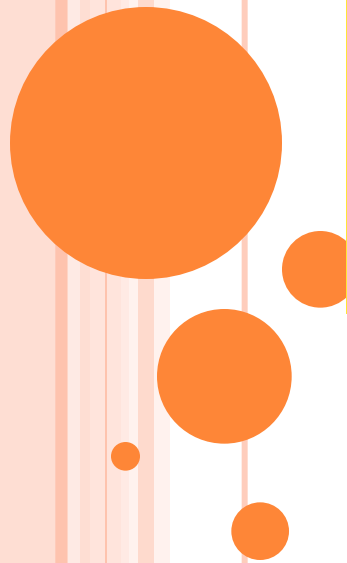


C-ARRAYS AND FUNCTION



ARRAYS

- An array is a collection of elements of the same type that are referenced by a common name.
- Compared to the basic data type (`int`, `float`) it is an aggregate or derived data type.
- All the elements of an array occupy a set of contiguous memory locations.
- Why need to use array type?
- Consider the following issue:
 - "We have a list of 1000 students' marks of an integer type. If using the basic data type (`int`), we will declare something like the following..."
 - `int studMark0, studMark1, ...studMark999`



- Can you imagine how long we have to write the declaration part by using normal variable declaration?

```
int main(void)
{
    int studMark1, studMark2, studMark3,
        studMark4, ..., ..., studMark998, stuMark999,
        studMark1000;

    ...

    ...

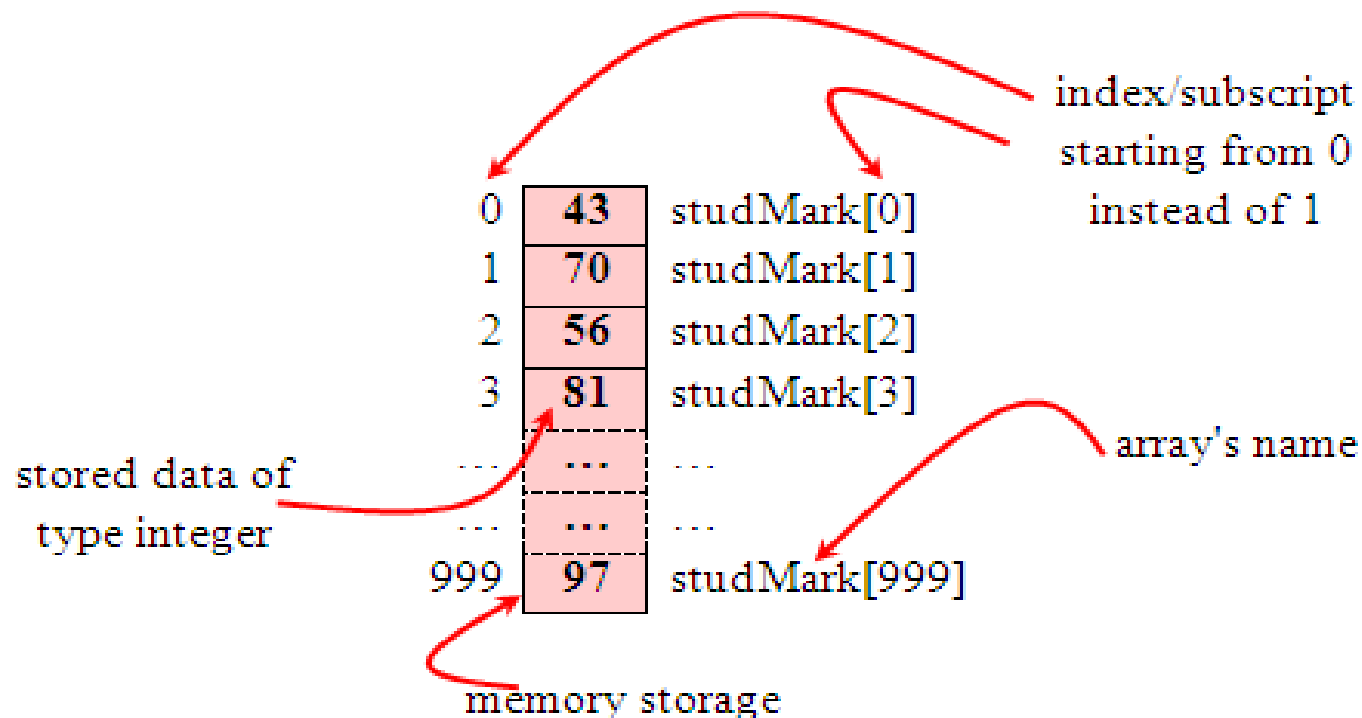
    return 0;
}
```



- By using an array, we just declare like this,

- `int studMark[1000];`

- This will reserve 1000 contiguous memory locations for storing the students' marks.
- Graphically, this can be depicted as in the following figure.



- This absolutely has simplified our declaration of the variables.
- We can use index or subscript to identify each element or location in the memory.
- Hence, if we have an index of `jIndex`, `studMark[jIndex]` would refer to the *jIndexth* element in the array of `studMark`.
- For example, `studMark[0]` will refer to the first element of the array.
- Thus by changing the value of `jIndex`, we could refer to any element in the array.
- So, array has simplified our declaration and of course, manipulation of the data.



- A single or one dimensional array declaration has the following form,
 - `array_element_data_type array_name[array_size];`
- Here, *array_element_data_type* define the base type of the array, which is the type of each element in the array.
- *array_name* is any valid C identifier name that obeys the same rule for the identifier naming.
- *array_size* defines how many elements the array will hold.



- For example, to declare an array of 30 characters, that construct a people name, we could declare,
 - `char cName[30];`
- Which can be depicted as follows,
- In this statement, the array character can store up to 30 characters with the first character occupying location `cName[0]` and the last character occupying `cName[29]`.
- Note that the index runs from 0 to 29. In C, an index always starts from 0 and ends with array's (size-1).
- So, take note the difference between the array size and subscript/index terms.

J	cName[0]
o	cName[1]
d	cName[2]
i	cName[3]
e	cName[4]
...	cName[5]
...	...
...	...
r	cName[29]



- Examples of the one-dimensional array declarations,
 - `int xNum[20], yNum[50];`
 - `float fPrice[10], fYield;`
 - `char chLetter[70];`
- The first example declares two arrays named `xNum` and `yNum` of type `int`. Array `xNum` can store up to 20 integer numbers while `yNum` can store up to 50 numbers.
- The second line declares the array `fPrice` of type `float`. It can store up to 10 floating-point values.
- `fYield` is basic variable which shows array type can be declared together with basic type provided the type is similar.
- The third line declares the array `chLetter` of type `char`. It can store a string up to 69 characters.
- Why 69 instead of 70? Remember, a string has a null terminating character (`\0`) at the end, so we must reserve for it.



ARRAY INITIALIZATION

- An array may be initialized at the time of declaration.
- Initialization of an array may take the following form,
 - `type array_name[size] = {a_list_of_value};`
- For example:
 - `int idNum[7] = {1, 2, 3, 4, 5, 6, 7};`
 - `float fFloatNum[5] = {5.6, 5.7, 5.8, 5.9, 6.1};`
 - `char chVowel[6] = {'a', 'e', 'i', 'o', 'u', '\0'};`
- The first line declares an integer array `idNum` and it immediately assigns the values 1, 2, 3, ..., 7 to `idNum[0]`, `idNum[1]`, `idNum[2]`, ..., `idNum[6]` respectively.
- The second line assigns the values 5.6 to `fFloatNum[0]`, 5.7 to `fFloatNum[1]`, and so on.
- Similarly the third line assigns the characters 'a' to `chVowel[0]`, 'e' to `chVowel[1]`, and so on. Note again, for characters we must use the single apostrophe/quote (') to enclose them.
- Also, the last character in `chVowel` is NULL character ('\0').



- Initialization of an array of type char for holding strings may take the following form,
 - ```
char array_name[size] =
 "string_lateral_constant";
```
- For example, the array chVowel in the previous example could have been written more compactly as follows,
  - ```
char    chVowel[6] = "aeiou";
```
- When the value assigned to a character array is a string (which must be enclosed in double quotes), the compiler automatically supplies the NULL character but we still have to reserve one extra place for the NULL.
- For unsized array (variable sized), we can declare as follow,
 - ```
char chName[] = "Mr. Dracula";
```
- C compiler automatically creates an array which is big enough to hold all the initializer.



# RETRIEVING ARRAY ELEMENTS

- If you want to retrieve specific element then then you have to specify not only the array or variable name but also the index number of interest.
- For example:
  - `int Arr[]={1,3,5,6,8};`
  - `printf(“%d\t%d\n”,Arr[1],Arr[2]);`
  - Output: 3 5



## ARRAY EXAMPLE

- Take 10 integer input from user and store then in an array and find the sum of all numbers stored in array.

```
#include<stdio.h>
int main(){
int i,sum=0,arr[10];
for(i=0;i<10;i++)
 scanf("%d",&arr[i]);
for(i=0;i<10;i++)
 sum+=arr[i];
printf("Sum of input integers is %d\n",sum);
return 0;
}
```



Summarize the response of a survey.

**Input:** Response of the survey, can be in the range between 0 and 10. Assume the population size to be 40.

**Output:** Frequency of each response.



- #include<stdio.h>
- #define SIZE 40
- #define ANS 11
  
- int main(void) {
- int response[SIZE];
- int freq[ANS] = {0};
- int i;
- for(i=0; i< SIZE; i++){
- scanf("%d",&response[i]);
- ++freq[response[i]];
- }
  
- for(i=0;i<ANS;i++)
- printf("Frequency of %d is %d\n",i,freq[i]);
- }



# ASSIGNMENT

- Read from user ages of all students in class and save them in an array which can store floating point and find average, minimum and maximum age.
- A six faced die is rolled 600 times. Find the frequency of the occurrence of each face?
- Store marks obtained by students in an array. Find if there is more than one student who scored same marks. Assume minimum marks obtained is 30 and maximum marks obtained is 85.



```
#include<stdio.h>
```

```
int main(){
```

```
int i,j,arr[7]={0};
```

```
srand(time(0));
```

```
for (i=0;i<600;i++){
```

```
 j=rand()%6;
```

```
 j=j+1;
```

```
 arr[j]++;
```

```
}
```

```
for(i=1;i<=6;i++)
```

```
 printf("%d came out for %d times\n",i,arr[i]);
```

```
return 0;
```

```
}
```





○ [sourav@gaya]\$ ./a.out

1 came out for 113 times

2 came out for 114 times

3 came out for 102 times

4 came out for 86 times

5 came out for 99 times

6 came out for 86 times



- **int rand(void):** returns a pseudo-random number in the range of 0 to RAND\_MAX.
- **RAND\_MAX:** is a constant whose default value may vary between implementations but it is granted to be at least 32767.
- **Issue:** If we generate a sequence of random number with rand() function, it will create the same sequence again and again every time program runs.



- The `srand()` function sets the starting point for producing a series of pseudo-random integers. If `srand()` is not called, the `rand()` seed is set as if `srand(1)` were called at program start.
- The pseudo-random number generator should only be seeded once, before any calls to `rand()`, and the start of the program.
- Standard practice is to use the result of a call to **`srand(time(0))`** as the seed. However, `time()` returns a `time_t` value which vary everytime and hence the pseudo-random number vary for every program call.



```
#include<stdio.h>
```

```
int main(){
int i,j,x,arr[10];
```

```
printf("Enter 10 integer number\n");
```

```
for(i=0;i<10;i++){
 scanf("%d",&arr[i]);
}
```

```
for(i=0;i<9;i++){
 x=arr[i];
 for(j=i+1;j<10;j++){
 if(x==arr[j])
 printf("%d number appeared more than once\n",x);
 }
}
return 0;
}
```



○ [sourav@gaya]\$ ./a.out

Enter 10 integer number

1

2

3

4

5

6

2

4

8

9

2 number appeared more than once

4 number appeared more than once



# *TWO DIMENSIONAL / 2D ARRAYS*

- A two dimensional array has two subscripts/indexes.
- The first subscript refers to the row, and the second, to the column.
- Its declaration has the following form,
  - `data_type        array_name[1st dimension size][2nd dimension size];`
- For examples,
  - `int            xInteger[3][4];`
  - `float        matrixNum[20][25];`
- The first line declares `xInteger` as an integer array with 3 rows and 4 columns.
- Second line declares a `matrixNum` as a floating-point array with 20 rows and 25 columns.



# DOUBLE SCRIPTED ARRAY WITH 3 ROWS AND 4 COLUMNS

|       | Column 0    | Column 1    | Column 2    | Column 3    |
|-------|-------------|-------------|-------------|-------------|
| Row 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Row 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Row 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

Diagram illustrating the double scripted array structure with 3 rows and 4 columns. The array is represented as a grid of cells, each containing a double-indexed expression (e.g., a[ row ][ column ]).

Arrows indicate the components of the indexing:

- Column index: Points to the second index (column) in the expression.
- Row index: Points to the first index (row) in the expression.
- Array name: Points to the variable 'a' in the expression.

## 2D array conceptual memory representation

Second subscript →

first subscript ↓

|           |           |           |           |
|-----------|-----------|-----------|-----------|
| abc[0][0] | abc[0][1] | abc[0][2] | abc[0][3] |
| abc[1][0] | abc[1][1] | abc[1][2] | abc[1][3] |
| abc[2][0] | abc[2][1] | abc[2][2] | abc[2][3] |
| abc[3][0] | abc[3][1] | abc[3][2] | abc[3][3] |
| abc[4][0] | abc[4][1] | abc[4][2] | abc[4][3] |

Here my array is `abc[5][4]`, which can be conceptually viewed as a matrix of 5 rows and 4 columns. Point to note here is that subscript starts with zero, which means `abc[0][0]` would be the first element of the array.





|           |           |           |           |           |         |           |           |
|-----------|-----------|-----------|-----------|-----------|---------|-----------|-----------|
| abc[0][1] | abc[0][2] | abc[0][3] | abc[1][0] | abc[1][1] | .... .. | abc[4][2] | abc[4][3] |
| 82206     | 82210     | 82214     | 82218     | 82222     |         | 82274     | 82278     |

memory locations for the array elements

Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

The addresses are generally represented in hex. This diagram shows them in integer just to show you that the elements are stored in contiguous locations, so that you can understand that the address difference between each element is equal to the size of one element(int size 4). For better understanding see the program below.

### Actual memory representation of a 2D array



- #include <stdio.h>
- int main() {
  - int abc[5][4] = { {0,1,2,3}, {4,5,6,7}, {8,9,10,11}, {12,13,14,15}, {16,17,18,19} };
  - for (int i=0; i<=4; i++) {
    - printf("%d ",abc[i]);
  - }
  - return 0;
- }
- Output: 1600101376 1600101392 1600101408  
1600101424 1600101440



# LIST OF STUDENTS AND THEIR SUBJECT MARKS

Marks  $\longrightarrow$

Students  $\downarrow$

|    |    |    |    |
|----|----|----|----|
| 10 | 23 | 31 | 11 |
| 20 | 43 | 21 | 21 |
| 12 | 22 | 30 | 13 |
| 30 | 31 | 26 | 41 |
| 13 | 03 | 41 | 15 |

Find the average mark scored by each student?



```
#include<stdio.h>
#define ROW 5
#define COL 4

int main(void)
{
 int i, j;
 double total;
 int marks[ROW][COL]= { 10, 23, 31, 11, 20, 43, 21, 21,12,
22, 30, 13, 30, 31, 26, 41,13, 03, 41, 15 };
 for(i = 0; i < ROW ; i++)
 {
 total = 0.0;
 for (j=0; j<COL; j++)
 total+=marks[i][j];
 printf("Average of student %d is %f\n", i, total/4.0);
 }
}
```



# INITIALIZATION OF 2D ARRAY

- `int disp[2][4] = { {10, 11, 12, 13}, {14, 15, 16, 17} };`
- OR
- `int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};`
- 1<sup>st</sup> one is recommended.



# THINGS THAT YOU MUST CONSIDER WHILE INITIALIZING A 2D ARRAY

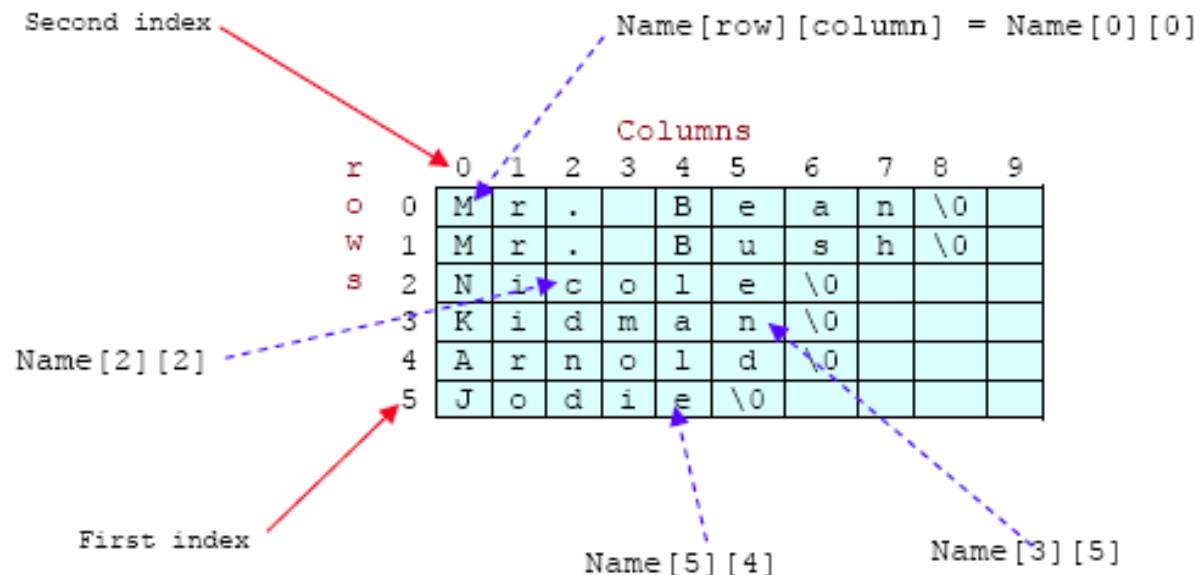
- We already know, when we initialize a normal **array** (or you can say one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if you are specifying elements during the declaration.
- `/* Valid declaration*/`
- `int abc[2][2] = {1, 2, 3 ,4 }`
- `/* Valid declaration*/`
- `int abc[][2] = {1, 2, 3 ,4 }`
- `/* Invalid declaration – you must specify second dimension*/`
- `int abc[][] = {1, 2, 3 ,4 }`
- `/* Invalid because of the same reason mentioned above*/`
- `int abc[2][] = {1, 2, 3 ,4 }`



- For array storing string

- ```
char Name[6][10] = {"Mr. Bean", "Mr. Bush",  
"Nicole", "Kidman", "Arnold", "Jodie"};
```

- Here, we can initialize the array with 6 strings, each with maximum 9 characters long.
- If depicted in rows and columns it will look something like the following and can be considered as contiguous arrangement in the memory.



ASSIGNMENTS

- Print Transpose of a Matrix
- Add Two Matrix Using Multi-dimensional Arrays
- Multiply to Matrix Using Multi-dimensional Arrays




```
#include <stdio.h>
```

```
void main()
```

```
{  
    static int array[10][10];  
    int i, j, m, n;  
    printf("Enter the order of the matrix \n");  
    scanf("%d %d", &m, &n);  
    printf("Enter the coefficients of the matrix\n");  
    for (i = 0; i < m; ++i){  
        for (j = 0; j < n; ++j){  
            scanf("%d", &array[i][j]);  
        }  
    }  
}
```



```
printf("The given matrix is \n");
for (i = 0; i < m; ++i){
    for (j = 0; j < n; ++j){
        printf(" %d", array[i][j]);
    }
    printf("\n");
}

printf("Transpose of matrix is \n");
for (j = 0; j < n; ++j){
    for (i = 0; i < m; ++i){
        printf(" %d", array[i][j]);
    }
    printf("\n");
}
}
```



```
#include <stdio.h>
int main()
{
    int r, c, a[100][100], b[100][100], sum[100][100], i, j;
    printf("Enter number of rows (between 1 and 100): ");
    scanf("%d", &r);
    printf("Enter number of columns (between 1 and 100): ");
    scanf("%d", &c);
    printf("\nEnter elements of 1st matrix:\n");
    for(i=0; i<r; ++i) {
        for(j=0; j<c; ++j) {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &a[i][j]);
        }
    }
}
```



```
printf("Enter elements of 2nd matrix:\n");
for(i=0; i<r; ++i)
    for(j=0; j<c; ++j) {
        printf("Enter element a%d%d: ",i+1, j+1);
        scanf("%d", &b[i][j]);
    }
// Adding Two matrices
for(i=0;i<r;++i)
    for(j=0;j<c;++j) {
        sum[i][j]=a[i][j]+b[i][j];
    }
// Displaying the result
printf("\nSum of two matrix is: \n\n");
for(i=0;i<r;++i)
    for(j=0;j<c;++j) {
        printf("%d ",sum[i][j]);
        if(j==c-1) { printf("\n\n"); }
    }
return 0;
}
```



- #include <stdio.h>
- int main() {
 - int a[10][10], b[10][10], result[10][10], r1, c1, r2, c2, i, j, k;
 - printf("Enter rows and column for first matrix: ");
 - scanf("%d %d", &r1, &c1);
 - printf("Enter rows and column for second matrix: ");
 - scanf("%d %d",&r2, &c2);
 - while (c1 != r2) {
 - printf("Error! Not compatible for multiplication\n");
 - }



- `printf("\nEnter elements of matrix 1:\n");`
- `for(i=0; i<r1; ++i)`
 - `for(j=0; j<c1; ++j) {`
 - `printf("Enter elements a%d%d: ",i+1, j+1);`
 - `scanf("%d", &a[i][j]);`
 - `}`
- `printf("\nEnter elements of matrix 2:\n");`
- `for(i=0; i<r2; ++i)`
 - `for(j=0; j<c2; ++j) {`
 - `printf("Enter elements b%d%d: ",i+1, j+1);`
`scanf("%d",&b[i][j]);`
 - `}`



- // Initializing all elements of result matrix to 0
- for(i=0; i<r1; ++i)
 - for(j=0; j<c2; ++j) {
 - result[i][j] = 0;
 - }
- // Multiplying matrices a and b and // storing result in result matrix
for(i=0; i<r1; ++i)
 - for(j=0; j<c2; ++j)
 - for(k=0; k<c1; ++k) {
 - result[i][j]+=a[i][k]*b[k][j];
 - }



- // Displaying the result
- `printf("\nOutput Matrix:\n");`
- `for(i=0; i<r1; ++i)`
 - `for(j=0; j<c2; ++j) {`
 - `printf("%d ", result[i][j]);`
 - `if(j == c2-1) printf("\n\n");`
 - `}`
- `return 0;`
- `}`



3D: A 3D ARRAY IS AN ARRAY OF 2D ARRAYS.

- `#include<stdio.h>`
- `int main(){`
- `int`
`i,j,k,x[][2][3]={{{1,2,3},{4,5,6}},{{7,8,9},{10,11,12}}};`
- `for(i=0;i<2;i++)`
- `for(j=0;j<2;j++)`
- `for(k=0;k<3;k++){`
- `printf("x[%d,%d,%d]=%d\n",i,j,k,x[i][j][k]);`
- `}`
- `return 0;`
- `}`



FUNCTIONS

- Functions
 - Modularize a program
 - All variables defined inside functions are local variables
 - Parameters
 - Communicate information between functions
- Benefits of functions
 - Divide and conquer
 - Software reusability
 - Avoid code repetition



PROGRAM TO COMPUTE FACTORIAL N

```
#include<stdio.h>
int factorial(int n);
int main(void) {
    int input, output;
    scanf("%d", &input);
    output = factorial(input);
    printf("Factorial of %d is %d\n", input, output);
}

int factorial(int n) {
    int i, prod = 1;
    for (i = 2; i <= n; ++i)
        prod *=i;
    return prod;
}
```



PROGRAM TO COMPUTE FACTORIAL N

```
#include<stdio.h>
int factorial(int n);
int main(void) {
    int input, output;
    scanf("%d", &input);
    output = factorial(input);
    printf("Factorial of %d is %d\n", input, output);
}
int factorial(int n) {
    int i, prod = 1;
    for (i = 2; i <= n; ++i)
        prod *=i;
    return prod;
}
```



PROGRAM TO COMPUTE

FACTORIAL N – MAIN() FUNCTION

```
#include<stdio.h>
```

```
int factorial(int n);
```

```
int main(void) {
```

```
    int input, output;
```

```
    scanf("%d", &input);
```

```
        output = factorial(input);
```

```
    printf("Factorial of %d is %d\n", input, output);
```

```
}
```



FUNCTION DECLARATION

- Function declarations are generated in various ways:
 - function definition
 - provides both declaration and definition.
 - no assumption about parameter list
 - function declaration/prototype
- Functions should be declared before they can be used, called the function prototype.
- Tells the compiler the number and type of argument that will be passed to the function and the type of value that will be returned.
- Syntax: *type function_name(parameter type list)*



PROGRAM TO COMPUTE FACTORIAL N

```
#include<stdio.h>
int factorial(int n);
int main(void) {
    int input, output;
    scanf("%d", &input);
    output = factorial(input);
    printf("Factorial of %d is %d\n", input, output);
}

int factorial(int n) {
    int i, prod = 1;
    for (i = 2; i <= n; ++i)
        prod *=i;
    return prod;
}
```



PROGRAM TO COMPUTE FACTORIAL N – FACTORIAL() FUNCTION

```
int factorial(int n) {  
    int i, prod = 1;  
    for (i = 2; i <= n; ++i)  
        prod *=i;  
    return prod;  
}
```



FUNCTION DEFINITIONS

- Function definition format

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Function-name: any valid identifier
- Return-value-type: data type of the result (default `int`) will be converted if necessary
 - `void` – indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters



FUNCTION DEFINITIONS (2)

- Definitions and statements: function body (block)
 - Variables can be defined inside blocks (can be nested)
 - Functions can not be defined inside other functions
- Returning control
 - If nothing returned
 - `return;`
 - If something returned
 - `return expression;`



FINDING THE MAXIMUM OF THREE INTEGERS

```
o #include <stdio.h>
```

```
int maximum( int, int, int );
```

```
int main() {
```

```
    int a, b, c;
```

```
    printf( "Enter three integers: " );
```

```
    scanf( "%d%d%d", &a, &b, &c );
```

```
    printf( "Maximum is: %d\n", maximum( a, b,  
    c ) );
```

```
    return 0;
```

```
}
```



```
/* FUNCTION MAXIMUM DEFINITION */
```

```
int maximum( int x, int y, int z ){  
    int max = x;  
    if ( y > max )  
        max = y;  
    if ( z > max )  
        max = z;  
    return max;  
}
```



○ Function declaration / prototype

- Function name
- Parameters – what the function takes in
- Return type – data type function returns
- Used to validate functions
- Prototype only needed if function definition comes after use in program
- The function with the prototype
 - `int maximum(int, int, int);`
 - Takes in 3 `ints`
 - Returns an `int`



CALLING FUNCTIONS: CALL BY VALUE AND CALL BY REFERENCE

- Used when invoking functions
- Call by value
 - Copy of argument passed to function
 - Changes in function do not effect original
 - Use when function does not need to modify argument
 - Avoids accidental changes
- Call by reference
 - Passes original argument
 - Changes in function effect original
 - Only used with trusted functions
- For now, we focus on call by value



ASSIGNMENT: GAME OF CHANCE

- Roll two dice
 - 7 or 11 on first throw, player wins
 - 12 on first throw, player loses
 - 2, 3, 4, 5, 6, 8, 9, 10 - value becomes player's "point"
 - Player will continue to play until he loses or wins the game.
 - If cumulative points becomes divisible by 7 or 11 player wins
 - If cumulative point is not divisible by 7 or 11 but divisible by 12 player loses.
- Find out number of throws for coming to a decision



STORAGE CLASSES

○ Storage class specifiers

- Storage duration – how long an object exists in memory
- Scope – where object can be referenced in program

○ Automatic storage

- Object created and destroyed within its block
- **auto**: default for local variables
`auto double x, y;`
- **register**: tries to put variable into high-speed registers

- Can only be used for automatic variables

`register int counter = 1;`



- Static storage
 - Variables exist for entire program execution
 - Default value of zero
 - **static**: local variables defined in functions.
 - Keep value after function ends
 - Only known in their own function
- **extern**: default for global variables and functions
 - Known in any function
- `extern int var;` declare a variable not define
- `int var;` define along with declaration



Example 1:

```
int var;  
int main(void)  
{  
    var = 10;  
    return 0;  
}
```

- This program is compiled successfully. Here var is defined (and declared implicitly) globally.



- Example 2:

```
extern int var;  
int main(void)  
{  
    return 0;  
}
```

- This program throws error in compilation. Because var is declared but not defined anywhere. Essentially, the var isn't allocated any memory.



- `int x; // is the content of sample.h`

`#include "full path of sample.h"`

`extern int var;`

`int main(void)`

`{`

`var = 10;`

`return 0;`

`}`

- This program will be compiled successfully.



```
extern int var = 0;  
int main(void)  
{  
    var = 10;  
    return 0;  
}
```

If a variable is only declared and an initializer is also provided with that declaration, then the memory for that variable will be allocated i.e. that variable will be considered as defined.



IMPORTANT USE OF EXTERN VARIABLE

- File `stdio.h`:
- `int errno;`
- `/* other stuff...*/`

- `myCFile1.h`:
- `#include <stdio.h>`
- Code...

- `myCFile2.c`:
- `#include <stdio.h>`
- Code...



- File `stdio.c`
- `int errno;`
- File `stdio.h`:
- `extern int errno;`
- `/* other stuff...*/`
- `myCFile1.c`:
- `#include <stdio.h>`
- Code...
- `myCFile2.c`:
- `#include <stdio.h>`
- Code...



- Content of file test22.c
- `#include<stdio.h>`
- `#include"/home/fac/sourav/practice/sample.h"`
- `int abc(){`
- `extern int xx;`
- `printf("value of xx in abc=%d\n",xx);`
- `return 0;`
- `}`



- Content of file sample.h
- `int xx=5;`



- Content of test23.c
- `#include<stdio.h>`
- `int main(){`
- `abc();`
- `printf("Inside main\n");`
- `return 0;`
- `}`



COMPILATION AND RUN

- gcc test23.c test22.c
- ./a.out
- value of xx in abc=5
- Inside main



STATIC FUNCTION

- Unlike global functions in C, access to static functions is restricted to the file where they are declared.



EXAMPLE

- Content of test22.c
- `#include<stdio.h>`
- `#include"/home/fac/sourav/practice/sample.h"`
- `int abc(){`
- `extern int xx;`
- `printf("value of xx in abc=%d\n",xx);`
- `return 0;`
- `}`
- `static int xyz(){`
- `printf("Inside xyz\n");`
- `return 0;`
- `}`



- Content of test23.c
- #include<stdio.h>
- int main(){
- abc();
- xyz();
- printf("Inside main\n");
- return 0;
- }



- `cc test22.c test23.c`
- `/tmp/ccBEYzN4.o: In function `main':`
- `test23.c:(.text+0x14): undefined reference to `xyz'`
- `collect2: ld returned 1 exit status`



SCOPE RULES

- File scope
- Function scope
- Block scope



STORAGE CLASS AND SCOPE EXAMPLE

```
#include <stdio.h>

void a( void );/* function prototype */
void b( void );/* function prototype */
void c( void );/* function prototype */

int x=1;

int main(){
    int x=5; /* local variable to main */
    printf("local x in outer scope of main is %d\n", x );
    {/*start new scope*/
        int x=7;
        printf( "local x in inner scope of main is %d\n", x
        );
    } /*end of inner scope*/
    printf( "local x in outer scope of main is %d\n", x );
    a(); b(); c(); a(); b(); c();
    printf( "local x in main is %d\n", x );
    return 0;
}
```



```
void a(void){  
    int x = 25;  /* initialized each time a is called */  
    printf( "\nlocal x in a is %d after entering a\n", x );  
    ++x;  
    printf( "local x in a is %d before exiting a\n", x );  
}
```

```
void b( void ){  
    static int x = 50; /* static initialization only first  
time b is called */  
    printf( "\nlocal static x is %d on entering b\n", x );  
    ++x;  
    printf( "local static x is %d on exiting b\n", x );  
}
```



```
void c( void ){  
    printf("\nglobal x is %d on entering c\n", x );  
    x *= 10;  
    printf( "global x is %d on exiting c\n", x );  
}
```



OUTPUT

- local x in outer scope of main is 5
- local x in inner scope of main is 7
- local x in outer scope of main is 5
-
- local x in a is 25 after entering a
- local x in a is 26 before exiting a
-
- local static x is 50 on entering b
- local static x is 51 on exiting b
-
- global x is 1 on entering c
- global x is 10 on exiting c
-
- local x in a is 25 after entering a
- local x in a is 26 before exiting a
-
- local static x is 51 on entering b
- local static x is 52 on exiting b
-
- global x is 10 on entering c
- global x is 100 on exiting c
- local x in main is 5



PASSING ARRAYS TO FUNCTIONS

○ Passing arrays

- To pass an array argument to a function, specify the name of the array without any brackets

```
int myArray[ 24 ] ;
```

```
myFunction ( myArray , 24 ) ;
```

- Array size usually passed to function
- Arrays passed call-by-reference
- Name of array is address of first element
- Function knows where the array is stored
 - Modifies original memory locations

○ Passing array elements

- Passed by call-by-value
- Pass subscripted name (i.e., **myArray[3]**) to function



- Function prototype

```
void modifyArray( int b[], int arraySize );
```

- Parameter names optional in prototype
 - `int b[]` could be written `int []`
 - `int arraySize` could be simply `int`



PASSING ARRAYS AND INDIVIDUAL ARRAY ELEMENTS TO FUNCTIONS

```
○ #include <stdio.h>
○ #define SIZE 5
○ void modifyArray( int [], int ); /* appears strange */
○ void modifyElement( int );
○ int main(){
○ int a[ SIZE ] = { 0, 1, 2, 3, 4 }, i;
○ printf( "The values of the original array are:\n" );
○ for ( i = 0; i <= SIZE - 1; i++ )
    • printf( "%d", a[ i ] );
○ modifyArray( a, SIZE ); /* passed call by reference */
○ printf( "The values of the modified array are:\n" );
○ for ( i = 0; i <= SIZE - 1; i++ )
    ○ printf( "%d", a[ i ] );
    • printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
    • modifyElement( a[ 3 ] );
    • printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
    • return 0;
    • }
```



```
void modifyArray( int b[], int size ){  
    int j;  
    for ( j = 0; j <= size - 1; j++ )  
        b[ j ] *= 2;  
}
```

```
void modifyElement( int e ){  
    e=e*2;  
    printf( "Value in modifyElement is %d\n",  
e) ;  
}
```



The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

The value of a[3] is 6

Value in modifyElement is 12

The value of a[3] is 6



```
#include <stdio.h>

const int M = 3;
const int N = 3;

void print(int arr[M][N])
{
    int i, j;
    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            printf("%d ", arr[i][j]);
}

int main()
{
    int arr[][N] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
    print(arr);
    return 0;
}
```



```
#include <stdio.h>
```

```
const int N = 3;
```

```
void print(int arr[][N], int m)
```

```
{
```

```
    int i, j;
```

```
    for (i = 0; i < m; i++)
```

```
        for (j = 0; j < N; j++)
```

```
            printf("%d ", arr[i][j]);
```

```
}
```

```
int main()
```

```
{
```

```
    int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

```
    print(arr, 3);
```

```
    return 0;
```

```
}
```

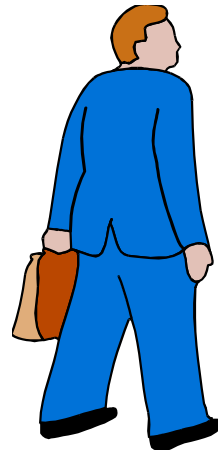


RECURSION

Handshake Problem: There are n people in a room. If each person shakes hands once with every other person. What is the total number of handshakes ($h(n)$)?

$$h(n) = h(n-1) + n-1$$

$$h(4) = h(3) + 3 \quad h(3) = h(2) + 2 \quad h(2) = 1$$



$$h(n): \text{Sum of integer from 1 to } n-1 = n(n-1) / 2$$

RECURSION

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a **smaller version** of the same problem.
- Example: the factorial function

$$6! = 6 * 5 * 4 * 3 * 2 * 1$$

We could write:

$$6! = 6 * 5!$$



EXAMPLE 2: FACTORIAL FUNCTION

In general, we can express the factorial function as follows:

$$n! = n * (n-1)!$$

Is this correct? Well... almost.

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

$$n! = 1 \quad (\text{if } n \text{ is equal to } 1)$$

$$n! = n * (n-1)! \quad (\text{if } n \text{ is larger than } 1)$$



FACTORIAL FUNCTION

The C equivalent of this definition:

```
int fac(int numb) {  
    if (numb<=1)  
        return 1;  
    else  
        return numb * fac(numb-1);  
}
```

recursion means that a function calls itself



FACTORIAL FUNCTION

- Assume the number typed is 3, that is, numb=3.

fac(3) :

3 <= 1 ? No.

fac(3) = 3 * fac(2)

fac(2) :

2 <= 1 ? No.

fac(2) = 2 * fac(1)

fac(1) :

1 <= 1 ? Yes.

return 1

fac(2) = 2 * 1 = 2

return fac(2)

fac(3) = 3 * 2 = 6

return fac(3)

fac(3) has the value 6

```
int fac(int numb) {  
    if(numb<=1)  
        return 1;  
    else  
        return numb * fac(numb-1);  
}
```


FACTORIAL FUNCTION

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive solution

```
int fac(int numb) {  
    if (numb<=1)  
        return 1;  
    else  
        return numb*fac (numb-1) ;  
}
```

Iterative solution

```
int fac(int numb) {  
    int product=1;  
    while (numb>1) {  
        product *= numb;  
        numb--;  
    }  
    return product;  
}
```



RECURSION

We have to pay a price for recursion:

- calling a function **consumes more time and memory** than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)



RECURSION

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
```

```
...
```



Oops!

```
int result = 1;  
while(result >0) {  
    ...  
    result++;  
}
```



Oops!

RECURSION

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb) {  
    return numb * fac(numb-1);  
}
```

Or:

```
int fac(int numb) {  
    if (numb<=1)  
        return 1;  
    else  
        return numb * fac(numb+1);  
}
```

Oops!
No termination
condition

Oops!

RECURSION

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain **at least one non-recursive branch**.
- The recursive calls must eventually lead to a non-recursive branch.



RECURSION

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- The smallest example of the same task has a non-recursive solution.

Example: The factorial function

$$n! = n * (n-1)! \text{ and } 1! = 1$$



HOW MANY PAIRS OF RABBITS CAN BE PRODUCED FROM A SINGLE PAIR IN A YEAR'S TIME?

○ Assumptions:

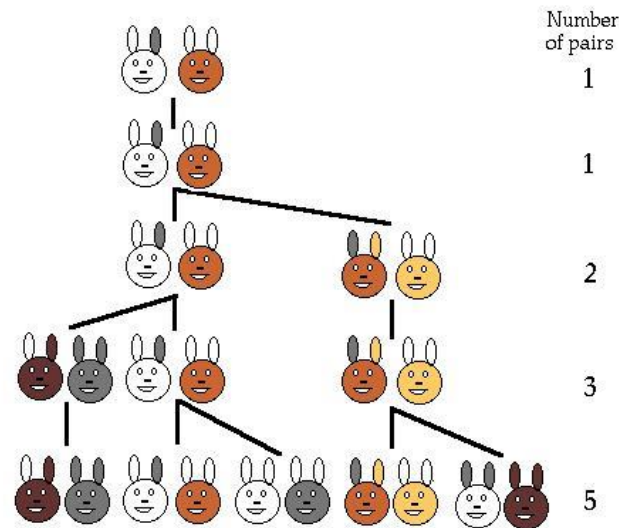
- Each pair of rabbits produces a new pair of offspring every month;
- each new pair becomes fertile at the age of one month;
- none of the rabbits dies in that year.

○ Example:

- After 1 month there will be 2 pairs of rabbits;
- after 2 months, there will be 3 pairs;
- after 3 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all).



POPULATION GROWTH IN NATURE



- Leonardo Pisano (Leonardo Fibonacci = Leonardo, son of Bonaccio) proposed the sequence in 1202 in *The Book of the Abacus*.
- Fibonacci numbers are believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.

DIRECT COMPUTATION METHOD

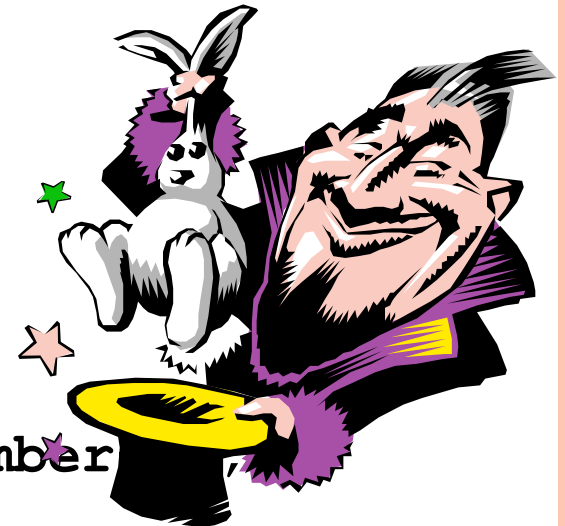
○ Fibonacci numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

where each number is the sum of the preceding two.

○ Recursive definition:

- $F(0) = 0;$
- $F(1) = 1;$
- $F(\text{number}) = F(\text{number}-1) + F(\text{number}-2)$



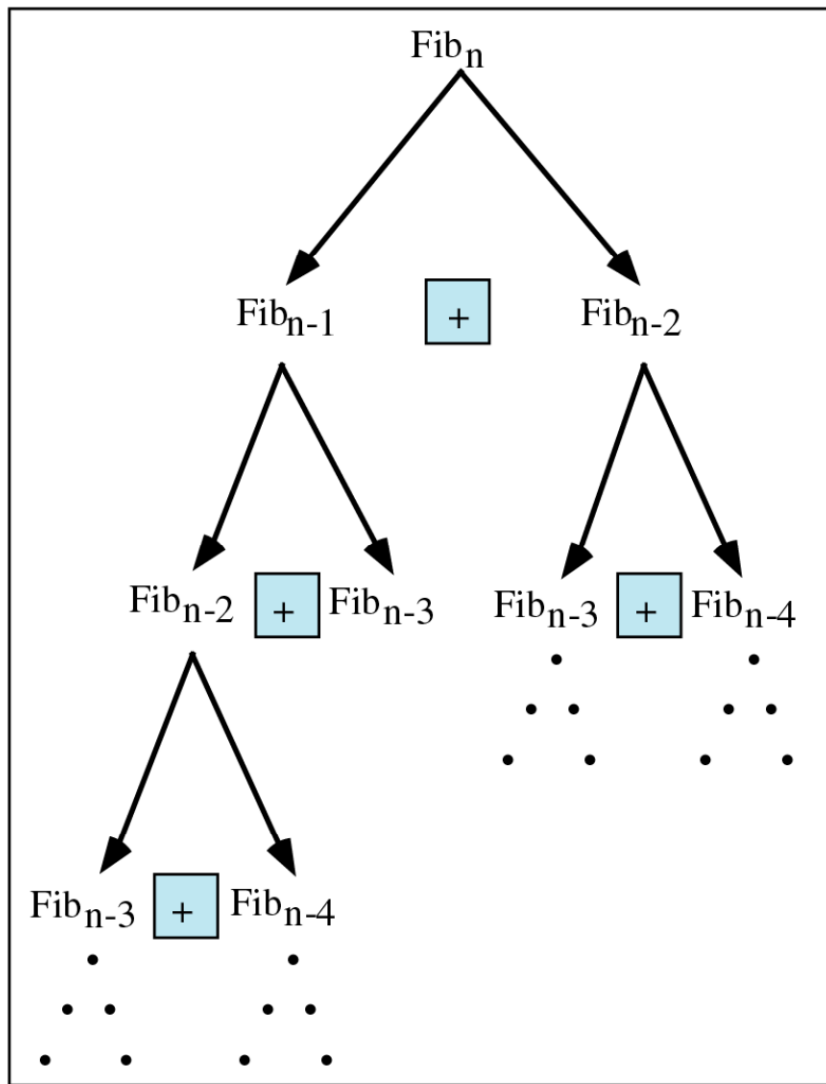
EXAMPLE 3: FIBONACCI NUMBERS

//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well

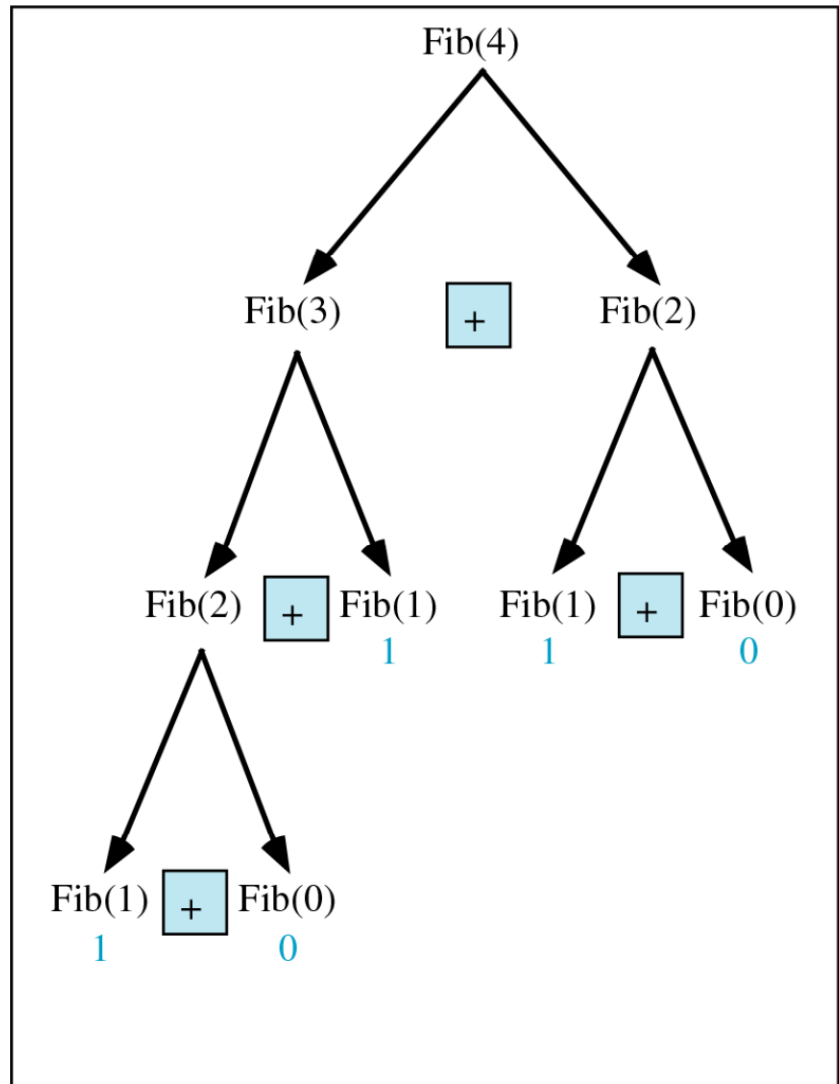
```
int fib(int number)
{
    if (number == 0) return 0;
    if (number == 1) return 1;
    return (fib(number-1) + fib(number-2));
}

int main(){    // driver function
    int inp_number;
    cout << "Please enter an integer: ";
    cin >> inp_number;
    cout << "The Fibonacci number for "<< inp_number
         << " is "<< fib(inp_number)<<endl;
    return 0;
}
```





(a) $\text{Fib}(n)$



(b) $\text{Fib}(4)$

TRACE A FIBONACCI NUMBER

- Assume the input number is 4, that is, num=4:

fib(4) :

4 == 0 ? No; 4 == 1? No.

fib(4) = fib(3) + fib(2)

fib(3) :

3 == 0 ? No; 3 == 1? No.

fib(3) = fib(2) + fib(1)

fib(2) :

2 == 0? No; 2==1? No.

fib(2) = fib(1) + fib(0)

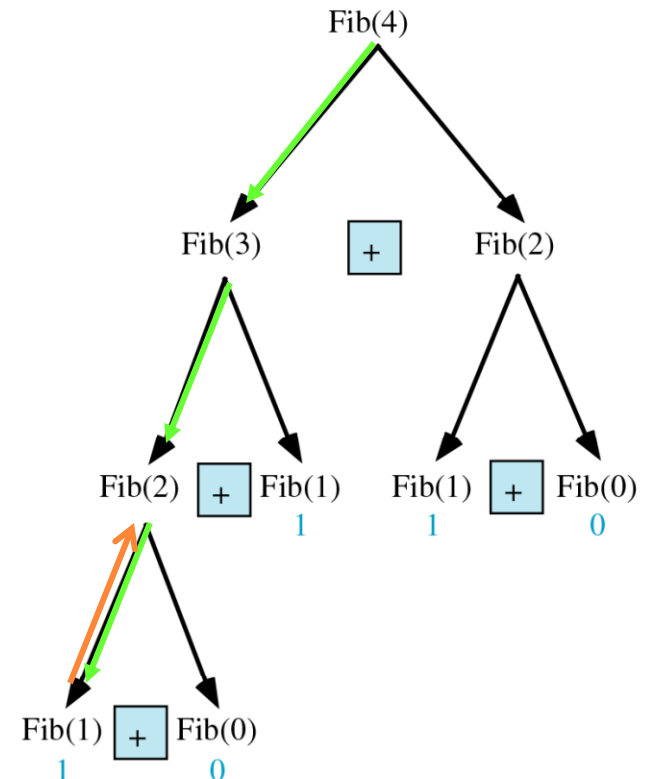
fib(1) :

1 == 0 ? No; 1 == 1? Yes.

fib(1) = 1;

return fib(1);

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



TRACE A FIBONACCI NUMBER

```
fib(0) :
```

```
    0 == 0 ? Yes.
```

```
    fib(0) = 0;
```

```
    return fib(0);
```

```
fib(2) = 1 + 0 = 1;
```

```
return fib(2);
```

```
fib(3) = 1 + fib(1)
```

```
fib(1) :
```

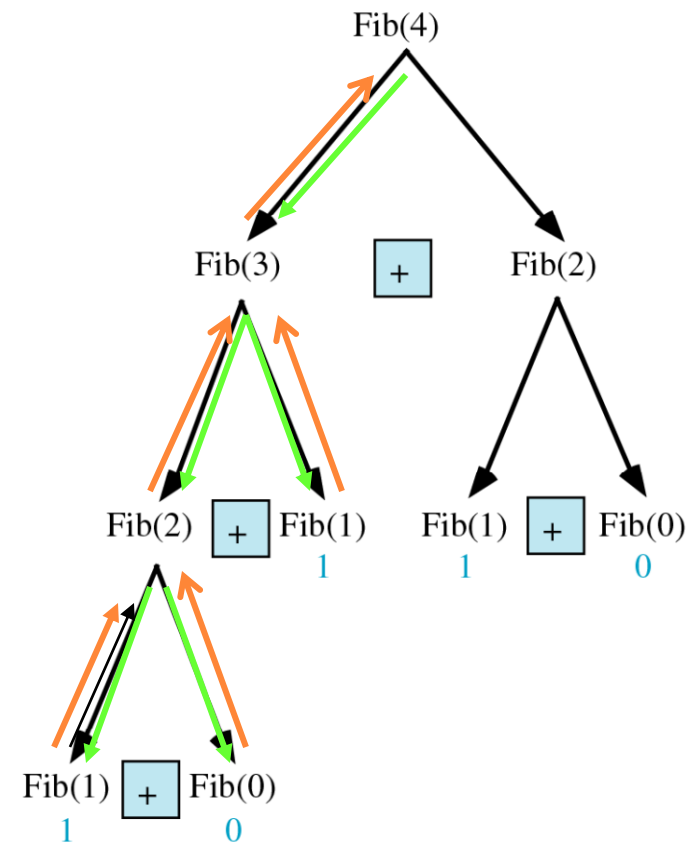
```
1 == 0 ? No; 1 == 1? Yes
```

```
fib(1) = 1;
```

```
return fib(1);
```

```
fib(3) = 1 + 1 = 2;
```

```
return fib(3)
```



TRACE A FIBONACCI NUMBER

```
fib(4)=fib(3)+fib(2)=2+fib(2)
```

```
fib(2):
```

```
2 == 0 ? No; 2 == 1?      No.
```

```
fib(2) = fib(1) + fib(0)
```

```
fib(1):
```

```
1 == 0 ? No; 1 == 1?  Yes.
```

```
fib(1) = 1;
```

```
return fib(1);
```

```
fib(0):
```

```
0 == 0 ?    Yes.
```

```
fib(0) = 0;
```

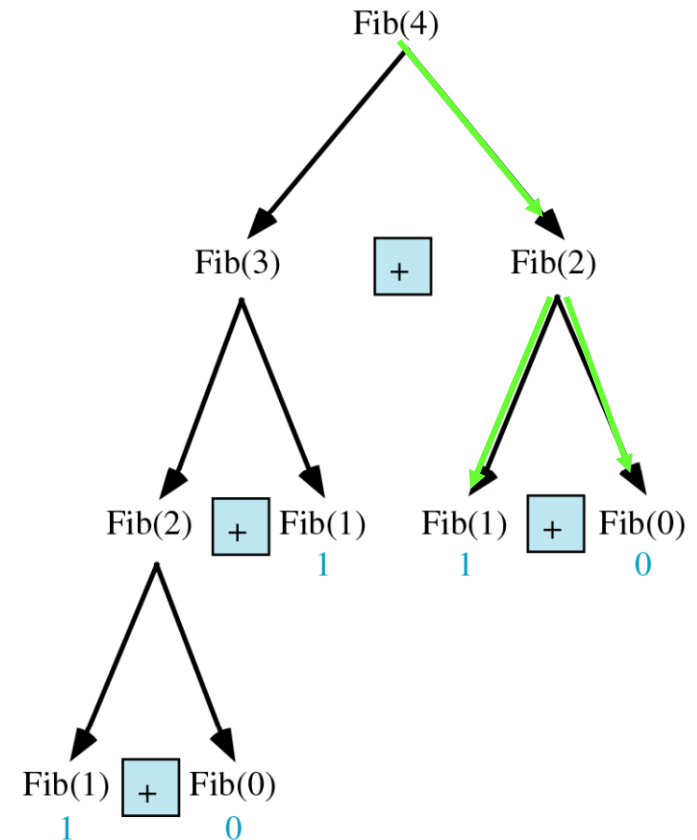
```
return fib(0);
```

```
fib(2) = 1 + 0 = 1;
```

```
return fib(2);
```

```
fib(4) = 2 + 1 = 3;
```

```
return fib(4);
```



EXAMPLE 4: FIBONACCI NUMBER W/O RECURSION

//Calculate Fibonacci numbers iteratively
//much more efficient than recursive solution

```
int fib(int n)
{
    int f[100];
    f[0] = 0; f[1] = 1;
    for (int i=2; i<= n; i++)
        f[i] = f[i-1] + f[i-2];
    return f[n];
}
```



EXAMPLE 5: BINARY SEARCH


- Search for an element in an array
 - Sequential search
 - Binary search
- Binary search
 - Compare the search element with the middle element of the array
 - If not equal, then apply binary search to half of the array (if not empty) where the search element would be.



BINARY SEARCH WITH RECURSION

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
            int first,        // input: lower bound
            int last,         // input: upper bound
            int value         // input: value to find
            )// output: index if found, otherwise return -1

{ //cout << "bsearch(data, "<<first<< ", last "<< ", "<<value << "); "<<endl;
  int middle = (first + last) / 2;
  if (data[middle] == value)
    return middle;
  else if (first >= last)
    return -1;
  else if (value < data[middle])
    return bsearchr(data, first, middle-1, value);
  else
    return bsearchr(data, middle+1, last, value);
}
```



BINARY SEARCH

```
int main() {  
    const int array_size = 8;  
    int list[array_size]={1, 2, 3, 5, 7, 10, 14, 17};  
    int search_value;  
  
    cout << "Enter search value: ";  
    cin >> search_value;  
    cout << bsearchr(list,0,array_size-1,search_value)  
        << endl;  
  
    return 0;  
}
```



BINARY SEARCH W/O RECURSION

```
// Searches an ordered array of integers
int bsearch(const int data[], // input: array
            int size,        // input: array size
            int value        // input: value to find
            ){               // output: if found, return
                             // index; otherwise, return -1

    int first, last, upper;
    first = 0;
    last = size - 1;
    while (true) {
        middle = (first + last) / 2;
        if (data[middle] == value)
            return middle;
        else if (first >= last)
            return -1;
        else if (value < data[middle])
            last = middle - 1;
        else
            first = middle + 1;
    }
}
```



RECURSION GENERAL FORM

- How to write recursively?

```
int recur_fn(parameters) {  
    if(stopping condition)  
        return stopping value;  
    // other stopping conditions if needed  
    return function of recur_fn(revised parameters)  
}
```



EXAMPLE 6: EXPONENTIAL FUNC

- How to write `exp(int numb, int power)` recursively?

```
int exp(int numb, int power) {  
    if(power ==0)  
        return 1;  
    return numb * exp(numb, power -1) ;  
}
```



EXAMPLE 6: NUMBER OF ZERO

- Write a recursive function that counts the number of zero digits in an integer
- **zeros(10200)** returns 3.

```
int zeros(int numb) {  
    if (numb==0)                // 1 digit (zero/non-zero):  
        return 1;              // bottom out.  
    else if (numb < 10 && numb > -10)  
        return 0;  
    else                        // > 1 digits: recursion  
        return zeros(numb/10) + zeros(numb%10);  
}
```

```
zeros(10200)  
zeros(1020)          + zeros(0)  
zeros(102)           + zeros(0) + zeros(0)  
zeros(10)             + zeros(2) + zeros(0) + zeros(0)  
zeros(1) + zeros(0) + zeros(2) + zeros(0) + zeros(0)
```

PROBLEM SOLVING USING RECURSION

Let us consider a simple problem of printing a message for n times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for $n-1$ times. The second problem is the same as the original problem with a smaller size. The base case for the problem is $n==0$. You can solve this problem using recursion as follows:

```
void nPrintln(char * message, int times)
{
    if (times >= 1) {
        printf("%s",message);
        nPrintln(message, times - 1);
    } // The base case is n == 0
}
```



RECURSIVE PALINDROME

Palindrome is a string that reads the same backwards as forwards.

```
bool isPalindrome(const char s[], int low, int high)
{
    if (high <= low) // Base case
        return true;
    else if (s[low] != s[high]) // Base case
        return false;
    else
        return isPalindrome(s, low + 1, high - 1);
}
bool checkPalindrome(const char s[], int size)
{
    return isPalindrome(s, 0, size - 1);
}
```


TOWERS OF HANOI



- Only one disc could be moved at a time
- A larger disc must never be stacked above a smaller one
- One and only one extra needle could be used for intermediate storage of discs



TOWERS OF HANOI


```
void hanoi(int from, int to, int num)
{
    int temp = 6 - from - to; //find the temporary
                               //storage column

    if (num == 1) {
        printf("move disc 1 from %d to %d\n", from, to);
    }
    else {
        hanoi(from, temp, num - 1);
        printf("move disc %d from %d to %d\n",
            num, from, to);
        hanoi(temp, to, num - 1);
    }
}
```



TOWERS OF HANOI

```
int main() {  
    int num_disc;    //number of discs  
  
    printf("Please enter a positive number (0 to quit)");  
    scanf("%d",&num_disc);  
  
    while (num_disc > 0){  
        hanoi(1, 3, num_disc);  
        printf("Please enter a positive number");  
        scanf("%d",&num_disc);  
    }  
    return 0;  
}
```



ASSIGNMENTS

- 1) Take two integer input x and n and find out x^n using recursion.
- 2) As input take an integer array A and an integer n , such that A has at least n elements. You have to find the sum of the first n integers in A *using recursion*.
- 3) As input take an array A and nonnegative integer indices i and j and find the reversal of the elements in A starting at index i and ending at j *using recursion*.
- 4) *Take two integer inputs m and n find multiplication of m and n (mn) using addition in recursive manner.*
- 5) *Write a recursive function which takes a character array and a character and find the number of occurrence of that character in the array*



RECURSIVE SQUARING

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x, n) = \begin{cases} 1 & \text{if } x = 0 \\ x \cdot p(x, (n-1)/2)^2 & \text{if } x > 0 \text{ is odd} \\ p(x, n/2)^2 & \text{if } x > 0 \text{ is even} \end{cases}$$

- For example,

$$2^4 = 2^{(4/2)^2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$$

$$2^5 = 2^{1+(4/2)^2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$$

$$2^6 = 2^{(6/2)^2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$$

$$2^7 = 2^{1+(6/2)^2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$$



POWER(X,N) USING RECURSION

```
#include<stdio.h>

int power(int x, int n){
    if(n==1)
        return x;
    else
        return (x * power(x,n-1));
}

int main(){
    int x,n;
    printf("Enter value of x and n\n");
    scanf("%d%d",&x,&n);
    printf("pow(%d,%d)=%d\n",x,n,power(x,n));
    return 0;
}
```



```
#include<stdio.h>
```

```
int sum(int A[],int n){  
if(n==1)
```

```
    return(A[0]);
```

```
else
```

```
    return (A[n-  
1]+sum(A,n-1));  
}
```

```
int main(){
```

```
int B[]={1,2,3,4,5,6,7,8,9,10},n;
```

```
printf("Enter value of n  
(<10)\n");
```

```
scanf("%d",&n);
```

```
printf("Sum(B,%d)=%d\n",n,s  
um(B,n));
```

```
return 0;
```

```
}
```



REVERSE A STRING

```
#include<stdio.h>
#include<string.h>
void reverse(char A[],int i, int j){
char temp;
if(i>j || i==j)
    return ;
else
{
temp=A[i];
A[i]=A[j];
A[j]=temp;
reverse(A,i+1,j-1);
}
}
```


```
int main(){
char x[50];
printf("Enter string x\n");
scanf("%s",x);
int m=strlen(x);
reverse(x,0,m-1);
printf("Reversed string
is\n%s\n",x);
return 0;
}
```




```
#include <stdio.h>
```

```
int multiply(int m, int n){  
    if(n==1)  
        return m;  
    else {  
        return  
        (m+multiply(m,n-1));  
    }  
}
```

```
int main(){  
    int m,n;  
    printf("Enter value of m  
    and n\n");  
    scanf("%d%d",&m,&n);  
    printf("Mul(%d,%d)=%d\n",  
    m,n,multiply(m,n));  
    return 0;  
}
```



FREQUENCY COUNT

```
#include<stdio.h>
#include<string.h>
int countfreq(char A[],int size, char c){
    if(size==1){
        if(A[0]==c)
            return 1;
        else
            return 0;
    }
    else{
        if(A[size-1]==c)
            return (1+countfreq(A,size-
1,c));
        else
            return (countfreq(A,size-1,c));
    }
}
```

```
int main(){
    char x[50],c;
    printf("Enter character
and input string\n");
    scanf("%c",&c);
    scanf("%s",x);
    int n=strlen(x);
    printf("Frequenct of %c in
%s is
%d\n",c,x,countfreq(x,n,c));
    return 0;
}
```

