

Applied Programming

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

INTRODUCTION

- ***What is Data Structures?***
 - ***A data structure is defined by***
 - ***(1) the logical arrangement of data elements, combined with***
 - ***(2) the set of operations we need to access the elements.***

Types of Data

- Elementary Data:
 - The data that is not further subdivided.
 - Contains an Atomic Data.
 - Atomic variables can only store one value at a time.
- Group Data:
 - The data that is further subdivided.
 - May contain multiple information in one unit.

What is Data Structures?

- Example:library
 - is composed of elements (books)
 - Accessing a particular book requires knowledge of the arrangement of the books
 - Users access books only through the librarian



*the logical arrangement of data elements,
combined with
the set of operations we need to access the
elements.*

Basic Operation

- Traversing
- Insertion
- Deletion
- Updating
- Searching

Secondary Operations

- Sorting
- Merging

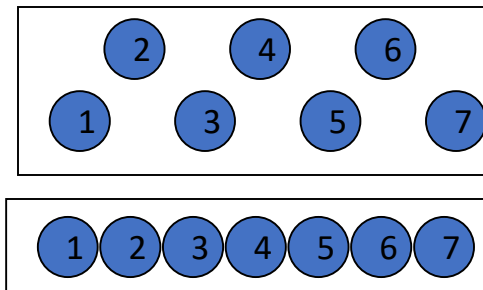
Basic Data Structures

- Structures include
 - Structure
 - Arrays
 - linked lists
 - Stack, Queue
 - binary trees
 - ...and others

What is Algorithm?

- **Algorithm:**

- A computable set of steps to achieve a desired result
- A step by step set of instructions to achieve a task
- Relationship to Data Structure
 - Example: Find an element



Summary

Algorithms + Data Structures = Programs

Algorithms \leftrightarrow Data Structures

Revision of some basic concepts

- 1. ADDRESS**
- 2. POINTERS**
- 3. ARRAYS**
- 4. ADDRESS OF EACH ELEMENT IN AN ARRAY**
- 5. ACCESSING & MANIPULATING AN ARRAY USING POINTERS**
- 6. ANOTHER CASE OF MANIPULATING AN ARRAY USING POINTERS**
- 7. TWO-DIMENSIONAL ARRAY**
- 8. POINTER ARRAYS**
- 9. STRUCTURES**
- 10. STRUCTURE POINTERS**

1. ADDRESS

For every variable there are two attributes: address and value

In memory with address 3: value: 45.
In memory with address 2: value "Dave"

1	4096
2	"Dave"
3	45
4	"Matt"
5	95.5
6	"wbru"
7	0
8	"zero"

```
int main() {  
    int y=7;  
    cout << "Value of y is:" << y << "\n";  
    cout << "Address of y is:" << &y << "\n";  
    return 0;  
}
```

Value of y is:7
Address of y is:0x70fe3c

2. POINTERS

1. *is a variable whose value is also an address.*
2. *A pointer to an integer is a variable that can store the address of that integer*

ia: value of variable

&ia: address of ia

***ia** means you are printing the value at the location specified by ia

1000, i	10
4000, ia	—, 1000

```
int i;    //A
int * ia; //B
cout<<"The address of i "<< &i << " value="<<i <<endl;
cout<<"The address of ia " << &ia << " value = " << ia<< endl;

i = 10;   //C
ia = &i;  //D

        cout<<"after assigning value:"<<endl;
cout<<"The address of i "<< &i << " value="<<i <<endl;
cout<<"The address of ia " << &ia << " value = " << ia<< " point to: "<< *ia;
```

The address of i 0x6ffe3c value=0

The address of ia 0x6ffe30 value = 0x47

after assigning value:

The address of i 0x6ffe3c value=10

The address of ia 0x6ffe30 value = 0x6ffe3c point to: 10

Points to Remember

- **Pointers give a facility to access the value of a variable indirectly.**
- **You can define a pointer by using * before the name of the variable.**
- **You can get the address where a variable is stored by using &.**

3. ARRAYS

- 1. *An array is a data structure***
- 2. *used to process multiple elements with the same data type when a number of such elements are known.***
- 3. *An array is a composite data structure; that means it had to be constructed from basic data types such as array integers.***

- 1. *int a[5];***
- 2. *for(int i = 0;i<5;i++)***
 - 1. *{a[i]=i; }***

4. ADDRESS OF EACH ELEMENT IN AN ARRAY

Each element of the array has a memory address.

```
void printdetail(int a[])
{
    for(int i = 0;i<5;i++)
    {
        cout<< "value in array "<< a[i] <<" at address: " << &a[i]);
    }
}
```


5. ACCESSING & MANIPULATING AN ARRAY USING POINTERS

- *You can access an array element by using a pointer.*
- *If an array stores integers->use a pointer to integer to access array elements.*

```
void printarr_usingpointer(int a[])
{
    int *pi;
    pi=a;
    for(int i = 0;i<5;i++)
    {
        cout<<"value array:" << *pi << " address: " << pi<<endl;
        pi++;
    }
}
```

6. ANOTHER CASE OF MANIPULATING AN ARRAY USING POINTERS

The array limit is a pointer constant : cannot change its value in the program.

```
int a[5]; int *b;  
a=b; //error  
b=a; //OK
```

```
void print_usingptr_a(int a[])  
{  
    It works correctly even using  
    a++ ???  
    for(int i = 0; i<5; i++)  
        {cout<<"a[" << i << "]= " <<*a<<endl;  
          a++;}  
}
```

7. TWO-DIMENSIONAL ARRAY

int a[3][2];

```
int a[3][2];
for(int i = 0;i<3;i++)
    for(int j=0;j<2 ;j++)
    {
        {
            a[i][j]=i+j+i*j;
        }
    }
```

```
void print_usingptr(int a[][2])
{
    int *b;
    b=a[0];
    for(int i = 0;i<6;i++)
    {
        cout<<"value :" << *b<<" in address: " <<b<<endl;
        b++; }
}
```

8. POINTER ARRAYS

- *You can define a pointer array (similarly to an array of integers).*
- *In the pointer array, the array elements store the pointer that points to integer values.*

```
int *a[5];
main()
{   int i1=4,i2=3,i3=2,i4=1,i5=0;
    a[0]=&i1;
    a[1]=&i2;
    a[2]=&i3;
    a[3]=&i4;
    a[4]=&i5;

    printarr(a);
    printarr_usingptr(a);
}
```

```
void printarr(int *a[])
{
    printf("Address1\tAddress2\tValue\n");
    for(int j=0;j<5;j++)
    {
        cout<<*a[j]<<"\t"<<a[j]<<"\t"<<&a[j]<<endl;
    }
}
```

9. STRUCTURES

- *Structures are used when you want to process data of multiple data types*
- *But you still want to refer to the data as a single entity*
- *Access data:
structurename.membername*

```
struct student
{
    char name[30];
    float marks;
};

main ( )
{
    student student1;
    char s1[30];float f;
    cin>>student1.name;
    cin>>student1.marks;

    cout<<"Name is:"<<student1.name;
    cout<<"Marks are:" << student1.marks;
}
```

10. STRUCTURE POINTERS

Process the structure using a structure pointer

```
struct student
{   char name[30];
    float marks;    };
main ( )
{
    struct student *student1;
    struct student student2;
    student1 = &student2;
    cin>>student1->name; // cin>>(*student1).name;
    cin>>student2.marks;
    cout<<(*student1).name<<" : " << student2.marks;
}
```

FUNCTION & RECURSION

- 1. FUNCTION
- 2. THE CONCEPT OF STACK
- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
- 4. PARAMETER PASSING
- 5. CALL BY REFERENCE
- 6. RESOLVING VARIABLE REFERENCES
- 7. RECURSION
- 8. STACK OVERHEADS IN RECURSION
- 9. WRITING A RECURSIVE FUNCTION
- 10. TYPES OF RECURSION

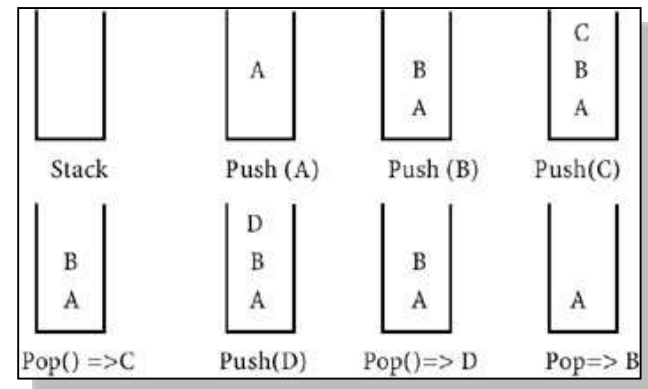
- 1. FUNCTION

- provide modularity to the software
- divide complex tasks into small manageable tasks
- avoid duplication of work

```
int add (int x, int y)
{
    int z;
    z = x + y;
    return z;
}
```


- 2. THE CONCEPT OF STACK

- A *stack* is memory in which values are stored and retrieved in "last in first out" manner by using operations called *push* and *pop*.



- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL

- When the function is called, the current execution is temporarily stopped and the control goes to the called function. After the call, the execution resumes from the point at which the execution is stopped.
- To get the exact point at which execution is resumed, the address of the next instruction is stored in the stack. When the function call completes, the address at the top of the stack is taken.

- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
 - Functions or sub-programs are implemented using a stack.
 - When a function is called, the address of the next instruction is pushed into the stack.
 - When the function is finished, the address for execution is taken by using the pop operation.

- 3. THE SEQUENCE OF EXECUTION DURING A FUNCTION CALL
- Result:?

```
main ( )
{
    printf ("1 \n");
    printf ("2 \n");
    f1();
    printf ("3 \n");
    printf ("4 \n");
}
void f1()
{
    printf ("f1-5 \n");
    printf ("f1-6 \n");
    f2 ( );
    printf ("f1-7 \n");
    printf ("f1-8 \n");
}
void f2 ( )
{
    printf ("f2-9 \n");
    printf ("f2-10 \n");
}
```

- 4. PARAMETER * REFERENCE PASSING

- *passing by value*

- the value before and after the call remains the same

- *passing by reference*

- *changed value after the function completes*

```
void (int *k)
{
    *k = *k + 10;
}
```

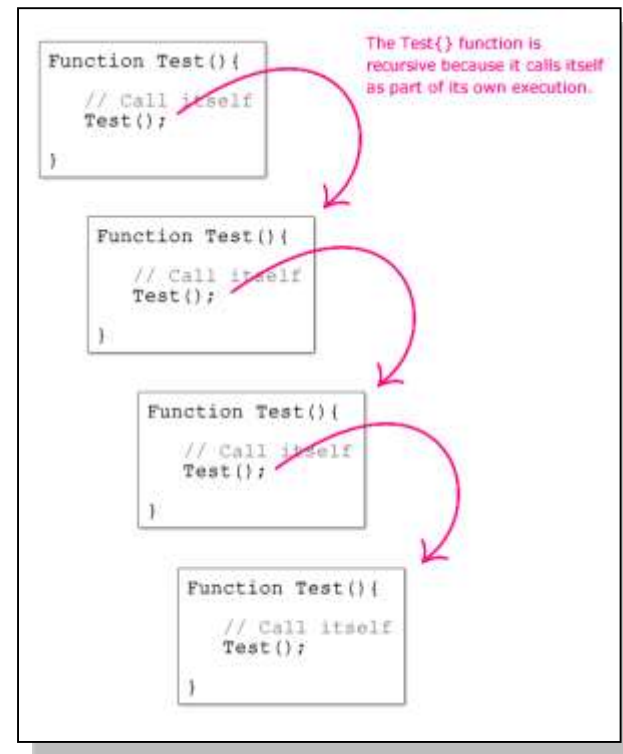
```
void (int &k)
{
    k = k + 10;
}
```

- 6. RESOLVING VARIABLE REFERENCES

When a variable can be resolved by using multiple references, the local definition is given more preference

```
int i =0; //Global variable
main()
{
    int i ;    // local variable for main
    void f1(void) ;
    i =0;
    cout<<"value of i in main: "<< i <<endl;
    f1();
    cout<<"value of i after call:" << i<<endl;
}
void f1()
{
    int i=0;    //local variable for f1
    i = 50;
}
```

- 7. RECURSION
 - A method of programming whereby a function directly or indirectly calls itself
 - Problems: stop recursion?



- 7. RECURSION

Example: Factorial

- The factorial is defined in this way:

$$n! = \begin{cases} 1 & n = 0 \\ 1 \times 2 \times \cdots \times n & n > 0 \end{cases}$$

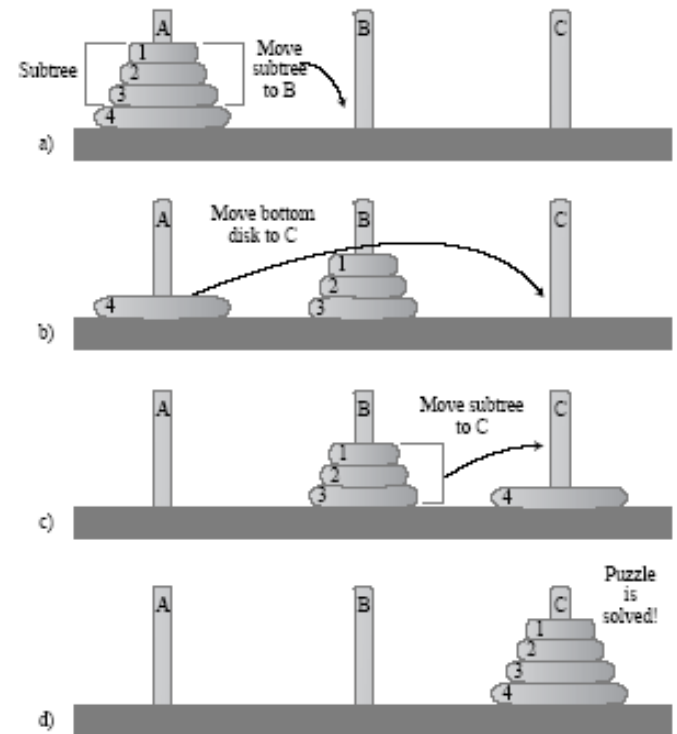
This can be computed by a loop.

- We can also define the factorial as:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

So, if we know how to compute the factorial of $n-1$, we know how to compute the factorial of n .

- 7. RECURSION: Hanoi tower
- Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
 - 1) Only one disk can be moved at a time.
 - 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
 - 3) No disk may be placed on top of a smaller disk.
- This problem can be solved programmatically via recursion.



- 7. RECURSION

Example: Factorial

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

To see how the computation is done, trace factorial(3):

```
factorial(3) = 3 * factorial(2)
              = 3 * (2 * factorial(1))
              = 3 * (2 * (1 * factorial(0)))
              = 3 * (2 * (1 * 1))
```

- 8. STACK OVERHEADS IN RECURSION

- two important results: the depth of recursion and the stack overheads in recursion

- 9. WRITING A RECURSIVE FUNCTION

- Recursion enables us to write a program in a natural way. The speed of a recursive program is slower because of stack overheads.
- In a recursive program you have to specify recursive conditions, terminating conditions, and recursive expressions.

- 10. TYPES OF RECURSION
 - LINEAR RECURSION
 - TAIL RECURSION
 - BINARY RECURSION
 - EXPONENTIAL RECURSION
 - NESTED RECURSION
 - MUTUAL RECURSION

- 10. TYPES OF RECURSION

- **LINEAR RECURSION**

- only makes a single call to itself each time the function runs
 -

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

- 10. TYPES OF RECURSION

- **TAIL RECURSION**

- Tail recursion is a form of linear recursion.
 - In tail recursion, the recursive call is the last thing the function does. Often, the value of the recursive call is returned.

```
// An example of tail recursive
function
void print(int n)
{
    if (n < 0) return;
    cout << " " << n;

    // The last executed statement is
    recursive call
    print(n-1);
}
```

```
int gcd(int m, int n)
{
    int r;
    if (m < n) return gcd(n,m);
    r = m%n;
    if (r == 0) return(n);
    else return(gcd(n,r));
}
```

- 10. TYPES OF RECURSION

- **BINARY RECURSION**

- Some recursive functions don't just have one call to themselves, they have two (or more).

```
int choose(int n, int k)
{
    if (k == 0 || n == k) return(1);
    else return(choose(n-1,k) + choose(n-1,k-1));
}
```


- 10. TYPES OF RECURSION

- **EXPONENTIAL RECURSION**

- An exponential recursive function is one that, if you were to draw out a representation of all the function calls, would have an exponential number of calls in relation to the size of the data set
 - (exponential meaning if there were n elements, there would be $O(a^n)$ function calls where a is a positive number)

- 10. TYPES OF RECURSION
 - **EXPONENTIAL RECURSION**

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

void print_permutations(int arr[], int n, int i)
{
    int j, swap;
    print_array(arr, n);
    for(j=i+1; j<n; j++) {
        swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
        print_permutations(arr, n, i+1);
        swap = arr[i]; arr[i] = arr[j]; arr[j] = swap;
    }
}
```

- 10. TYPES OF RECURSION

- NESTED RECURSION

- In nested recursion, one of the arguments to the recursive function is the recursive function itself
 - These functions tend to grow extremely fast.

```
int ackerman(int m, int n)
{
    if (m == 0) return(n+1);
    else if (n == 0) return(ackerman(m-1,1));
    else return(ackerman(m-1,ackerman(m,n-1)));
}
```

- 10. TYPES OF RECURSION

- **MUTUAL RECURSION**

- A recursive function doesn't necessarily need to call itself.
 - Some recursive functions work in pairs or even larger groups. For example, function A calls function B which calls function C which in turn calls function A.

- 10. TYPES OF RECURSION
 - **MUTUAL RECURSION**

```
int is_even(unsigned int n)
{
    if (n==0) return 1;
    else return(is_odd(n-1));
}

int is_odd(unsigned int n)
{
    return (!is_even(n));
}
```