

COMPUTER PROGRAMMING

- Lab (2% per lab) $10 \times 2\% = 20\%$ total (Best 10)
- Assignment (≈ 3 hrs/week, 2.5%) $6 \times 5\% = 30\%$ (2 week assn)
- Total Suggested Qs/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()
=multiple fine comment { : execution starts inside this f" }
} named main.

Example: #include<stdio.h>

```
int main()
{
    printf("Hello World\n");
    // Syntax (print formatted)
    return 0;
}
```

① After nano main.c to convert it into binary that the 'genie' understands : [gcc main.c -o main]

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

② Makefile:

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

→ nano Makefile → default name (not changeable)

run: x86_64-softmmu/qemu-system-x86_64 ~ anna.vm ~

```
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

*] DATA TYPES:

•] Integer:

- C type: 'int' → 32 bits
 - Generally, arithmetic operat" used on them

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

Float (Real Numbers):

- C type : ‘float’ → 32 bits

- Arithmetic functions can be formed

Character:

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

*] IDENTIFIERS:

- Unique names assigned to data objects

- Each identifier is stored at a unique address.

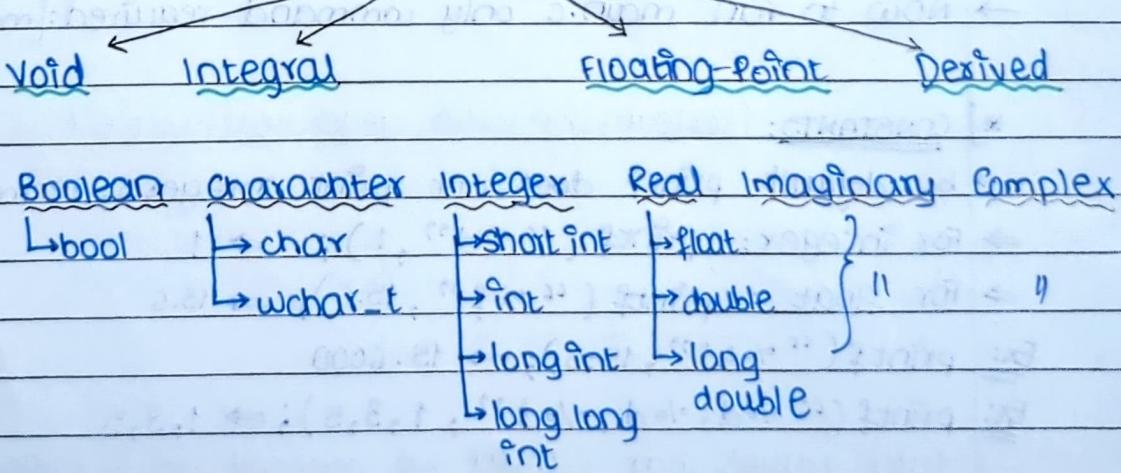
o] Rules:

- 1] First character must be alphabetical or underscore '_.'
 - 2] Must contain only alphanumeric characters or '_.'
 - 3] The first 63 characters of an identifier are sufficient.
 - 4] can not duplicate a keyword.

Ex: my-name ✓, I-name ✗, I-am-Yoda ✗



C Types



* SYMBOLIC NAMES FOR CONTROL CHARACTERS:

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

* SCANF():

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

Ex:
int a = 5;
scanf ("%d", &a);
 ^ Address of variable

Ex:
int age;
printf ("Enter your age: ");
scanf ("%d", &age);
 ^ use required conversion character

* OPERATORS:

- follows BODMAS

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

* TOKENS:

- 1] Keywords: if, else, while.
- 2] Identifiers: int obj, float array
- 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.

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. ($i=2$ doesn't occur with z for BODMAS 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: → $a = 5;$ } Gives output 5 as due to post
 $b = a++;$ } increment: $b = a$ then $a++$
 $\text{printf}(" \%d", b);$

→ $a = 5;$ }
 $b = ++a;$ } Output = 6
 $\text{printf}(" \%d", b);$

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

→ Ily, '--' = -1

* Relational Operators:

→ '==' - To check conditional equality

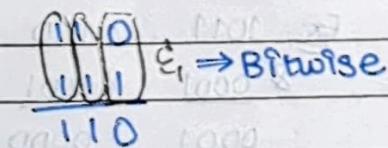
→ '!=' - not equal to

→ '>=', '<='

* Bitwise Operators:

→ '&' = logical and

Ex: $6 \& 7 \Rightarrow (110)_2 \& (111)_2$

Ex: $6 \& 7 \Rightarrow$ 

→ Ily,

→ '|' = logical or

→ '^' = logical xor

* IF loop:

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

*] IF ELSE / IF ELSE IF LOOPS:

Syntax: if (condn) { illy, if () {
 // statement;
}
}
else if {
}
else {
}

*] Bitwise operations:

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

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

$$\rightarrow +9 = 1001 \Rightarrow -9 \text{ ka } 1's = 0110 \\ 2's = 0111 = -9$$

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.

Q) Checking Odd/Even using Bitwise operatn:

1] XOR

2] AND

3] OR

$$\begin{array}{r} \underline{\text{Ex:}} & 1011 & 1110 \\ & \underline{\overset{\wedge}{0001}} & \underline{\overset{\wedge}{0001}} \\ & 1010 & 1111 \end{array}$$

Ex: 1011 1110
& 0001 & 0001
0001 0000

$$\begin{array}{r} \text{Ex: } \\ \underline{\underline{1}} \quad \underline{\underline{0111}} \quad \underline{\underline{1110}} \\ | \quad \underline{0001} \quad | \quad \underline{0001} \\ 1011 \quad 1111 \end{array}$$

$\Rightarrow \wedge$ with $1 \Rightarrow +1$ if even $\Rightarrow \&$ with $1 \Rightarrow 1$ if odd $\Rightarrow |$ with $1 \Rightarrow +1$ if even
 -1 if odd 0 if even same if odd

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

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

$$\Rightarrow 10000 \quad \therefore 10000$$

$$\begin{array}{r} -00001 \\ \hline 01111 \end{array}$$

& 01111

00000 \Rightarrow power of 2

*] Bitsift operators:

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

Ex: $p=15$, $p \ll 2 \Rightarrow 00001111$ to 00111100

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

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

$p=15$, $p>12 \Rightarrow 00001111$ to 00000011

★ Q] swap numbers using XOR.

$$\rightarrow \textcircled{1} a = a^{\wedge} b \quad \textcircled{2} \quad b = a^{\wedge} b \quad \textcircled{3} \quad a = a^{\wedge} b$$

$$b = \underbrace{(a \wedge b)}_0 \wedge b \rightarrow (a \wedge b) \wedge a \rightarrow b$$

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

$$\rightarrow r = y \wedge ((x \wedge y) \& - (x < y))$$

$$1^{\text{st}} \quad x < y \Rightarrow y^{\wedge}((x^{\wedge}y) \triangleq \underline{(1)})$$

$$\Rightarrow y \wedge ((x \wedge y)) = \underline{x} - z (a = b) ??$$

*] Alternate syntax for IF-ELSE:

→ expr1 ? expr2 : expr3
 If Else

$\Rightarrow \text{if } \text{expr}^1 \text{ then } \text{expr}^2, \text{else } \text{expr}^3.$

*| switch case:

- Alternative to if-else if ladder.

Syntax: switch(xyz)

1

case abc: printf("def");
break;

\Rightarrow AS evident not as

case ghi : printf ("ijkl");

convenient as if-else &
& also can't handle
multiple conditions.

break;

default : printf ("N/A");

break;

If no user →
matches

matches

*] 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:

Ex: `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.`

Ex: `int i=10;`

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

*] Type casting:

- A process of converting one datatype into another.

o] Implicit Type Casting:

Ex: `short a=10;`

`int b;`

`b = a;`

} converts short data type to int without explicitly mentioning it.

o] 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 ($n \times 4$ 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};

o] Scanning elements into an array: for (i=0; i<5; i++) { scanf("%d", &arr[i]); }

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

o] Printing an array:

- for (i=0; i<n; i++) {

 printf("% type", arr[i]);

}

- ~~variable~~ If using a variable make sure its defined before arr[i].

*] Scope of variables:

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

o] static variable:

- lifetime is entire run-time of program.

o] Global variable:

- visible throughout the program (generally defined at top)

o] Local variable:

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

•] 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]; swaps posns. of array
 a[end] = temp;
 start++, end--;} }

*] Leaders in an array:

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

```

 $\Rightarrow$  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 $\text{int } a[2][3] = \begin{bmatrix} 1, 6, 10 \\ 4, 15, 7 \end{bmatrix}$

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

③ $\text{int } a[1][3] = \{ \{ 1, 6, 10 \}, \{ 4, 15, 7 \} \}$

↳ not required, assigned based on $\{ \{ \dots \}, \{ \dots \} \dots \}$

→ 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.)



Try printing $\&arr[i][j]$

→ $\text{int } arr[3][5] = \{ \{ 1, 2, 3, 4, 5 \}, \{ 2, 3, 4, 5, 6 \}, \{ 3, 4, 5, 6, 7 \} \};$

$\text{for (int } i = 0; i < 3; i++) \{$

$\text{for (int } j = 0; j < 5, j++) \{$

$\text{printf(} "%d", \&arr[i][j]) \};$

}

}

Output:

108 7629696 700 04 08 12

16 20 24 28 32

36 40 44 48 52

→ separated by as array of type int , $\text{sizeof(int)} = 4$ 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 $\rightarrow 168$ (count columnwise)

Row Major Order $\rightarrow 226$ (count Rowwise)

*] Sorting in Arrays:

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

- Sets a minimum & compares to it \rightarrow 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;  
    }
```

1] Insertion Sorting: $O(n^2)$

Ex: $1 \ 3 \ 5 \ 7 \ 4$

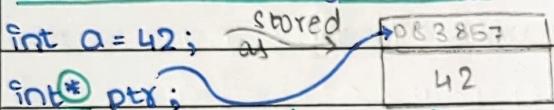
```

→ 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;



int* ptr; → used to define an integer pointer

ptr = &a; → my, float, char, double, etc.

printf ("%d", a); → 42

printf ("%d", ptr); → 083857

printf ("%d", *ptr); → 42

→ In such syntax used to ~~get~~ dereference memory address, i.e. give the value stored.

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

2] 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 0th 1-D array

Ex: int arr[10]; ∴ arr ≡ &arr[0]. → arr + i = 9th row

- arr[i] = *(arr + i) → basically arr[0+i] (goes to ith element)

- arr[i] = *(arr + i) as it is a const. can't do arr++ → 1 address shift differs by \times bytes

for 2D array
address of 0th element in ith row

depends on
4 for int ← datatype
etc.

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

o] 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]`:

$\star a = b \therefore a = \&b, b+i = \&b[i] \rightarrow \star \&b[i] = b[i]$

o] 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]); }`

o] Pointer to a pointer: [Double Pointer]

Syntax: `int var = 10;`

`int * ptr = &var;` = `0x4879` → `0x3452` → `0x45476`
Second.ptr ptr val = 10

`int ** second_ptr = &ptr;`

→illy pointerception can be made (n stars for nth 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 loops to print

* Memory Allocations (~~malloc~~):

#

Static

Dynamic

memory size: Once allocated, it is fixed

size is changeable

Storage: stored in a data structure called stack

stored in a data structure called heap.

Allocat' time: During compiling

During running of code (Run-Time)

•] malloc function:

→ malloc (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.

•] Malloc and Array of pointers:

- `for (i=0 ; i<2 , i++) {`

`arr[i] = (int*) malloc (sizeof(int)*3);`

`}`

•] Malloc and Pointer of pointer:

- `int r=3, c=4, i, j, count=0;`

`int **arr = (int**) malloc(r*c);`

`for (i=0, i<r, i++) {`

`arr[i] = (int*) malloc(c*c);`

`for (j=0, j<c, j++) {`

`arr[i][j] = ++count;`

`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};`

`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

}

Ex: CALL BY VALUE
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 output

}

What happens is swap happens

Locally inside the swap function

E. thus calling x & y outside

swap gives unupdated values

CALL BY REFERENCE: void swap(int* a,int* b){

temp=*a;

*a=*b;

*b=temp;

calls upon the pointers

int main(){

swap(&x,&y);

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

```
void print(int m, int n, int arr[m][n]){
    int i, j;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            printf("%d", arr[i][j]);
        }
        printf("\n");
    }
}

int main () {
    int arr[3][3] = {{1,2,3}, {4,5,6}, {7,8,9}};
    int a = 3, b = 3;
    print(a, b, arr);
}
```

* 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);
}
```

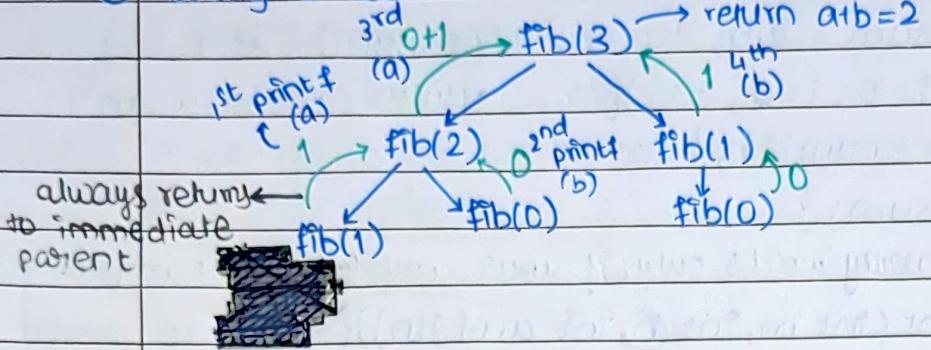
→ Recursion Trees are used to map out recursions, consider the following example:

```
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[r] - (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.iit.ac.in",

25.5

};

o Printing Structures:

- printf ("Student Details:\n Name: %s\n Email: %s\n Marks: %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

s.name = "Raju", s.marks = 25.5 }

o Updating Input:

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

solved by using string copy

E R.H.S = string constant

NOTE: o strcpy (String copy func) used to assign

#include <string.h>

new values to char arr

→ strcpy (class[i].name, "Ramu");

④ 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 Rectangle Rect;`

NOTE: $(\ast r.\text{height}) \equiv r \rightarrow \text{height}$ (minus greater than)
used for dereference pointer (' \ast ')

*] 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:

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

- `(gdb) b @` → line to which code is to be run
junc name also works

- `(gdb) n`: next line

•] Pinning stuff:

- `(gdb) p [stuff identifier]`

*] Enumeration:

- Numbers starting from 0

- `typedef` required

Ex: `enum Weekday {`

underlying
type int
Sunday,
Saturday,

Monday, Tuesday, Wednesday, Thursday, Friday}

`enum Weekday day = Wednesday;`

`print (day + 2)`

$\Rightarrow 5$: (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; } Node;
typedef Node* LinkedList;` → recursive struct defns have to be done using struct pointers

`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
}`

⇒ `NULL` then `→ 1 NULL` then `→ 2 NULL`

*] Printing a linkedlist:

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

•] 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);
}

•] Returning i^{th} 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;
```

```

if (l == NULL) {
    return n;
}
else {
    LinkedList i = l;
    while (i->next != NULL) {
        i = i->next;
    }
    i->next = n;
    return l;
}

```

* Control Flow:

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

o Structural Abstraction:

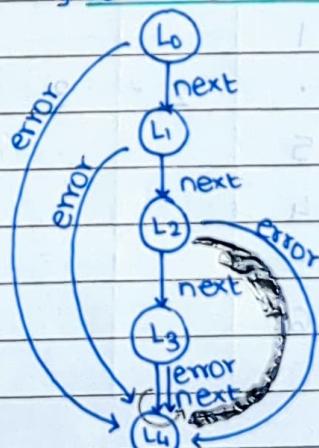
Ex:

- 0 $a = 4 \Rightarrow$ expression assignment
- 1 $b = \text{read}() \Rightarrow$
- 2 $c = a \& b \Rightarrow$
- 3 $d = c - a \Rightarrow$
- 4 #end \Rightarrow #end

o Control Transfer Functions:

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

o Control Flow Graphs:



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

◦ Structurally Feasible Executions:

- Execution: valid path from L_0 to L_n .

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

◦ Logically Feasible Executions:

- $b \neq 0$:

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

- $b = 0$: ~~at least one of points~~ $\rightarrow L_0 \rightarrow L_1 \rightarrow L_2 \xrightarrow{\text{error}} L_4$

- b not number:

$\Rightarrow L_0 \rightarrow L_1 \rightarrow L_2 \xrightarrow{\text{error}} L_4$

◦ Actual Execution:

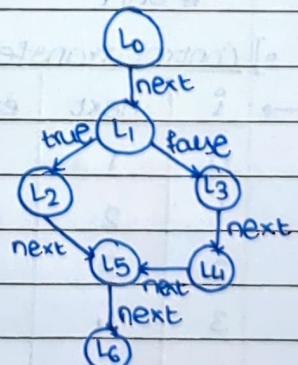
→ One instance of a logically feasible executions ⊆ structurally feasible executions.

Ex: $b = 5$

◦ Conditional Control Flow:

→ $a = \text{read}()$ \Rightarrow expression assignment
 $'\text{if } a < 5:$ \Rightarrow $'\text{if}$
 $'\text{if } b = 7$ \Rightarrow $'\text{exprn assn}$
 $'\text{else:}$ \Rightarrow $'\text{else}$
 $'\text{if } b = 2 * a$ \Rightarrow $'\text{exprn assn}$
 $'\text{c} = a + b$ \Rightarrow $'\text{exprn assn}$
 $'\text{\# end}$ \Rightarrow $'\text{\# end}$

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] $a < 5$: $L_0 \rightarrow L_1 \rightarrow L_2 \rightarrow L_5 \rightarrow L_6$

2] $a > 5$: $L_0 \rightarrow L_1 \rightarrow L_3 \rightarrow L_4 \rightarrow L_5 \rightarrow L_6$

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

→ $a=3$; $b=7 \rightarrow c=10$

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