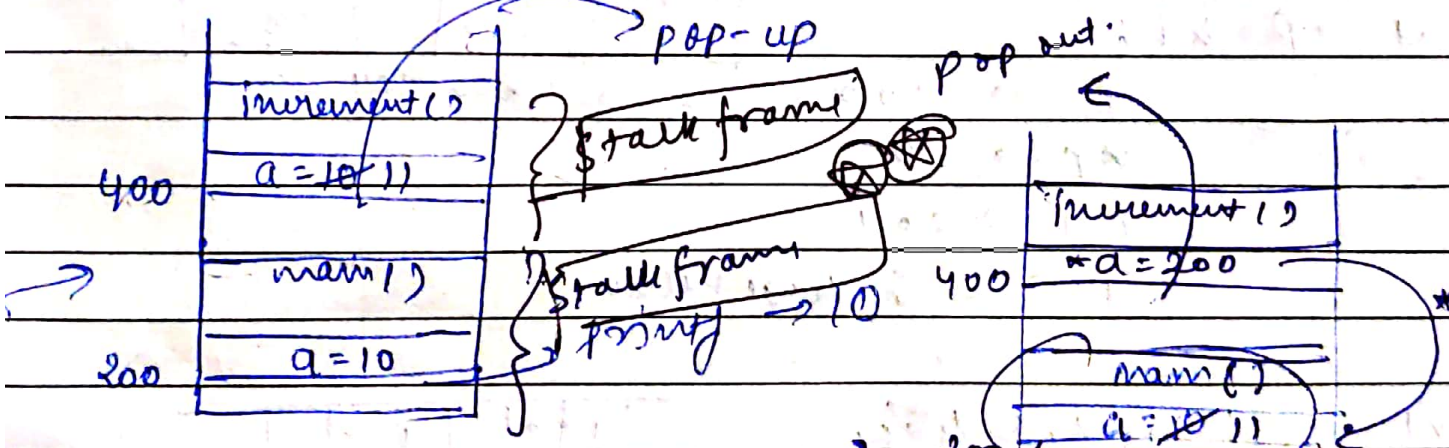
memory will remain in heap until need "free"

This Heap ≠ Heap data structure

This Stack == Stack ds

↳ LIFO Last in First Out

FILO First in Last out




```
int main() {
    int a=10;
    increment(&a);
    printf("%d", a);
}
increment(int *a){
    *a++;
}
```

printf(. .) .

o/p = 11 wrong

precedence of ++ is higher than *

∴ *(a++)

*(204) → Garbage value

29.08.2022 Monday. Put Brackets whenever necessary.

Date: / / Page no. 56

Pointer ~~as an~~ and Array :-

```
int x;   // Global
int main(){
~~int~~ x=5;
p(&x)
printf(x);
```

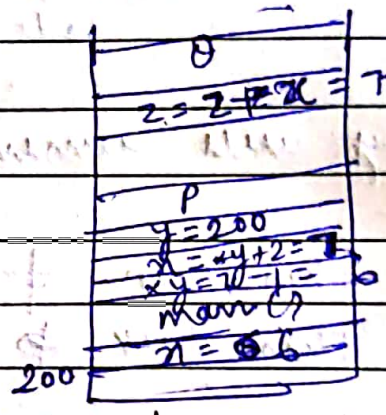
Q) int Q(int z)
{ z+= x
printf(z); }

z → 7 (700)

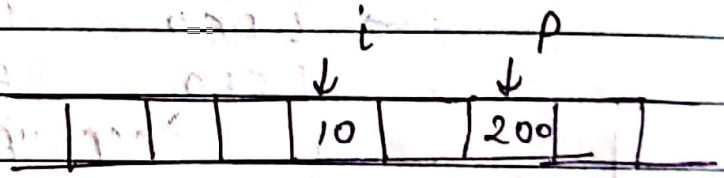
```
void p(int* y){ y → 200 (M.O.)
int x = *y + 2;
Q(x);
*y = x-1;
printf(x);
}
```

x → 7 (500)

12, 7, 6

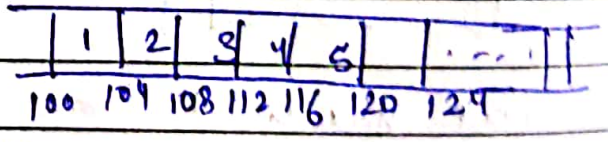



```
int i = 10;
int *p = &i;
```



p → 200
*p → 10
p+1 → 204
*(p+1) → Garbage value

```
int a[5] = {1,2,3,4,5};
int *p = &a;
```



or
int *p = a;

p = 100
*p = 1
p+1 = 104
*(p+1) = 2

$\boxed{a[r] = *(a+r)}$

Array is like ptr ~~so~~.

or int *p = &a[0];

NOTE int *p = a[0]; // Error as ~~it~~ int is assigned to ptr.

Scanned with CamScanner

$\boxed{\&a[0] = a = \&a}$

char → 8 bits | ASCII num 0-255
∴ 28 characters

Date: / / Page no:

Arithmetic operations on pointer of array but not on array.

*(p++) ✓ but *(arr++); ✗

Pointer of strings == Pointer in arrays

char a [5] = {a, b, c, d, e} ;
array of characters.

or char a [5] = "MANIT" ;

M	A	N	I	T	'\0'
0	1	2	3	4	

String ends with Null character.
String initialised by null ;

("%s", p) → MANIT

Q) char c[] = {"GATE2011"};
0 1 2 3
char *p = c;
printf("%s", p + p[3] - p[1]);
base + E - G
2011 -- GATE 2011

A B C D
E F G H.
E - A
65 - 90
69 - 65
= 4
or *a

Arrays as a function argument :-

```
int main() {
    int a[5] = {1,2,3,4,5};
    int total = SOE(a, size)
    int size = sizeof(a)/sizeof(a[0]);
    //Correct
```

20/4 = 5

```
int SOE (int a[], int size) {
    int sum = 0
    for(int i=0; i<size; i++){
        sum += a[i]
    }
    return sum;
}
```

but if here it will

sizeof(a) = 4

Scanned with CamScanner

Coding questions [Explanation + memory diagram + Ans]

Date / / Page no.

Array is passed by Call by Reference not by Value

		Byte
In main()	array is whole	4
In function()	just base address	1

Q) char P[20]
char *S = "string";
int length = strlen(S);
for (int i=0; i < length; i++)
P[i] = S[length - i]
printf("%s", P);

// 6 = length

P[0] = S[6-0] = \0
P[1] = S[6-1] = S[5] =

Sol:

P[0] = S[5-0] = S[5]
P[1] = S[4]
P[2] = S[3]
P[3] = S[2]
P[4] = S[1]
P[5] = S[0]

0	5
1	4
2	3
3	2
4	1
5	0

~~strin~~ ~~nirts~~ gnirts

P[5] = S[1]

P	\0	g	n	i	r	t

s	t	r	i	n	g	\0

O/P = No O/P

NOTE: ① Array does not support Pointer Arithmetic like pointers & other pointers to same array.

② a[0] = *(a+0) = *(0+a) = 0[a]

Scanned with CamScanner

30.08.2022
Tuesday

Date / / Page no.

2D - Array and Pointers

$a[2][3] = \{\{1,2,3\}, \{4,5,6\}\}$

Rows

Columns

Physically there is no partition of memory block.

Logical separation but Physically memory

a →

1	2	3
4	5	6

a →

1	2	3	4	5	6

∴ 2D - array = collection of 1D array.

a[0] → | 1 | 2 | 3 |

a[1] → | | | |

a[m][n] = There are m 1D arrays in which there are n elements in each.

a →

1	2	3	4	5	

100 104 108 112 116 120

0 | | | |

100 104 108 112

1 | | | |

112 116 120

printf("%d", a → 100

a[0] → 100

&a[0] → 100

a[0][0] → 1

&a[0][0] → 100

&a[1][0] → 112

*(a[1]) → 4

NOTE

① Until unless we write both attributes (without *) we can't have value

② a[0] → Base address of 1st 1D array = a = &a

&a[0][0] = &a[0]

a[1] → ——— 2nd ——— = &a[1]

③ a[0][0] = *(a[0]+0) = *(*(a+0)+0)

~~*(*(a+0)+0)~~

Q) a[1][2] in ptr. form

Sol: a[1][2] = *(a[1]+2) = *(*(a+1)+2)

a+1 → 100 + 1*3 → 112

112 → 112 + 2*4 → 120

Pointer to array type

pointer to int

Scanned with CamScanner

int *p = a; // No error but can't access by other way

int pointer → 1D array pointer

~~int *p[] = a;~~

~~int **p = a~~

int (*p)[3] = a; // Pointer of array

↳ no. of columns // link to 2D array

each 1D array has 3 elements

*p[3]

not ~~*p[2]~~

a[m][n]

*p[0] = &a[0]

*p[1] = &a[1]

*p[2] = &a[2]

direct access.

int a[2][3][4] = ~~{1,2,3}, {4~~

int (*p)[3][4] = ~~a[2][3][4~~

P.. 2D array as function arguments

int func (int *a[col])

func (&a);

Q) int f(int *a, int n) {

if (n<=0) return 0;

else if (*a % 2 == 0)

return *a + f(a+1, n-1);

else

return *a - f(a+1, n-1);

}

int main() {

int a[] = {12, 7, 13, ~~14~~, 11, 6};

printf("%d", f(a, 6));

return 0;

}

Scanned with CamScanner

function → readability & reusability

Date / / Page no.

f(a,6)

o/p = 15

6 | 12 % 2 == 0
12 + f(a+1, 5) | 3

7 % 2 == 0
7 + f(a+2, 4) | 4

Stack frame { 13 % 2 == 0
13 + f(a+3, 3) | 9

4 + f(a+4, 2) | 5

1 + f(a+5, 1) | 6

6 + f(a+6, 0) | 0

type of functions
User defined — ex int fun()
Pre defined — ex sqrt()

always program execution starts from main()

function overloading
↳ based on arguments differentiation
① type of arguments.
② No. of arguments.

else same name & same things → Error multiple declaration

functions stack are of limited size
overflow error if size exceeds

03.09.2022
Saturday

Date: / / Page no:

STORAGE CLASSES

variables = Storage Class + data type + var-name
↙
lifetime, scope & memory

① For complete defination of variable one needs to mentioned not only its type but also its storage class.

② We have not defined storage class because it use default.

③ It uses depends on context its uses

④ It tells (i) where the variable can be stored → memory → CPU register
(ii) what will be the default initial value?
(iii) what is the scope of variable? → area of access
(iv) → lifetime → ?

Types of storage Class :-

(I) Automatic :- auto

(i) Storage = memory main (Storage will be in memory)
(ii) Default initial value = Garbage
(iii) Scope = local to the block in which variable is defined.
(iv) lifetime = Till the control remains within the block where variable is defined.

```
main() {
    func() {
    auto int a
    }
}
```

increment & decrement in loop

scope & lifetime

(v) "auto" keyword is used. auto int i;

(II) Register Storage Class :- register

(i) Storage = CPU Register

Scanned with CamScanner

Date / / Page no:

register int i = 0

(ii) Default initial value = Garbage

(iii) Scope = Local to the block where it is defined

(iv) Lifetime = Till the control remains within the block.

Diff b/w (I) & (II) CPU Register Access Time is less than Main memory

(b) CPU Register value can accessed faster.

NOTE CPU Registers are limited and of fixed size

If memory is not there it will store as 'Auto' automatic

(III) Static Memory Class static

(i) ~~Storage~~ = memory static int i = 0

(ii) Default initial value = 0

(iii) Scope local to the block in which variable is defined.

* (iv) Lifetime = value of variable persists b/w different function calls.

Q) func() {

~~sta~~ auto/static int i = 1;

i++;

printf("%d", i); }

int main() {

func(); func(); func(); func();

}

Sol: (a)

	auto	static
i =	2	2
i =	2	3
i =	2	4
i =	2	5

Static can't be initialised again & again

In 2nd func() call, it will not execute static int i = 1

05.09.2022
Monday

Date / / Page no.

(IV) External Storage Class extern extern

(i) Storage = memory
(ii) Default initial value = 0
(iii) Scope = Global
(iv) Lifetime = As long as program execution does n't come to end.
(v) extern keyword is used and declared outside the all functions.

STRUCTURE

(1) Primitive data types
(2) Collection of Homogeneous elements → arrays
(3) Collection of Heterogeneous elements → structures

(1) In programming collection of Heterogeneous type of data can be grouped to form a structure. This entire collection can be referred by a structure name.
(2) The individual components which are fields or members can be processed separately.
(3) Each field of a structure must have its own data type which can be either simple or structured data type.
(4) A structured data type also permits to declare one or more structures as a member of another structure.
(5) When a structure is declared as a member or field of another it is called as a Nested structure. → Structure inside structure

```
struct <name>
    { data_type member1;
      data_type member2;
      :
      data_type member n;
    };
```

Scanned with CamScanner

$ptr \to x == (*ptr).x = (*a).x$
$== a.x$

Date: / / Page no:

an:

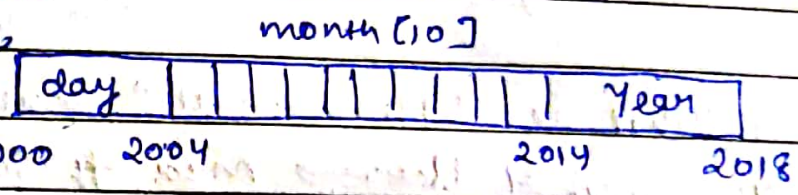
~~Q~~ struct date { int day;
char ~~int~~ month[10];
int year;
};

struct date curr; // can be declared here

data-type

curr

Memory Representation



Accessing data type of curr

① curr . day ; // dot operator
② curr → day ; // Arrow operator

Copying structure

✗ curr1 = curr2 ; ✗ Not directly

At field level curr1 . day = curr2 . day ; ✓
curr1 . ~~date~~ year = curr2 . year ; ✓

Nested structure

struct name { } struct date {
int day;
struct name x;
int year;
} curr1, curr2;

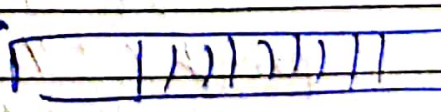
Nested structure itself

struct date {
int day;
struct date x; }

Pointer & Structure :

struct date *curr2 = & curr1;

curr1



curr2

if incremented by 1 it get incremented by 18 bytes

Structure is used to get direct accessing

Date / / Page no.

→ When memory is concern

Union :-

sizeof (union) = sizeof(max.(individual datatype))

union data {

can access only one element

};

Q) What is difference betn union & structure?

Sol: size of Union → max of size

Structure → Sum of sizes of all dt.

Gate

Q)
```
struct Test {
        int i;
        char *c;
};

struct Test s[ ] = { {5, "become"},
                     {4, "better"},
                     {6, "Jungle"},
                     {8, "ancestor"},
                     {7, "brother"}};

main ()
{ struct Test *p = s;
    p+ = 1;                 // p = p+1
    ++p → c;   //better   //Jungle
    printf ("%s", p++ → c);   //(Jungle) (better)
    printf ("%s", *++p → c);   // ancestor brother
    prnif ("%d", p[0].i);   //
    printf ("%s", p → c);   // address (brother)
};
```

p → 200

p = p+1 → 400

p++ → 600

Scanned with CamScanner

precedence (arrow) > precedence (++)

Date: / / Page no:

5	200	4	300	6	400	8	500	7	600
100	105		110		115		120		

p+=1 ; p → ~~100~~ 105 ++p → c

300 301

b	e	t	t	e	r \0

either
u
6
uncle

p++ → c position after this

p → ~~105~~ 110

400 401

Jungle

Gate CS 2000

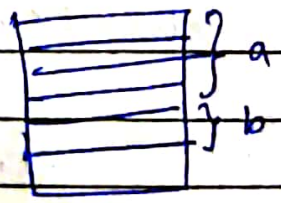
```
struct node { int i; float j; };
struct node *s[10];
```

Sol: precedence [ ] > precedence *

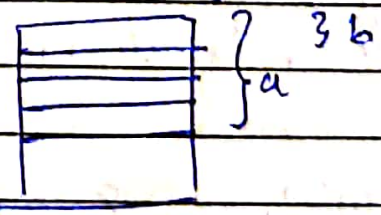
*s[10] — s[10] array. s is an array, each element of which is a pointer to structure of type node.

Unions:- union is a user defined data type but unlike structures, union members share same memory locations.

```
struct abc {
    int a;      // M1
    char b; };  // M2
```




```
union abc {
    int a;      // M1
    char b; };  // M2
```



Facts: ① union members share same memory location. If we make changes in one member then it will be reflected to other members as well.

Scanned with CamScanner

Date / / Page no

Q) union abc {
int a;
char b;
} var;

```
int main() { var.a = 65;
printf("a = %d \n", var.a);
printf("b = %c \n", var.b);
return 0; }.
```

Sol: a = 65
b = A

Size of Union = Size of the largest member of the union

Size of (union abc) = 4

Accessing Members using Pointers;
Use (→) arrow operator (like in structures)

Application of Union:-

① #pragma pack(1) → no padding

When we want to store info about shirt & book in shop.

```
union {
    struct {

    } book;
    struct {

    } shirt;
} item;
```

② Making array of different data type.
padding & packing in structures.

Q) struct { double a; union { char b[4]; // 4 double c; // 8 int d; } e; // 4 char f[4]; } s;

no 'holes' betn members 8 + 8 + 4 = 20

Find size

Sol:

Q) change struct ⟷ union

Find size of (s)

Sol: max (8, 8, 4) = 16

Scanned with CamScanner

Structure Padding :-

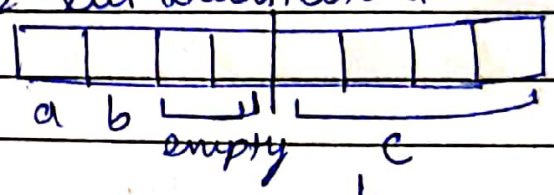
① Processor does not read 1 byte at a time from memory. It reads 1 word at a time.

② If we have 32 bit processor then it means it can access 4 bytes at a time which means word size is 4 bytes. 32/8

③ If → 64 → 8 bytes.

```
struct abc {
    char a;
    char b;
    int c; } var;
```

In 32 bit architecture

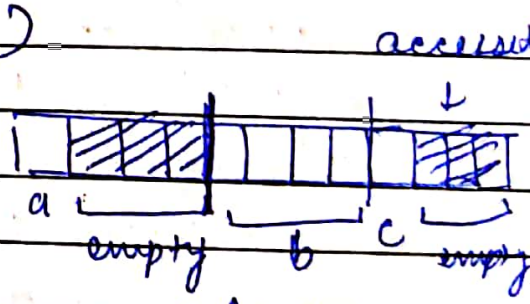


so that c can be accessed in one cycle

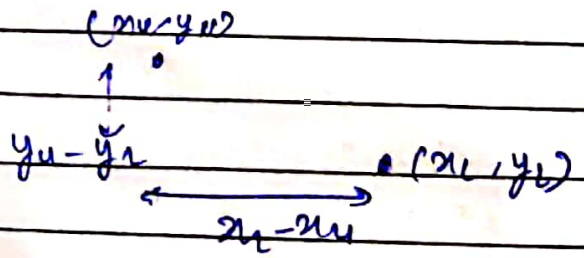
∴ Size of (var) = 8 bytes (and not 6)

If we change order.

```
struct abc {
    char a;
    int b;
    char c; } var;
```



Size ↑ser → 12 bytes



06.09.2022
Tuesday

Date / / Page no.

Object Oriented Programming

emphasis on data rather than procedures

OOPs is a software development philosophy based on the following central ideas:-

1. Class
2. Objects
3. Data Abstraction
4. Data Hiding
5. Data Encapsulation
6. Dynamic Binding
7. Message Passing
8. Inheritance
9. Polymorphism

friend function

① **Class:-** It is a user defined data type which can hold both data and functions.

(i) The internal data of class is called Member data

(ii) The internal function of a class is called member function

(iii) The member functions of a class are mostly manipulate the internal data.

(iv) The variables of a class are called objects or instance of a class.

(v) Every object is a collection of data members and the associated member functions

(vi) The general syntax of a class is:

```
class <class-name>
{ Private:
    Member data        → int a;
                         float cost;
    Member function    → increment();
                         add();
  Public:
    Member data
    Member function
};
```

Scanned with CamScanner

st Unlike Struct, Class can be assigned to each other

ClassA = Class B

Class even being user defined is make in the form of predefined (in built)

③ **Data Abstraction** :- It refers providing only essential information to the outside world and holding their background details.

④ **Data Hiding** :- It is technique of hiding the internal members of an object.

In C++, the class construct allows to declare data and functions as a public, private and protected group.

⑤ **Data Encapsulation** :- It is a mechanism of binding the data and function ~~that~~ and use them in a single unit class.

→ Capsule → wrapping up of data & function in capsule class.

③ function call ( )	④ private → no one access that variable outside the class
↓	Access through only class data & function
get ans;	
* don't focus on inside fn	
ex. sqrt( ) → returns ans;	

↳ No focus on what is there in sqrt function code only ans.

⑥ **Dynamic Binding** :- Binding refers to the linking of a procedure call ~~call~~ to the code to be executed in response to the call.

Dynamic Binding means that code ~~that~~ associated to the ~~with~~ given procedural call is not known until the time of the call at runtime.

It is also known as late Binding. It is associated with Polymorphism & Inheritance.

Scanned with CamScanner

Date / / Page no

Compile Time Binding

Static Binding (early Binding) → known at the time compilation {before execution}

Dynamic Binding (Late Binding) → until execution we don't know where to go for code in memory.

↳ Supported by C++

↳ Example : function Overloading
① either argument diff.
② type of argument diff.

function Overriding (Inheritance, virtual fn)

⑦ **Message Passing** :- Objects communicate with one another by sending and receiving information much the same way as people pass message to one another.
The concept of message passing makes it easier to talk about building system that directly model or simulate their real world counter part.

⑧ **Inheritance** :- It is the process by which object of one class acquires the properties of another class.
(i) It provides the idea of reusability. This means that we can add additional features to an existing class without modifying it.

(iv) functions vs prev. Class +
(v) Same code multiple times — other things with it if necessary

(vi) ex. Base class Cars → four tyres, steering, Gear box
↳ Now each model will have atleast these properties + other
like Tesla + (Auto drive + protection + Bullet proof window)
Mercedes + (Power window + Power steering)

Rebbal
Know it make new class if you have base class just add it through inheritance

(C++ structure can have fn too.)

(Backward compatibility of C++ whatever C supports, C++ will also support)

~~opt: by default public~~ can be made private

Date: / / Page no:

(9) Polymorphism :- Polymorphism is a greek term means the ability to take ~~many~~ more than one forms

(i) An operation may exhibit different behaviour in different instances.

(ii) The behaviour depends upon the type of data used in the operations.

one name = many forms

at different instance perform differently.

[+ operator → int a + int b = c (int)
→ complex a + complex b = complex c
No. No. No.]

diff arg. Operator overloading

diff. Type of arg. Function Overloading → func()
→ func(int)
→ func(char)
→ func(int, int)

[one name, diff. call.]

(*) Why Class when we have Structures?

(1) struct complex
{ float x; float y; } c1, c2, c3;
c3 = c1 + c2 ; // Not allowed ✗

To give more feel of inbuilt data type ~~we~~, created Class

Structures does not support arithmetic operations of struct variables

(2) Do not permit data variables.

(i) So C++ / OOPs try to bring user defined data types as close as possible to the builtin data types.

(ii) By default member of the class are Private while member of structure are public.

Scanned with CamScanner