COMPUTER PROGRAMMING

- Lab (2% per lab) 10 x 2% total (Best 10)
- Assignment (≈ 3 hrs/week, 2.5%) 6 x 5% (2 week assn)
- Total Suggested Os/week: 3 + 2 + 2 + 2 = 9
  Lab · class? · Assn · Practise
- Mid-Sem: 8% + 12% (20%)
  written + Lab
- End-Sem: 10% + 20% (30%)
  written + Lab

__/__/____

*] INTRODUCTION TO C: [CASE SENSITIVE]

1] '//' : comment (ignored by compiler)

2] '#include<library>' : include std. libraries for input/output

3] 'int main()
{ : execution starts inside this f^n named main.
}'

"/*" = multiple line comment "*/"

Example:
```
#include<stdio.h>
int main()
{ printf("Hello World\n");
   Syntax (print formatted)
   return 0;
}
```

(#) After nano main.c to convert it into binary that the 'genie' understands : [gcc main.c -o main]

→ Now it is executable using : [./main]

o] Makefile:

→ to reduce redundancy of gcc & ./main 'Makefile' is used.

⇒ nano Makefile → default name (not changeable)

```
run:
    gcc main.c -o main
    ./main
```

→ Now to run main.c only command required : [make run]

*] CONSTANTS: (Boolean, Character, Integer, real, complex, string)

→ by default printf does not print integers, decimals, etc.

⇒ For integer : printf("%d", 1); ⇒ 1

⇒ For float : printf("%f", 15.6); ⇒ 15.6

Ex: printf("%4f", 15.6); → 15.6000

Ex: printf("%d, %d, %d", 1, 3, 5); ⇒ 1, 3, 5

→ data values that cannot be changed during execution

__/__/____

*] DATATYPES:

o] Integer:

→ C type: 'int' → 32 bits

→ Generally, arithematic operatⁿ used on them

Ex: printf("circumference is %f"; 2*3.14*5);

o] Float (Real Numbers):

→ C type: 'float' → 32 bits

→ Arithematic functions can be formed

o] Character:

→ C type: 'char' ↷ used when individual character separate from string is wanted

Ex: printf("This is a charachter", 'a');

*] IDENTIFIERS:

- Unique names assigned to data & objects.
- Each identifier is stored at a unique address.

o] Rules:

1] First charachter must be alphabetical or underscore '_'.

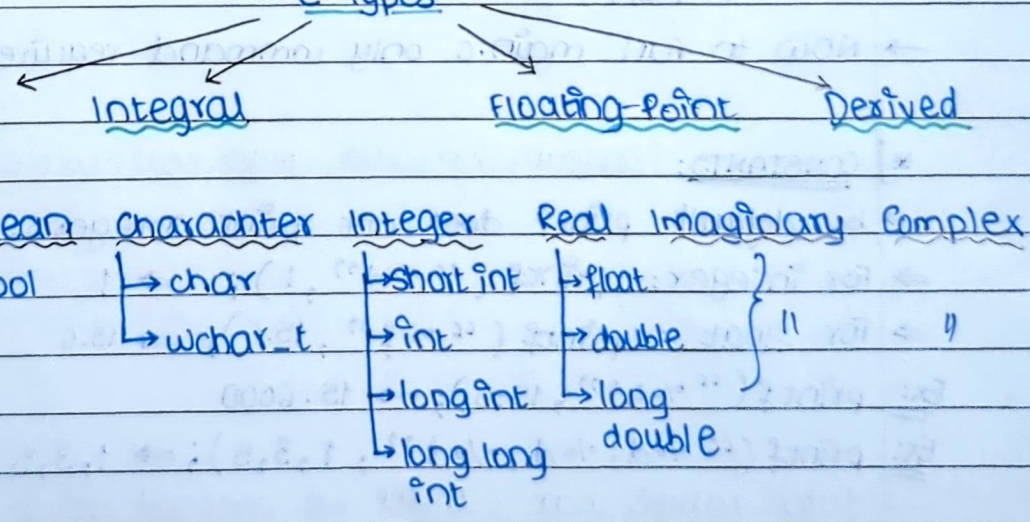
2] Must contain only alphanumeric charachters or '_'.

3] The first 63 charachters of an identifier are sufficient.

4] can not duplicate a keyword.

Ex: my_name ✓ , 1_name x , I-am-Yoda x

#



__/__/____

*] SYMBOLIC NAMES FOR CONTROL CHARACHTERS:

1] Newline: '\n'
2] Horizontal tab: '\t'
3] Vertical tab: '\v'
4] Backspace: '\b'
5] Carriage return: '\r'
6] Form feed: '\f'
7] Alert(bell): '\a'
8] Backslash: '\\'
9] Single quote: '\''
10] Double quote: '\"'
11] Question mark: '\?'
12] Null charachter: '\0'

*] SCANF():

- Function reads data from the standard input stream.
- Reads from left to right.

Ex:
```
int a=5;
scanf("%d", &a);
```
(L-value → a; ~~R-value~~ → 5; &a → Address of variable)

Ex:
```
int age;
printf("Enter your age: ");
scanf("%d", &age);
```
→ use required conversion characters

*] OPERATORS:

- Follows BODMASS

1] Parentheses: ()
2] Postfix operators: ++, --
3] Unary operators: +, -, !, ~, ++, --
4] Multiplicative -u-: *, /, %
5] Additive: +, -
6] Relational operators: <, >, <=, >=
7] Equality operators: ==, !=
8] Logical AND: &&
9] Logical OR: ||
10] Assignment: =, +=, -=, ... → always last

*] TOKENS:

1] Keywords: if, else, while
2] Identifiers: int obj, float anoy
3] Constants: 3, 10, 5
4] Strings: "Hello"
5] Symbols: ~, %, !

__/__/____

NOTE: Code overwriting, i.e. if a variable has been assigned multiple values in a code then the last value of that variable is assumed.

HW Q] Figure out why a=a+b-(b=a) works for swap.

→ most probably cause it reads from left to right and variable assignment stays confined until complete execution of that line. (1=2 doesn't occurr with = for BODMASS logic)

*] WHILE LOOP:

Syntax: while (expression)
{ // write statement here }

Ex:
```
int n;
int digit;
while (n>0) {
    digit = n%10;
    n = n/10;
    printf("%d", digit);
}
```

*] FOR LOOP:

Syntax: for (initial condn; condn; increment)
{ // write statement here }

Ex:
```
for (a=10; a<20; a++)
{
printf("%d", a);
}
```

For checking factors of n you only need to check upto $\sqrt{n}$ as factor $< \sqrt{n}$ covers complementary factor $> \sqrt{n}$ and hence more efficient

__/__/____

*] Increment Operators:

→ '++' = +1

⇒ Ex: a++ = post increment, ++a = pre increment

Imp: → a=5;
b=a++;
printf("%d",b);
} Gives output 5 as due to post increment: b=a then a++

→ a=5;
b=++a;
printf("%d",b);
} output = 6

BUT: a=5 a++ & a=5 ++a } without an identifier have same meaning

→ Ily, '--' = -1

*] Relational Operators:

→ '==' - To check conditional equality

→ '!=' - not equal to

→ '>=', '<='

*] Bitwise Operators:

→ '&' = logical and

Ex: 6&7 ⇒ $(110)_2$ & $(111)_2$

⇒ 1 1 0
1 1 1
1 1 0 ← ⇒ Bitwise

→ Ily,

→ '|' = logical or

→ '^' = logical xor

*] If loop:

Syntax:
```
if (condition){
    //statement;
}
```

__/__/____

*] IF ELSE / IF ELSE IF loops:

Syntax:
```
if (condn) {
   // Statement;
}
else {
}
```

Ily,
```
if ( ) {
}
elseif {
}
```

*] Bitwise operations:

- Generally has 8 bit signed representation.
- 1's compn : flips all digits, 2's compn : 1's + 1

Q] Represent (-9) in 1's complement.

→ +9 = 1001 ⇒ -9 ka 1's = 0110

2's = 0111 = -9

Q] Why 2's complement for -ve no.s?

→ The advantages of the 2's complement over the signed system are that addn & subtractn are implemented without having to check the sign of the operands, and that the 2's complement has only one representn of zero, unlike the signed system.

o] Checking Odd/Even using Bitwise operatn:

1] XOR

Ex: 1011 ^ 0001 = 1010 ; 1110 ^ 0001 = 1111

⇒ ^ with 1 ⇒ +1 if even, -1 if odd

2] AND

Ex: 1011 & 0001 = 0001 ; 1110 & 0001 = 0000

⇒ & with 1 ⇒ 1 if odd, 0 if even

3] OR

Ex: 1011 | 0001 = 1011 ; 1110 | 0001 = 1111

⇒ | with 1 ⇒ +1 if even, same if odd

o] Using AND to check if a number is a power of 2:

Ex: 00010000 (#) If n is a power of 2, n & (n-1) = 0

→ 10000 − 00001 = 01111 ∴ 10000 & 01111 = 00000 ⇒ power of 2

__/__/____

*] Bitshift operator:

- 'x<<n' & 'x>>n'
  left shift right shift

Ex: p=15 , p<<2 ⇒ 00001111 to 00111100

basically, for left shift $x \times 2^n$

lly, for right shift $x / 2^n$

p=15 , p>>2 ⇒ 00001111 to 00000011

*] Q] Swap numbers using xor.

→ ① a = a^b ② b = a^b ③ a = a^b

b = (a^b)^b ↳ (a^b)^a = b

becomes 0 a^a = 0

Q] Find min. of two numbers using Bitwise & comparison operator

→ r = y^ ((x^y) && - (x<y))

If x<y ⇒ y^((x^y) && - (1))

2's = 11111111

⇒ y^((x^y)) = x

*] Alternate syntax for IF-ELSE:

→ expr 1 ? expr 2 : expr 3

If Else

⇒ If expr1 then expr2 , else expr 3.

*] Switch case:

- Alternative to if-else if ladder.

Syntax:

```
switch (xyz)
{
 case abc: printf("def");
           break;
 case ghi: printf("jkl");
           break;
 default : printf("N/A");
           break;
}
```

If no case matches

⇒ As evident not as convinient as if-else if & also can't handle multiple conditions.

__/__/____

*] **Break Statement:**

- used to exit a recurring loop according to our needs.

Ex:
```
for(i=0,i<10,i++)
{
   for(j=0,j<10,j++){
      if(i+j=15){
         break;
      }
   }
}
```
Goes inside the if 4 times for i=6,7,8,9

*] **Continue Statement:**

- skips all statements after it and restarts the loop it is a part of. "control transferred to end of loop"

*] **Null Statement:**

Ex1:
```
int d=5;
if(d==0); → {} ke jagah ; makes the if a Null Statement
   printf("Division"); printing Division without accepting if, with a warning while compiling that if has an empty body.
```

Ex2:
```
int i=10;
while(--i>0);
{printf("%d",i);}
return 0;
```
→ outputs 0 as the condition is still run just that { } isn't inside while

*] **Type casting:**

- A process of converting one datatype into another.

a] **Implicit Type Casting:**

Ex:
```
short a=10;
int b;
b=a;
```
converts short data type to int without explicitly mentioning it.

__/__/____

◦] Explicit Type Casting:

```
int a = 15, b = 2;
float div;
div = (float) a/b;
```

ensures float division

*] Arrays:

- A collection of elements of same datatypes stored at continuous memory locations having a fixed memory size.

Syntax: `int arr[n];`

name ↵ ↳ no. of elements (n×4 bytes memory occupied as int)

→ Indexing: arr[0] = 1st element, arr[1] = 2nd element & so on.

Ex: `int arr[5] = {2,7,8,10,15};`

◦] Scanning elements into an Array:

```
for(i=0; i<5; i++){
    scanf("%d", &a[i]);}
```

`int arr[] = {1,2, . . . . n}` creates array of length n+1 without having explicitly specified its size.

◦] Printing an array:

```
for(i=0; i<n, i++){
    printf("%type", arr[i]);
}
```

- If using a variable make sure its defined before use [var].

*] Scope of variables:

- Variables can only be accessed to the extent of validity of their definition.

◦] Static Variable:

- lifetime is entire run-time of program.

◦] Global Variable:

- visible throughout the program (generally defined at top)

◦] Local variable:

- Referenceable only in the function or block which it is declared in.

__/__/____

6] Scanning a 2-D Array:

→
```
for(i=0; i<5; i++){
    for(j=0; j<4, j++){
        scanf("%d", &a[i][j]);
    }
}
```

*] Pointer technique to solve questions related to Arrays:

- Doesn't actually use 'pointers' (taught later)
- Basically start from a[0] & a[n] & come towards centre.

Ex:
```
start=0, end=(n-1)
while (start < end){
    temp = a[start];
    a[start] = a[end];
    a[end] = temp;
    start++, end--;}
```
} Swaps pairs of array

*] Leaders in an array:

- An element is a leader if it is greater than all elements to its right.

⟹
```
int arr[] = {21,16,17,4,6,3,5,2}
int n = sizeof(arr)/sizeof(arr[0]);
int max_from_right = arr[n-1];
printf("%d", max_from_right);
for (int i=n-2; i>=0, i--){
  if (max_from_right < arr[i]){
      max_from_right = arr[i];
      printf("%d", max_from_right);
  }
}
printf("\n");
}
```

__/__/____

*] Initializing a 2-D array:

- The same array can be initialized in various ways:

① int a[2][3] = { 1,6,10, 4,15,7 }
row 1 row 2

② int a[2][3] = { {1,6,10}, {4,15,7} }
rows specified

③ int a[ ][3] = { {1,6,10}, {4,15,7} }
↳ not required, assigned based on {{},{}....}

→ 1 6 10
4 15 7

→ 2-D arrays can be thought of as a collection of 1-D arrays (useful later in array of pointers, etc.)

HW Try printing &arr[i].

→
```
int a[3][5] = {{1,2,3,4,5},{2,3,4,5,6},{3,4,5,6,7}};
for (int i = 0; i < 3; i++){
    for (int j = 0; j<5 , j++){
        printf("%d", &arr[i][j]);
    }
}
```

Output: 1087629696 ____ 700 ____ 04 ____ 08 ____ 12
____ 16 ____ 20 ____ 24 ____ 28 ____ 32
____ 36 ____ 40 ____ 44 ____ 48 ____ 52

→ separated 4 as array of type int, sizeof(int) = 4 bytes

→ These ~~hexa-decimal~~ values refer to the decimal representation of the memory address of the values.

__/__/____

*] Row Major Order & Column Major Order:

Ex: char arr [10][15]
Say base address of a[0][0] = 100
Find address of a[8][6]?

→ Column Major order ⇒ 168 (count columnwise)
Row Major order ⇒ 226 (count Rowwise)

*] Sorting in Arrays:

o] Selection Sorting: $O(n^2)$

- Sets a minimum & compares to it ⇒ sorted & unsorted sub arrays

⇒
```
int n=10, a[]={3,2,6,5,4,7,8,9,10,1}, min_index;
for (int i=0; i<n-1, i++){
   min_index=i;
   for (int j=i+1; j<n, j++){
      if (a[min_index]>a[j]){
         min_index=j;}
   if (min_index!=i){
      int temp=a[i];
      a[i]=a[min_index];
      a[min_index]=temp;}
   }
```

o] Bubble Sort:

- Compares 2 elements from left to right

⇒
```
for (int i=0; i<size-1; ++i){ int swapped=0;
   for (int j=0; j<size-i-1; ++j){
      if (arr[j]>arr[j+1]){
         int temp=arr[j];
         arr[j]=arr[j+1];
         arr[j+1]=temp;
         swapped=1;}}
   if (swapped==0) break;
}
```

__/__/____

◦] Insertion Sorting: $O(n^2)$

Ex: 1 3 5 7 (4)

```
for(i=1; i<n ; i++){
key = arr[i];
j = i-1;
while( j>=0 && key < arr[j]){
  arr[j+1] = arr[j];
  j--;
}
arr[j+1] = key;
}
```

*] Pointers in C-Programming:

Ex: `int a = 42;` (stored at 083857; value 42)

`int *ptr;` → used to define an integer pointer
ily, float, char, double, etc.

`ptr = &a;`

`printf("%d", a);` → 42

`printf("%p", ptr);` → 083857

`printf("%d", *ptr);` → 42
↳ In such syntax used to dereference memory address, i.e. give the value stored.

→ `*ptr = 50; printf("%d", a);` → gives 50

◦] Pointers and Arrays:

- The name of an array is a constant pointer to the address of the first element of the array. → for a 2D it refers to the 0^{th} 1-D array → arr+1 = 1^{st} row

Ex: `int arr[10];` ∴ arr ≡ &arr[0].
↳ name

- `arr[i] = *(arr + i)` → basically arr[0+i] (goes to i^{th} element)
as it is a const. can't do arr++
↳ 1 address shift differs by (x) bytes
↳ depends on datatype
4 for int ⇐ etc.

for 2-D array address of 0^{th} element i^{th} row

&arr = pointer to the whole array, however the output is same as &arr[0], but &arr+1 will refer to element after the array.

__/__/____

◦] Pointer to an array:

Syntax: `int (*a)[n];` can't do `int *ptr;`

```
int b[n] = {1, 2, ...., n};        int b[n];
a = &b;                            a = &b;
```

⇒ To print elements `*(*a+i)` or `(*a)[i]`:

`*a = b` ∵ `a = &b`, `b+i = &b[i]` ⇒ `*&b[i] = b[i]`

◦] Array of pointers:

Syntax: `int *a[n];`

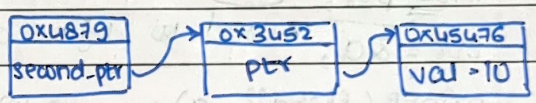
```
int b[n] = {1, 2, ...., n};
for (i=0; i<n; i++){                } → Assignment
  a[i] = &b[i]; }
for (i=0; i<n; i++){
  printf("%d", *a[i]); }
```

◦] Pointer to a pointer: [Double Pointer]

Syntax:

```
int var = 10;
int * ptr = &var;
int ** second_ptr = &ptr;
```

≡



→ Ily pointer-ception can be made (n stars for n^{th} pointer)

→ print ** second_ptr would give var.

NOTE: • Pointers in 2-D array:

- `*(arr+i) + j` = address of `a[i][j]`
- `arr[i][j]` = `*(*(arr+i) + j)` → Put in 2 for loops to print

*] Memory Allocations (Malloc):

#	Static	Dynamic
Memory size:	Once allocated, it is fixed	Size is changable.
Storage:	Stored in a data structure called stack	Stored in a data structure called heap.
Allocat^n time:	During compiling	During running of code (Run-Time)

__/__/____

1] malloc function:

→ malloc(n) (n → bytes) returns a pointer to the start of the memory of the type "void".

→ To use malloc, a library named <stdlib.h> has to be included.

Ex: Writing a code to allocate memory to an array.

→
```
#include<stdio.h>
#include<stdlib.h>
int main(){
int* arr;
arr = (int*)malloc(sizeof(int)*5);
for(int i=0 ; i<5 ,i++){scanf("%d",arr+i);
                        {printf("%d",*(arr+i));
printf("\n");
return 0;
}
```

- ↗ = dynamic memory allocation of array.

2] Malloc and Array of pointers:

```
- for(i=0 ; i<2 (→m) ,i++){
    arr[i] = (int*)malloc(sizeof(int)*3 (→n));
  }
```

for → int arr[m][n]

3] Malloc and Pointer of pointer:

```
- int r=3,c=4,i,j,count=0;
int** arr = (int**)malloc(r*4);
for(i=0,i<r,i++){
 arr[i]=(int*)malloc(c*4);}
for for  a[i][j]=++count;
for for printf("%d",(*(arr+i)+j));
printf("\n");
for(int i=0;i<r;i++){
  free(arr[i]);
  free arr;}
}
```

```
int arr[] = {11,22,33};i;
int* ptr[3];
for(i=0 ; i<3,i++){
  ptr[i] = &arr[i];
}
for(i=0;i<3,i++){
  printf("value of arr[%d] =
          %d\n",i,ptr[i]);
}
```

__/__/____

*] Functions:

- 2 parts: Return type & name
  Ex: int ↙ main() ↖

Syntax: return-type func-name(input parameters){
  local variables;
  * return [ ]; → whatever want
}

→ define func before main().
OR
→ declare the function before main:

```
int sum(int,int); → declaration
int main(){
}
int sum(int a,int b){ → definition
}
```

CALL BY VALUE

Ex: Swapping two numbers:

→ void swap(int a, int b) → no return
{ temp = a;
  a = b;
  b = temp;
}
int main()
{ int x=5, y=10;
  swap(x,y)
  print(x,y) → gives 5,10 as ouput
}

→ What happens is swap happens locally inside the swap function & thus calling x & y outside swap gives unswapped values

CALL BY REFERENCE:

```
void swap(int* a, int* b){
temp = *a;
*a = *b;
*b = temp;}
int main(){
swap(&x,&y);}
```

calls upon the pointers

__/__/____

→ // Calculating sum of 1-D array elements by passing it in a fn

```
→ float calcSum(float num[]){
    float sum = 0.0;
  for (int i = 0; i<n; ++i){
    sum += num[i];}
  return sum;}
```

⇒ for num[] = {23.4, 55}
result = calcSum(num)

→ // for 2-D array (works only if your compiler is C99 compatible)

```
void print(int m, int n, int arr[][n]){
 int i,j;
 for{for {printf("%d",arr[i][j]);}
    printf("\n");}
int main(){
int arr[][3] = {{1,2,3},{4,5,6},{7,8,9}};
int a=3,b=3;
print(a,b,arr);
}
```

→ must be passed before 2-D array

*] Recursions:

Ex: Factorial:

```
int factorial(int n){
if (n==0)
   return 1;
else
  return n*factorial(n-1)
}
```

Ex: Fibonacci Sequence:

```
int fib(int n){
if (n==1) → return 0;
else if (n==2) → return 1;
else return fib(n-1) + fib(n-2);
}
```

→ Recursion Trees are used to map out recursions, consider the following example:

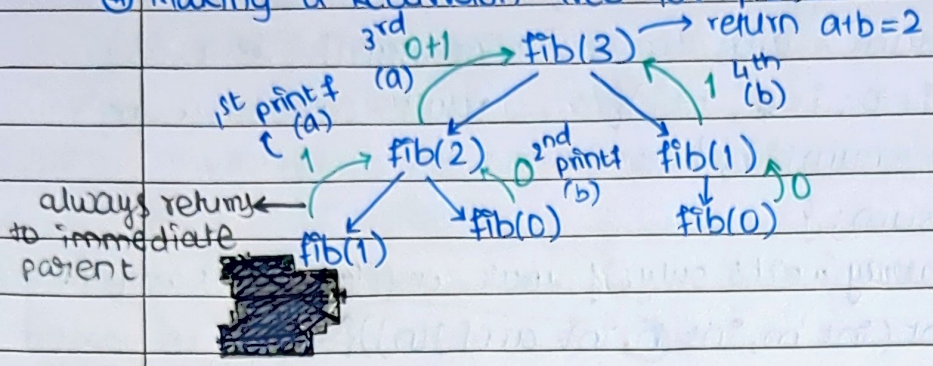
Ex:

```
int fib(int i){
int a,b;
 if(i==0){return 0;}
 if (i==1){return 1;}
 a = fib(i-1);
 printf("%d\n",a);
 b = fib(i-2);
 printf("%d\n",b);
 return (a+b);}
```

P.T.O.

__/__/____

Making a Recursion Tree for fib(3):



→ Output: 1 0 1 0 1 2

NOTE:
- string = character array
- ⇒ #include<string.h>
- ⇒ char num[n];
- ⇒ scanf("%s", num); → scans string into array

String Recursion:

```
→ void permute(char str[], int l, int r)
{ if (l == r){
      printf("%s\n", str);
}
  else{
     for(int i = l ; i<=r ,i++){
        str[i] = str[l] - (str[l] = str[i]);
        permute(str, l, r);
        str[i] = str[i] + str[l] - (str[l]=str[i]); //BACKTRACKING
     }
  }
}
```

⇒ Backtracking: abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid soln.

MID SEMESTER UNTIL HERE

Second half taken by Professor Girish Varma

__/__/____

*] Structures:

- Allows you to create custom datatypes.

Syntax: struct [name] {
    // parameters
};
↳ Imp

Ex:
```
struct Student {
    char name[100];
    char email[100];
    float marks;
};
```

inside main()
```
struct Student S = {
    "Raju",
    "raju@student.iiit.ac.in",
    25.5
};
```

o] Printing Structures:

- printf("Student Details:\n Name:\t %s\n Email:\t %s\n Marks:\t %f\n", S.name, S.email, S.marks);

o] Multiple Entries in a Structure:

Ex:
```
struct Student class[2] = {
    {"Raju", "raju@email", 25.5},
    {"Ammu", "ammu@email", 45.0}
};
```

o] Selective Input:

- say omitting email
  {.name = "Raju", .marks = 25.5}

o] Updating Input:

- class[i].<identifier> = <updated value> → won't work for char array as LHS = arr & RHS = string constant

  Solved by using string copy ←

NOTE: o] strcpy (string copy func) used to assign new values to char arr
#include <string.h>
→ strcpy(class[0].name, "Ramu");

* strlen only calculates length until first \0, including whitespaces [space]

→ All pointers have same size
y 32 bit processor → 4 Bytes
y 64 bit processor → 8 Bytes

__/__/____

∘] Typedef:
- To avoid rewriting stuff tediously we use type def.

Syntax: typedef [Actual] [New];
Ex: typedef struct Rectrangle Rect;

NOTE: (*r.height) ≡ r→height (minus greater than) used for dereference pointer '.'

*] Debugger:
- Every programming language has an inbuilt debugger.

Syntax: gcc -g main.c -o main.o
gdb main.o
(gdb) r (r to run)

∘] Setting a Breakpoint:
- (gdb) b (x) → line to which code is to be run.
→ func name also works
(#) (gdb) n : next line

Multiple break pts can be set
;c to cont. to next one

∘] Printing stuff:
- (gdb) p [stuffidentifier]

*] Enumeration:
- Numbers starting from 0
- Typedef required

Ex:
```
enum Weekday {
    Sunday,
    ;
    Saturday,
};
```

underlying type int

```
enum Weekday day = Wednesday;
print (day + 2)
⇒ 5 ∵ (Wednesday = 3)
```

∘] Modifications:
1] Say above Sunday = 1,
then 1, 2, . . . , 7
2] Say Monday = 5 & Thursday = 10,
then 0, 5, 6, 7, 10, 11, 12, 13

__/__/____

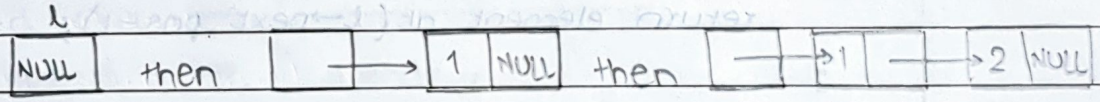
*] **Macros:** → convention to use capital letters

Syntax: #define [NAME] [VAL]

- The advantage is that there's no confusion b/w which value is being referred to, in case there are more than one of the same values.
- It does not use extra memory, the compiler just does 'find & replace'.

*] **LinkedLists:**

Ex:
```
typedef struct Node{
    int data;
    struct Node* next; → recursive struct defs have to be done using struct pointers
}Node;
typedef Node* LinkedList;
int main(){
  LinkedList l = NULL; //empty LinkedList
  Node A = {1, NULL};
  Node B = {2, NULL};
  l = &A; // l is a size 1 LinkedList
  l->next = &B; // l is a size 2 LinkedList
}
```

⇒ 

o] **Printing a LinkedList:**

```
- void print_list(LinkedList l){
    if(l != NULL){
      printf("%d", l->data);
      print_list(l->next);
    }
}
```

P.T.O.

__/__/____

- Finding size of LinkedList:

```
int size_list(LinkedList l){
  if (l==NULL){
    return 0;
  }
  else{
    return 1 + size_list(l->next);
  }
}
```

OR

```
int size_list(LinkedList l){
  return l==NULL? 0 : 1 + size_list(l->next);
}
```

while doing l-> ensure that l != NULL, if l is NULL then l-> will give seg fault

- Returning ith element:

→ Index starts from 0.

→

```
int element_at(LinkedList l, int pos){
  if (l != NULL){
    if (pos == 0){
      return l->data;
    }
    else{
      return element_at(l->next, pos-1);
    }
  }
  else{
    return -1;
  }
```

- Append elements:

```
LinkedList append(LinkedList l, int data){
  Node * n = (Node*) malloc(sizeof(Node));
  n->data = data;
  n->next = NULL;
```

P.T.O.

__/__/____

```
if (l == NULL){
    return n;
}
else{
    LinkedList i = l;
    while (i->next != NULL){
        i = i->next;   = ++ing in linkedlists
    }
    i->next = n;
    return l;
}
}
```

*] Control Flow:

- Examination of possible paths a program can take during its execution.

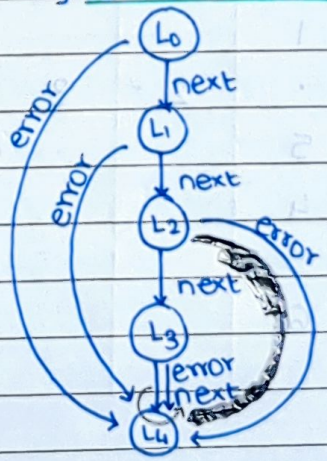
•] Structural Abstraction:

Ex:
0 a = 4 ⇒ 0 expression assignment
1 b = read() ⇒ 1 ——— 4 ———
2 c = a || b ⇒ 2 ——— 4 ———
3 d = c - a ⇒ 3 ——— 4 ———
4 # end ⇒ 4 # end

•] Control Transfer Functions:

i	next	error
0	1	4
1	2	4
2	3	4
3	4	4
4		

•] Control Flow Graphs:



→ To draw one with implicit error edges, remove error wale connections

__/__/____

- Structurally Feasible Executions:

- Execution: valid path from L_0 to L_n.

Ex: $L_0 \to L_1 \to L_2 \to L_3 \to L_4$; $L_0 \xrightarrow{error} L_4$, $L_0 \to L_1 \xrightarrow{error} L_4$, etc.

- Logically Feasible Executions:

- b≠0:

$\Rightarrow L_0 \to L_1 \to L_2 \to L_3 \to L_4$

- b=0:

$\Rightarrow L_0 \to L_1 \to L_2 \xrightarrow{error} L_4$

- b not number:

$\Rightarrow L_0 \to L_1 \to L_2 \xrightarrow{error} L_4$

- Actual Execution:

→ one instance of a Logically Feasible Executions ⊆ Structurally Feasible Executions.

Ex: b=5

- Conditional Control Flow:

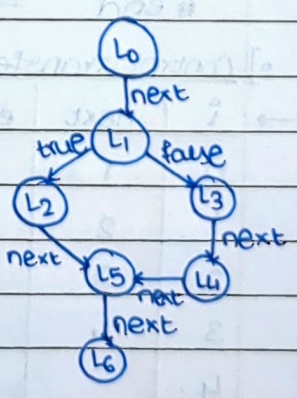
→

0	a=read()	⇒	0 expression assignment
1	if a<5:	⇒	1 if
2	[tab] b=7	⇒	2 exprⁿ assⁿ
3	else:	⇒	3 else
4	[tab] b=2*a	⇒	4 exprⁿ assⁿ
5	c=a+b	⇒	5 ——u——
6	# end	⇒	6 # end

→

i	next	true	false	error
0	1			6
1	·	2	3	6
2	5			6
3	4			6
4	5			6
5	6			6
6				

⇒



error edges implicit

__/__/____

→ 1] a<5: $L_0 \to L_1 \to L_2 \to L_5 \to L_6$

2] a≥5: $L_0 \to L_1 \to L_3 \to L_4 \to L_5 \to L_6$

3] a not num: $L_0 \to L_1 \xrightarrow{error} L_6$

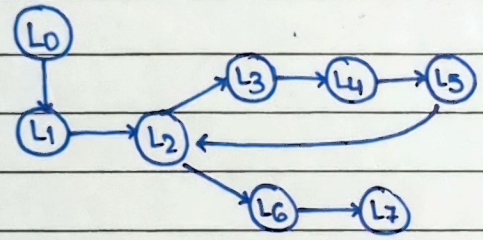
⇒ a=3; b=7 → c=10

$L_0 \xrightarrow{next} L_1 \xrightarrow{next} L_2 \xrightarrow{next} L_5 \xrightarrow{next} L_6$

*] Iterative control flow:

			i	next	true	false	error
→ 0	i = read()	⇒ 0 exp^n ass^n					
1	a = 1	⇒ 1 ——u——	0	1			7
2	while i > 0 :	⇒ 2 while :	1	2			7
3	a = a*i	⇒ 3 exp^n ass^n	2	·	3	6	7
4	i--	⇒ 4 ——u——	3	4			7
5	continue	⇒ 5 continue	4	5			7
6	x = a	⇒ 6 exp^n ass^n	5	2	[redacted]		
7	# end	⇒ 7 #end	6	7			7

Error edges implicit ⇒



→ i>0 : $L_0 \to L_1 \to L_2 \xrightarrow{true} L_3 \to L_4 \to L_5 \to L_2 \to L_6 \to L_7$

i≤0 : $L_0 \to L_1 \to L_2 \xrightarrow{false} L_6 \to L_7$

i not [redacted] num: $L_0 \to L_1 \to L_2 \xrightarrow{error} L_7$

*] Procedural Control Flow:

- similar to functions in mathematics

→ 0	def square(x):	⇒ 0	def square :
1	y = x*x	⇒ 1	exp^n ass^n
2	return y	⇒ 2	return
3	a = 4	⇒ 3	exp^n ass^n
4	b = square(a)	→ 4	call square assignment
5	c = a+b	⇒ 5	exp^n ass^n
6	# end	→ 6	#end

(x: formal, first)

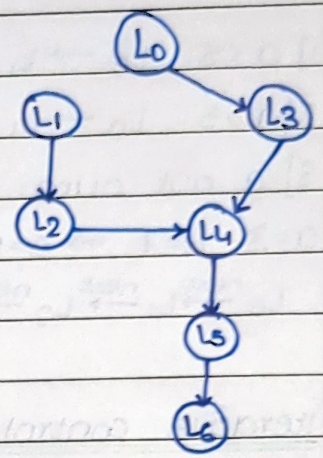
P.T.O.

__/__/____

→

i	next	call	return	error
0	3			[redacted]
1	2			[redacted]f
2	[redacted]		{4}	[redacted]6
3	4			[redacted]6
4	5	0		[redacted]6
5	6			[redacted]6
6				

⇒



(Error edges implicit)

→ Make logically feasible executions accordingly