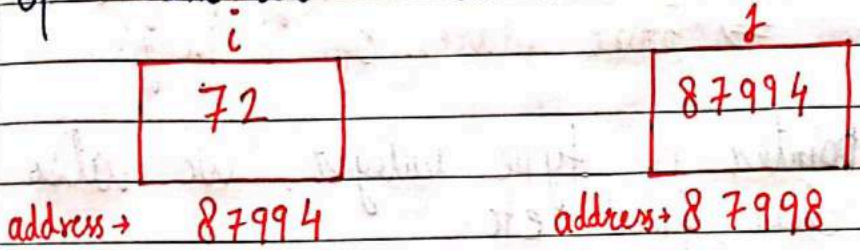
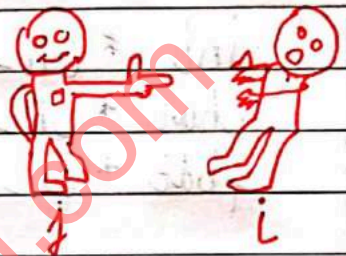
URBAN EDGE

Chapter 6 - Pointers

A pointer is a variable which stores the address of another variable



j is a pointer
j points to i



The "address of" (&) operator
The address of operator is used to obtain the address of a given variable

If you refer to the diagrams above

&i ⇒ 87994
&j ⇒ 87998

Format specifier for printing pointer address is '%u'

The 'value at address' operator (*)
The value at address or * operator is used to obtain the value present at a given memory address. It is denoted by *

*(&i) = 72
*(&j) = 87994

How to declare a Pointer ?
A pointer is declared using the following Syntax

```
int *j;      ⇒ declare a variable j of type int-pointer
j = &i       ⇒ Store address of i in j
```

Just like pointer of type integer, we also have pointers to char, float etc.

```
int *ch_ptr;   → Pointer to integer
char *ch_ptr;  → Pointer to character
fbdt *ch_ptr;  → Pointer to float
```

Although its a good practice to use meaningful variable names, we should be very careful while reading & working on porgrams from fellow programmers.

A Program to demonstrate pointers

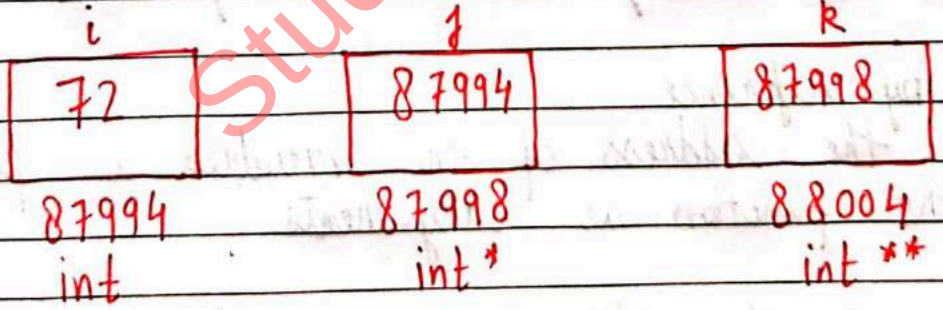
```
#include<stdio.h>
int main() {
   int i=8;
   int *j;
   j = &i;
  printf(" Add i = %u\n", &i);
  printf(" Add i = %u \n", j);
  printf(" Add j = %u \n", &j);
  printf(" Value i = %d \n", i);
  printf(" Value i = %d \n", *(&i));
  printf(" Value i = %d \n", *j);
  return 0;
}
```

Output:
Add i = 87994
Add i = 87994
Add j = 87998
Value i = 8
Value i = 8
Value i = 8

This program sums it all. If you understand it, you have got the idea of pointers.

Pointer to a pointer
Just like j is pointing to i or storing the address of i, we can have another variable k which can further store the address of j. What will be the type of k

```
int **k;
k = &j;
```



We can even go further one level and create a variable l of type int*** to store the address of k. We mostly use int* and int** sometimes in real world programs.

Types of function calls
Based on the way we pass arguments to the function, function calls are of two types:

1. Call by value → Sending the values of arguments
2. Call by reference → Sending the address of arguments

Call by value
Here the value of the arguments are passed to the function. Consider this example:

int c = Sum(3, 4); ⇒ assume x=3 and y=4
        x, y

if Sum is defined as Sum(int a, int b), the values 3 and 4 are copied to a and b. Now even if we change a and b, nothing happens to the variables x and y.
This is call by value.

In C we usually make a call by value.

Call by reference
Here the address of the variables is passed to the function as arguments.

Now since the addresses are passed to the function, the function can now modify the value of a variable in calling function using * and & operators. Example:

URBAN EDGE

```
Void Swap (int *x, int *y)
{
  int temp;
  temp = *x;
  *x = *y;
  *y = temp;
}
```

This function is capable of swapping the values passed to it. if a = 3 and b = 4 before a call to swap(a, b), a = 4 and b = 3 after calling swap.

```
int main() {
int a = 3
int b = 4        => a is 3 and b is 4
Swap(a,b)
return 0;        => Now a is 4 and b is 3
}
```

URBAN EDGE

Chapter 7 - Arrays

An array is a collection of similar (imp) elements.

One variable ⇒ capable of storing multiple values

Syntax

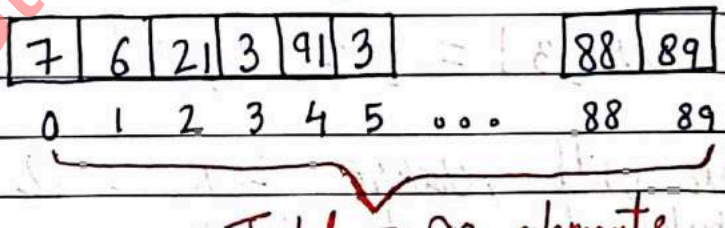
The syntax of declaring an Array looks like this:

```
int marks[90];    ⇒ Integer array
char name[20];    ⇒ Character array or String
float percentile[90];  ⇒ float array
```

The values can now be assigned to marks array like this:

```
marks[0] = 33;
marks[1] = 42;
```

Note: It is very important to note that the array index starts with 0.



URBAN EDGE

Accessing elements
Elements of an array can be accessed using:

scanf ("%d", &marks[0]); ⇒ Input first value

printf ("%d", marks[0]); ⇒ output first value of the array

Quick Quiz → Write a program to accept marks of five students in an array and print them to the screen.

Initialization of an Array
There are many other ways in which an array can be initialized.

int cgpa[3] = {9, 8, 8} ⇒ Arrays can be initialized while declaration
float marks[] = {33, 40}

Arrays in memory
Consider this array:

int arr[3] = {1, 2, 3} ⇒ 1 integer = 4 bytes

This will reserve 4×3 = 12 bytes in memory 4 bytes for each integer.

1	2	3
62302	62306	62310

⇒ arr in memory

URBAN EDGE

Pointer Arithmetic

A pointer can be incremented to point to the next memory location of that type.

Consider this example

```
int i = 32;
int *a = &i;   => a = 87994
a++;           => Now a = 87998
```

i → 32

address → 87994

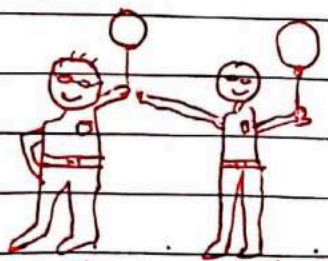
```
char a = 'A';
char *b = &a;  => b = 87994
b++;           => Now b = 87995
```

```
float i = 1.7;
float *a = &i; => Address of i or a = 87994
a++;           => Now a = 87998
```

Following operations can be performed on a pointers:

1. Addition of a number to a pointer
2. Subtraction of a number from a pointer
3. Subtraction of one pointer from another
4. Comparison of two pointer variables

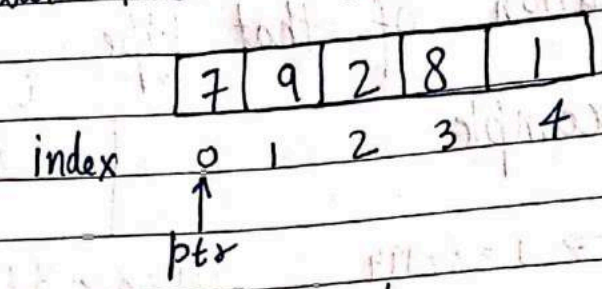
Quick Quiz → Try these operations on another variable by creating pointers in a separate program. Demonstrate all the four operations.



Yay! we understood pointer arithmetic

Accessing Arrays using pointers

Consider this array



If ptr points to index 0, ptr++ will point to index 1 & so on...

This way we can have an integer pointer pointing to first element of the array like this:

```
int *ptr = &arr[0]; → or simply arr
ptr++;
*ptr => will have 9 as its value
```

Passing arrays to functions

Arrays can be passed to the functions like this

```
printArray(arr,n);      => function call
Void printArray(int *i, int n); => function prototype
        or
Void printArray(int i[], int n);
```

URBAN EDGE

Multidimensional Arrays

An array can be of 2 dimension / 3 dimension / n dimensions

A 2 dimensional array can be defined as:

```
int arr[3][2] = { { 1, 4 }
                  { 7, 9 }
                  { 11, 22 } };
```

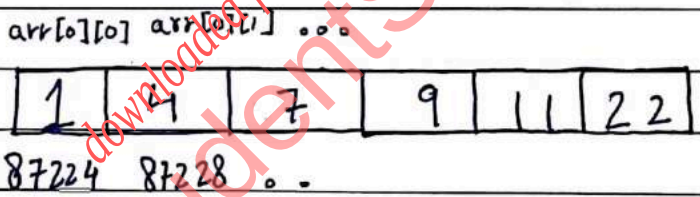
We can access the elements of this array as arr[0][0], arr[0][1] & so on...

arr[0][0] → Value = 1

arr[0][1] → Value = 4

2-D arrays in Memory

A 2d array like a 1-d array is stored in contiguous memory blocks like this:



Quick Quiz: Create a 2-d array by taking input from the user. Write a display function to print the content of this 2-d array on the screen.

Chapter 11 - Dynamic Memory Allocation

C is a language with some fixed rules of programming. For example: changing the size of an array is not allowed.

Dynamic Memory Allocation

Dynamic memory allocation is a way to allocate memory to a data structure during the runtime. We can use DMA functions available in C to allocate and free memory during runtime.

Functions for DMA in C

Following functions are available in C to perform Dynamic memory Allocation:

1. malloc()
2. calloc()
3. free()
4. realloc()

malloc() function

malloc stands for memory allocation. It takes number of bytes to be allocated as an input and returns a pointer of type void

Syntax:

```
ptr = (int*) malloc(30 * sizeof(int))
```

- (int*): casting void pointer to int
- 30: space for 30 ints
- sizeof(int): returns size of 1 int

The expression returns a null pointer if the memory cannot be allocated.

Quick Quiz: Write a program to create a dynamic array of 5 floats using malloc().

Calloc() function
calloc stands for continuous allocation.
It initializes each memory block with a default value of 0.

Syntax:

```
ptr = (float*) calloc (30, sizeof(float));
```

↓ Allocates contiguous space in memory for 30 blocks (floats)

If the space is not sufficient, memory allocation fails and a NULL pointer is returned.

Quick Quiz: Write a program to create an array of size n using calloc where n is an integer entered by the user.

free() function
We can use free() function to de allocate the memory.
The memory allocated using calloc/malloc is not deallocated automatically.

Syntax:

free(ptr); ⟹ Memory of ptr is released.

Quick Quiz: Write a program to demonstrate the usage of free() with malloc().

realloc() function

Sometimes the dynamically allocated memory is insufficient or more than required.

realloc is used to allocate memory of new size using the previous pointer and size.

Syntax:

ptr = realloc(ptr, newSize);

ptr = realloc(ptr, 3 * sizeof(int));

⇓

ptr now points to this new block of memory capable of storing 3 integers.

URBAN EDGE

Chapter 9 - Structures

Arrays and strings ⇒ Similar data (int, float, char)

Structures can hold ⇒ dissimilar data

Syntax for creating Structures
A C structure can be created as follows:

```
struct employee {
    int code;
    float salary;
    char name[10];
};
```

⇒ This declares a new user defined data-type!

→ Semicolon is important

We can use this user defined data type as follows:

```
Struct employee e1;
Strcpy(e1.name, "Harry");
e1.code = 100;
e1.salary = 71.22;
```

⇒ Creating a structure variable

So a structure in C is a collection of variables of different types under a single name.

Quick Quiz: Write a program to store the details of 3 employees from user defined data. Use the structure declared above:

Why use structures?
We can create the data types in the employee structure separately but when the number of properties in a structure increases, it becomes difficult for us to create data variables without structures. In a nut shell:

(a) Structures keep the data organized.
(b) Structures make data mangement easy for the programmer.

Array of Structures
Just like an array of integers, an array of floats and an array of characters, we can create an array of structures.

struct employee facebook[100]; ⇒ An array of structures

We can access the data using:
facebook[0].code = 100;
facebook[1].code = 101;
... & so on

Initializing Structures
Structures can also be initialized as follows:

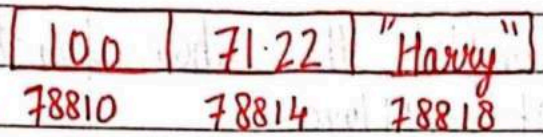
struct employee harry = {100, 71.22, "Harry"};

struct employee shubh = {0}; ⇒ All elements set to 0

URBAN EDGE

Structures in memory

Structures are stored in contiguous memory locations
For the structure e1 of type struct employee, memory layout looks like this:



In an array of structures, these employee instances are stored adjacent to each other.

Pointer to structures

A pointer to structure can be created as follows:

```
struct employee *ptr;
ptr = &e1;
```

Now we can print structure elements using:

```
printf("%d", *(ptr).code);
```

Arrow Operator

Instead of writing *(ptr).code, we can use arrow operator to access structure properties as follows

*(ptr).code or ptr -> code

Here -> is known as the arrow operator.

URBAN EDGE

Passing Structure to a function

A structure can be passed to a function just like any other data type.

```
Void show (struct employee e);
```
⇒ function prototype

Quick Quiz : Complete this show function to display the content of employee.

Typedef keyword

We can use the typedef keyword to create an alias name for data types in C. typedef is more commonly used with structures.

```
struct complex {
    float real;
    float img;
};
```
⇒ struct complex C_1, C_2;
for defining Complex numbers

```
typedef struct complex {
    float real;
    float img;
} ComplexNo;
```
⇒ ComplexNo C_1, C_2;
for defining Complex numbers

Ans 1) Typedef: Every variable has a datatype, typedef is a word to define new data type name to make a program more readable to the programmer.

Example:

```
main ( )
{
  int money;
  money = 2;
}
```

```
main ( )
{
typedef int pounds;
pounds money = 2;
}
```

Ans 2) Enum datatype: Enumerative or Enum in C is a special kind of data type defined by the user. It consists of constant integrals or integers that are given names by a user. The use of enum in C to name the integer values is to make the entire program easy to learn, understand, and maintain by the same or even different program.

Example 1: Printing the values of week days

```
# include < stdio . h >
    enum days { Sunday = 1, Monday = 2, Tuesday = 3,
                Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7 }
```

```
int main () {
    for (int i = Sunday; i<=Saturday; i++){
        printf ("%d,", i)
    }
    return 0;
```

Output : 1, 2, 3, 4, 5, 6, 7,

ans 3) Multi File C Programs : A large C or C++ program should be divided into multiple files. This makes each file short enough to conveniently edit, print etc. It also allows some of the code e.g. utility functions such as linked list handlers or array allocation code to be shared with other programs

EXAMPLE :

First Program [second.c]

```
int sum (int a, int b){
    return a+b;
}
int sub (int a, int b) {
    return a-b;
}
int multiply (int a, int b){
    return a*b;
}
```

Second Program

```
#include "second.c"
#include <stdio.h>
int main () {
    int a=5, b=5;
    int ans = sum (a, b);
    printf ("Sum : %d", ans);
    ans = sub (a, b);
    printf ("Subtraction : %d", ans);
    ans = multiply (a, b);
    printf ("Multiply : %d", ans);
    return 0;
}
```

OUTPUT : Sum : 9
Subtraction : -1
Multiply : 20

Unions in C

Union is a user-defined data type in C language that can contain elements of the different data types like structure. But unlike structure, all the members of C union are stored in same memory location.
So, only one member at a time can be accessed.

Syntax :-

```
Union Example
{
    char x;
    float y;
} obj;
```

ways to define a union variable :-

1. with union declaration
2. After union declaration

We need to define a variable of the union type using union members.

Variable with declaration	Variable after declaration
Union union_name { datatype member1; datatype member2; -- } var1, var2, --- ;	Union union_name var1, var2, var3 ---- ; where Union_name is the name of an already declared union.

Access Union members

var1.member1;

where, var1 = union variable
member1 = member of union.

Initialization

var1.member1 = Some_value;

Remember: only one member can contain some value at a given instance of time.

PAGE NO.
DATE: / /

Example

```
Union un {
    int member1;
    char member2;
    float member3;
};

// driver code
int main()
{
// defining a union variable
Union un var1;

// initializing the union member
var1.member1 = 15;

Printf("the value stored in member 1 = %",
        var1.member1);
Return 0;
```

Output

value stored in member1 = 15

Nested Unions

- These are unions having another union as a member in that union.
- A member of union can be union itself; this what we call as a nested union.
- Examples ahead,

Example

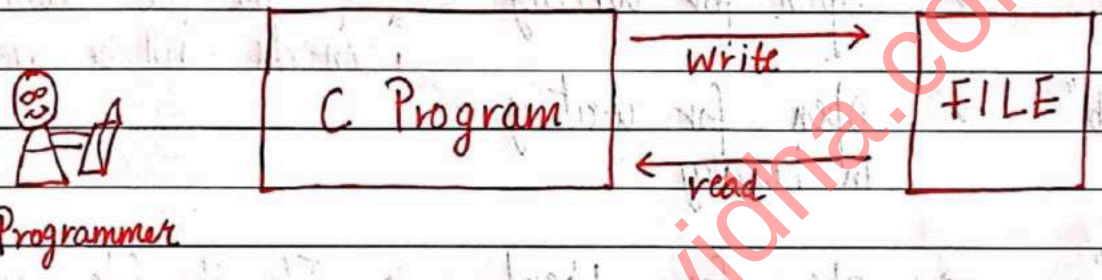
```
Union name {
    char first name [20];
    char last name [20];
};
Union student
{
    int rollno;
    char div;
    union name n1;
};
```

Here union name is a pre defined union in second union.

Chapter 10 - File I/O

The Random Access Memory is volatile and its content is lost once the program terminates. In order to persist the data forever we use files.

A file is data stored in a storage device. A C program can talk to the file by reading content from it and writing content to it.



FILE pointer

The "FILE" is a structure which needs to be created for opening the file.

A file pointer is a pointer to this structure of the file.

FILE pointer is needed for communication between the file and the program.

A FILE pointer can be created as follows:

```
FILE *ptr;
ptr = fopen("filename.ext", "mode");
```

EDGE

File opening modes in C

C offers the programmers to select a mode for opening a file.

Following modes are primarily used in C File I/O

"r" → open for reading → If the file does not exist, fopen returns NULL

"rb" → open for reading in binary → If the file does not exist, fopen returns NULL

"w" → open for writing → If the file exists, the contents will be overwritten

"wb" → open for writing in binary → If the file exists, the contents will be overwritten

"a" → open for append → If the file does not exist, it will be created

Types of files

There are two types of files:

1. Text files (.txt, .c)
2. Binary files (.jpg, .dat)

Reading a file

A file can be opened for reading as follows:

```
FILE *ptr;
ptr = fopen("Harry.txt", "r");
int num;
```

Let us assume that "Harry.txt" contains an integer We can read that integer using:

```
fscanf (ptr, "%d", &num);
```

⇒ fscanf is file counterpart of scanf

This will read an integer from file in num variable.

Quick Quiz: Modify the program above to check whether the file exists or not before opening the file.

CLOSING the file

It is very important to close the file after read or write. This is acheived using fclose as follows:

```
fclose (ptr);
```

This will tell the compiler that we are done working with this file and the associated resources could be freed.

Writing to a file

We can write to a file in a very similar manner like we read the file

```
FILE *fptr;
fptr = fopen ("Harry.txt", "w");
```

```
int num = 432;
fprintf (fptr, "%d", num);
fclose (fptr);
```

fgetc() and fputc()
fgetc and fputc are used to read and write a character from/to a file

```
fgetc (ptr)
```
⟹ used to read a character from file

```
fputc ('c', ptr);
```
⟹ used to write character 'c' to the file

EOF : End of File
fgetc returns EOF when all the characters from a file have been read. So we can write a check like below to detect end of file

```
While (1) {
    ch = fgetc (ptr);
    if (ch == EOF) {
        break;
    }
// code
}
```

⟹ When all the content of a file has been read, break the loop!

C standard Libraries

**] Stdio.h :-

The file stdio.h is a built-in header file in C. The acronym stdio stands for Standard Input and Output.

Syntax : # include <stdio.h>

5 functions of stdio.h are :

1. printf() :

This function is used to print the character, string, float, integer, octal and hexadecimal values onto the output screen.

2. scanf() :

This function is used to read a character, string, numeric data from keyboard.

3. gets() :

It reads line from keyboard.

4. puts() :

It ~~dis~~ writes line to the output screen.

5. remove() :

It deletes the given file.

[***] stdlib.h :-

It is a header file and also the standard Library of C-programming language that declares various utility functions for type conversions, memory allocation, algorithms and other similar use cases.

Syntax: # include <stdlib.h>

5 functions of stdlib.h are :

1. malloc() :
This function is used to allocate space in memory during the execution of the program.

2. calloc() :
Similar to malloc(), but calloc() initializes the allocated memory to zero.

3. realloc() :
This function modifies the allocated memory size by malloc() and calloc() functions to new size.

4. free() :
This function frees the allocated memory by malloc(), calloc(), realloc() functions and returns the memory to the system.

5. abort() :
This function terminates the C-program.

[***] Assert.h :-

Syntax: # include <assert.h>

Function of assert.h :

Assert :
This is actually a macro and not a function, which can be used to add diagnostics in your C-program.

[***] math.h :-

It is a header file that declares a set of functions to perform mathematical operations.

Syntax: # include <math.h>

5 functions of math.h are :

1. Sqrt ():
   It is used to find the square root of a number.
2. pow ():
   This function returns the value of x to the power of y (x^y)
3. abs ():
   This function returns the absolute value of x.
4. Cbrt ():
   This function returns the cube root.
5. exp (x):
   This function returns the value of E^x.

[**] time.h :-

The time.h header file contains definitions of functions to get and manipulate date and time information.

Syntax: # include <time.h> Functions are ↓

1. setdate (): This function is used to modify the system date.
2. getdate (): This function is used to get the CPU ~~date~~ time.
3. time (): This function is used to get current system time as structure.
4. mktime (): This func. interprets tm structure as calendar time.
5. asctime (): Time structure contents are interpreted by this func. as calendar time. This time is converted into string.

[***] ctype.h :-

The C ctype.h header file declares a set of functions to classify (and transform) individual characters.

Syntax: #include <ctype.h>

5 functions of ctype.h are:

1. isalpha(): checks whether character is alphabetic.
2. isdigit(): checks whether character is digit.
3. isspace(): checks whether character is space.
4. isalnum(): checks whether character is alphanumeric.
5. isprint(): check whether character is printable character.

[***] setjmp.h :-

It is a header defined in the C standard library to provide "non-local jumps": control flow that deviates from the usual subroutine call and return sequence.

Syntax: #include <setjmp.h>

Functions of setjmp.h are:

longjmp():

This function restores the environment saved by the most recent call to setjmp() macro in the same invocation of the program with the corresponding jmp_buf argument.

[***] string.h :-

This header file must be included in the C-program to use the string handling functions.

Syntax: # include <string.h>

5 function of string.h are :

1. strcat() : concatenates str2 at the end of str1.
2. strcpy() : copies str2 into str1.
3. strlen() : Gives the length of str1.
4. strchr() : Returns pointer to first occurrence of char in str1
5. strrev() : Reverses the given string.

[***] stdarg.h :-

It is a header file in the C-standard library of the C-programming language that allows functions to accept an indefinite number of arguments.

The following macros are defined :

1. va_start() : initializes the list of variable arguments.
2. va_arg() : Expands the next argument in the parameter list of the function with a type.
3. va_end() : Allows a func. with variable arguments to return.

There is one type described in stdarg.h :

va_list() : Hold the information about variable arguments.
The typedef va_list is used by the macros.

[***] unistd.h :-

The <unistd.h> header defines miscellaneous symbolic constants and types, and declares miscellaneous functions.

Syntax: # include <unistd.h>

5 functions of unistd.h are:

1. crypt : password and data encryption
2. encrypt : encrypt 64-bit messages
3. gethostid : get the unique identifier of the current host.
4. swab : swap adjacent bytes.
5. sysconf : get configuration at run time.