

TATA ELXSI

engineering creativity

C Programming

Day -2

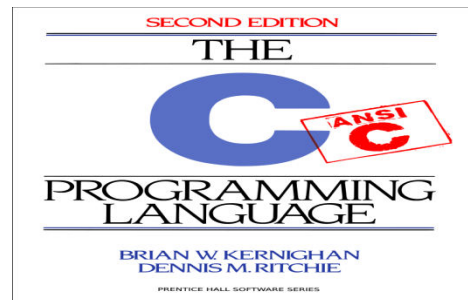


Disclaimer

- This material is developed for in-house training at TATA ELXSI.
- The training material therefore was solely developed for educational purposes. Currently, commercialization of the prototype does not exist, nor is the prototype available for general usage. TATA ELXSI, its staff, its students or any other participants can use it as a part of training. This material should not be found on any website.
- For the preparation of this material we have referred from the below mentioned links or books. Excerpts of these material has been taken where there is no copy right infringements. Some examples and concepts have been sourced from the below links and are open source material

<http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>

ANSI C K&R



Day – 2 Agenda

- functions
- Arrays
- Need for Arrays
- Single Dimensional Array
- Initialization of Array Elements
- Partial initialization of Array Elements
- Two Dimensional Array
- Three Dimensional Array

Functions



Functions introduction

- Functions are block of instructions, which break large computing tasks into smaller ones.
- A function is defined as a name given to an independent section of C code that performs a specific task
- Any C program contains at least one function it must be main().
- There is no limit on the number of functions in a C program.
- A function will not be executed until the function is called by another part of the program.

functions

- Functions have three parts:
 - Function Prototype (declarations) : : consists of information regarding the function's name, return type, and types and names of parameters.
 - Function invocation : Calling the function
 - Function Definitions :To specify what a function actually does (along with the body of the function).
 - The function body is a series of statements enclosed in braces; in fact it is simply a block (see Blocks).

functions

- Here is the general form of a function definition:

```
return-type  function-name (parameter-list)
{
    function-body
}
```

- return-type and function-name are the same as what you use in the function declaration.
- parameter-list is the same as the parameter list used in the function declaration, except you must include names for the parameters in a function definition.

Function: example

```
int fnAdd(int , int) ; //Function prototype
```

```
main()
```

```
{
```

```
    int iNum1, iNum2;
```

```
    int iSum;
```

```
    printf("Enter an integer");
```

```
    scanf("%d",&iNum1);
```

```
    printf("Enter an integer");
```

```
    scanf("%d",&iNum2);
```

```
    iSum = fnAdd(iNum1, iNum2); //Function invocation
```

```
    printf("The sum of %d and %d is %d\n",iNum1,iNum2,iSum);
```

```
}
```

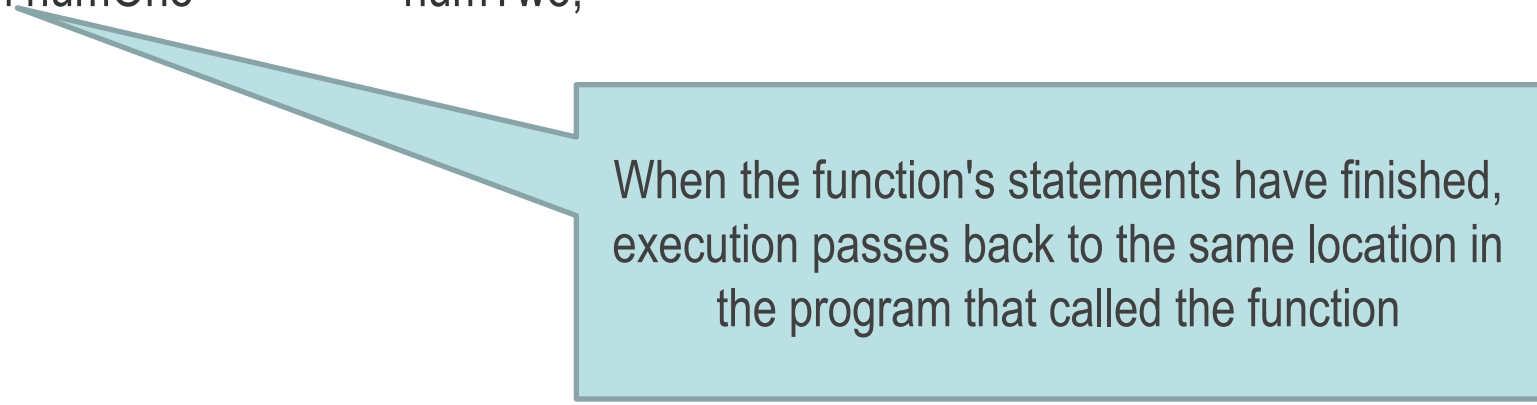
Function signature

An *argument* is a mechanism used to convey information to the function

Function: example

```
/* Function definition */
```

```
int fnAdd(int numOne, int numTwo)  
{  
    return numOne + numTwo;  
}
```



When the function's statements have finished, execution passes back to the same location in the program that called the function

Void function

- If the function has no return value, then it is void function.

```
void swap (int, int) ;  
main()  
{  
    int      iNum1, iNum2;  
    printf("Enter two integers\n");  
    scanf("%d%d",&iNum1,&iNum2);  
    printf("\niNum1 = %d,iNum2 = %d\n",iNum1,iNum2);  
    swap(iNum1, iNum2);      //function invocation  
    printf("\niNum1 = %d,iNum2 = %d\n",iNum1,iNum2);  
}
```

Void function : example

```
void swap(iNum1, iNum2)
{
    int iTemp;
    iTemp = iNum1;
    iNum1 = iNum2;
    iNum2 = iTemp;
}
```

Recursion

The term recursion refers to a function calls itself either directly or indirectly.

Indirect recursion occurs when one function calls another function that then calls the first function.

C allows recursive functions, and they can be useful in some situations.

For Example:

```
main()
{
    puts("I will be printed till stack overflows");
    main();
}
```

```
/* PROGRAM TO FIND THE factorial OF THE NUMBER recursively */  
#include <stdio.h>
```

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

```
main()
```

```
{
```

```
    int num;
```

```
    printf("Enter a number \n");
```

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

```
    printf("%d\n",factorial(num));
```

```
}
```

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

```
{
```

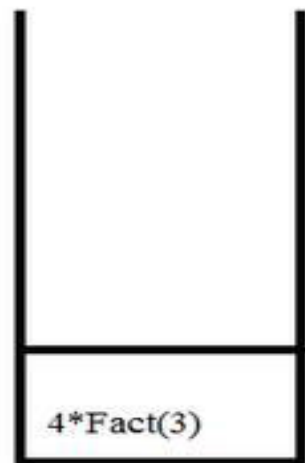
```
    if(n == 0)
```

```
        return 1;
```

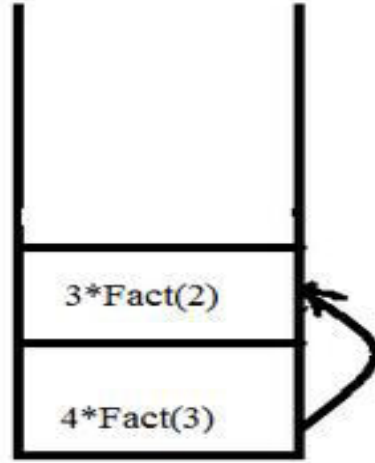
```
    return (n * factorial(n - 1));
```

```
}
```

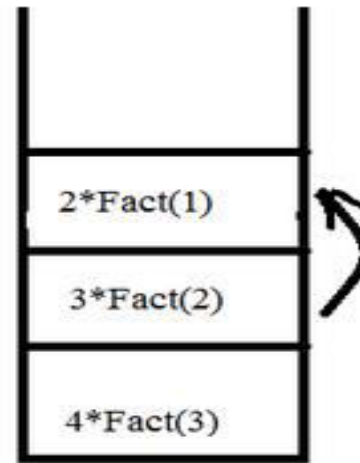
When function call happens previous variables gets stored in stack



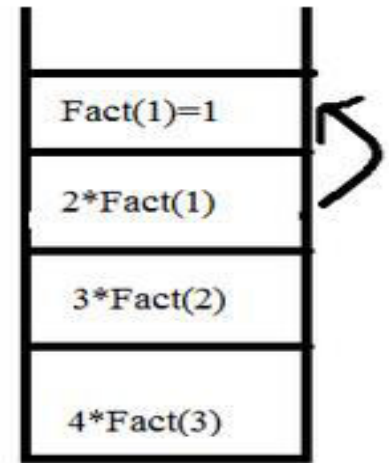
After the first call



second call

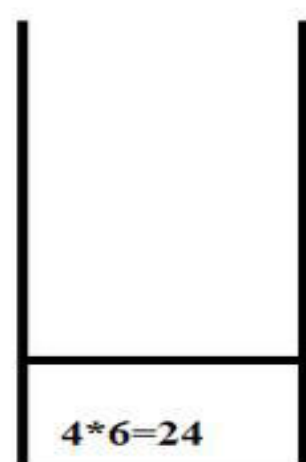
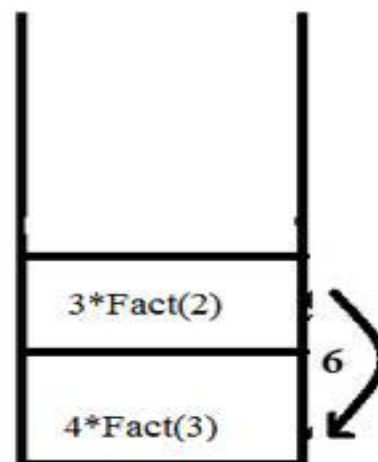
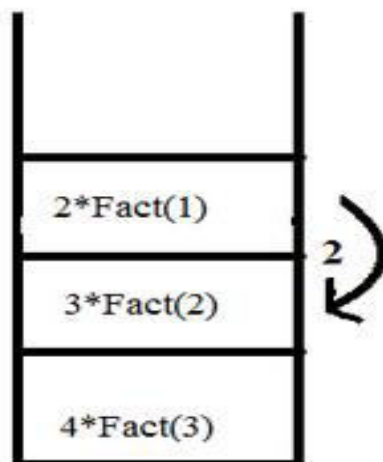
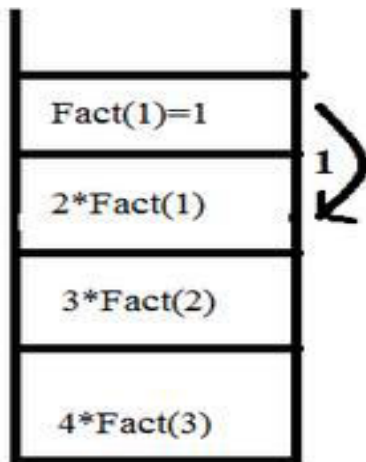


third call



fourth call

Returning values from base case to caller function



More example on recursion

- recursive version of the Fibonacci function

```
int fib(int num)
/* Fibonacci value of a number */
{
    switch(num) {
        case 0:      return(0);
                     break;
        case 1:      return(1);
                     break;
        default: /* Including recursive calls */
                     return(fib(num - 1) + fib(num - 2));
                     break;
    }
}
```


More example on recursion

- recursive version power function

```
double power(double val, unsigned pow)
{
    if(pow == 0) /* pow(x, 0) returns 1 */
        return(1.0);
    else
        return(power(val, pow - 1) * val);
}
```

Storage class



Storage Class Specifiers

There are four storage class specifiers supported by C:

- extern
- static
- register
- auto

These specifiers tell the compiler how to store the subsequent variable.

The general form of a declaration that uses one is shown here.

storage_specifier type **var_name**;

Auto (Local) Variables

Variables that are declared inside a function are called local variables.

In some C/C++ literature, these variables are referred to as automatic variables.

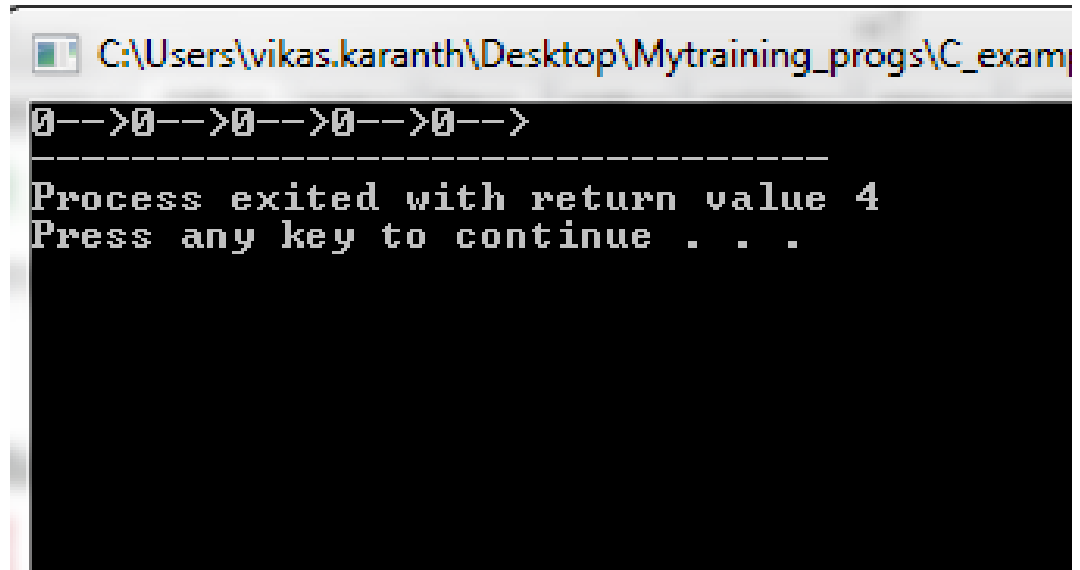
local variable is created upon entry into its block and destroyed upon exit.

Auto variables are stored in stack segment of the process address space.

Auto (Local) Variables: example

```
#include<stdio.h>
void foo(void);

main()
{
    foo();
    foo();
    foo();
    foo();
    foo();
}
void foo(void)
{
    int i = 0;
    printf("%d-->",i++);
}
```



```
C:\Users\vikas.karanth\Desktop\Mytraining_progs\C_exam
0-->0-->0-->0-->0-->
-----
Process exited with return value 4
Press any key to continue . . .
```

Global(extern) Variables

Unlike local variables, global variables are known throughout the program and may be used by any piece of code.

Also, they will hold their value throughout the program's execution.

global variables are created by declaring them outside of any function.

Any expression may access them, regardless of what block of code that expression is in.

Extern Variables: example

```
int count; /* count is global */
void func1(void);
void func2(void);

int main(void)
{
    count = 100;
    func1();
    return 0;
}
```

```
void func1(void)
{
    int temp;
    temp = count;
    func2();
    printf("count is %d", count); /* will
print 100 */
}

void func2(void)
{
    int count;
    for(count=1; count<10; count++)
}
```

extern

C/C++ allows separate modules of a large program to be separately compiled and linked together.

then there must be some way of telling all the files about the global variables required by the program.

extern lets the compiler know what the types and names are for these global variables without actually creating storage for them again.

When the linker links the two modules, all references to the external variables are resolved.

File One

```
int x, y;  
char ch;  
int main(void)  
{  
    /* ... */  
}
```

```
void func1(void)  
{  
    x = 123;  
}
```

File Two

```
extern int x, y;  
extern char ch;  
void func22(void)  
{  
    x = y / 10;  
}
```

```
void func23(void)  
{  
    y = 10;  
}
```

static Variables

static variables are permanent variables within their own function or file.

Unlike global variables, they are not known outside their function or file, but they maintain their values between calls.

This feature makes them useful when you write generalized functions and function libraries that other programmers may use.

static has different effects upon local variables and global variables.

```
void fun(void);  
main()  
{  
    fun();  
    fun();  
    fun();  
    fun();  
}  
void fun(void)  
{  
    static int i = 0;  
    printf("I is %d\n",i++);  
}
```

static global Variables

//app.c

```
void fun(void);
```

```
void fun1 (void);
```

```
static int i = 0;
```

```
main()
```

```
{
```

```
    fun();
```

```
    fun();
```

```
    fun1();
```

```
    fun1();
```

```
}
```

```
void fun(void)
```

```
{
```

```
    printf("I is %d\n",i++);
```

```
}
```

// Lib.c

```
void fun1(void)
```

```
{
```

```
    printf("I is %d\n",i++);
```

```
}
```

Linker error

register Variables

The register storage specifier originally applied only to variables of type int, char, or pointer types. However, in Standard C, register's definition has been broadened so that it applies to any type of variable.

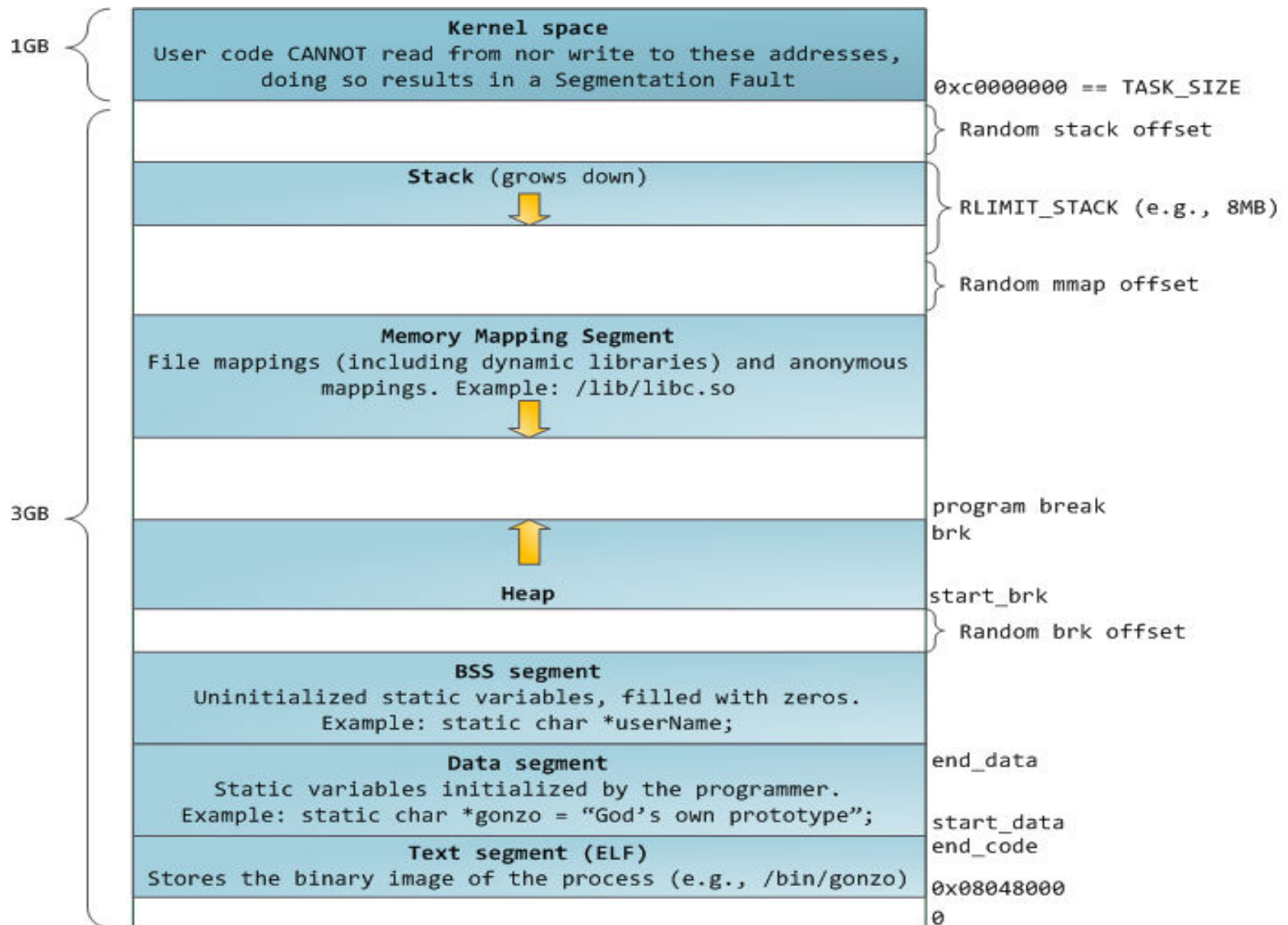
Originally, the register specifier requested that the compiler keep the value of a variable in a register of the CPU rather than in memory, where normal variables are stored.

This meant that operations on a register variable could occur much faster than on a normal variable because the register variable was actually held in the CPU and did not require a memory access to determine or modify its value.

```
int int_pwr(register int m, register int e)
{
    register int temp;
    temp = 1;
    for(; e; e - -) temp = temp * m;
    return temp;
}
```

In this example, `e`, `m`, and `temp` are declared as register variables because they are all used within the loop. The fact that register variables are optimized for speed makes them ideal for control of or use in loops.

In C, you cannot find the address of a register variable using the `&` operator



1) Consider the following C code

```
main(){  
    int i=3,x;  
        while(i>0){  
            x=func(i);  
            i--;  
        }  
    int func(int n){  
        static sum=0;  
        sum=sum+n;  
        return(sum);}
```

The final value of x is

- | | |
|-------|-------|
| (a) 6 | (b) 8 |
| (c) 1 | (d) 3 |

way to examine the contents of an executable file is to use the nm or dump utilities.

Compile the source below, and run nm on the resulting a.out.

```
char pear[40];
static double peach;
int mango = 13;
static long melon = 2001;

main ()
{
    int i=3, j, *ip;
    ip=malloc(sizeof(i)); // memory allocated dynamically in heap
    pear[5] = i;
    peach = 2.0*mango;
}
```

Excerpts from running nm are shown below (minor editing changes have been made to the output to make it more accessible):

```
% nm -sx a.out
```

Symbols from a.out:

[Index]	Value	Size	Type	Bind	Segment	Name
...						
[29]	0x00020790	0x00000008	OBJT	LOCL	.bss	peach
[42]	0x0002079c	0x00000028	OBJT	GLOB	.bss	pear
[43]	0x000206f4	0x00000004	OBJT	GLOB	.data	mango
[30]	0x000206f8	0x00000004	OBJT	LOCL	.data	melon
[36]	0x00010628	0x00000058	FUNC	GLOB	.text	main
[50]	0x000206e4	0x00000038	FUNC	GLOB	UNDEF	malloc

Arrays



Objective

Why Arrays?

What Is an Array?

Single-Dimensional Arrays

Naming and Declaring Arrays

Initializing Arrays

Multidimensional Arrays

Initializing Multidimensional Arrays

Maximum Array Size

Arrays

An *array* is a collection of data storage locations, each having the same data type and the same name.

Each storage location in an array is called an *array element*.

Single-Dimensional Arrays

A *single-dimensional array* has only a single subscript.

A *subscript* is a number of indexes that follows an array's name.

This number can identify the number of individual elements in the array.

Arrays declaration

Declaration of array is as follows :

```
float expenses[12];
```

The array is named expenses, and it contains 12 elements.

Each of the 12 elements is the exact equivalent of a single float variable.

All of C's data types can be used for arrays .

C array elements are always numbered starting at 0, so the 12 elements of expenses are numbered 0 through 11.

Arrays

When an array is declared, the compiler sets aside a block of memory large enough to hold the entire array.

Individual array elements can be stored in sequential memory locations.

Individual elements of the array are accessed by using the array name followed by the element subscript enclosed in square brackets.

e.g.

```
expenses[1]=34.90;
```

```
expenses[10] = expenses[11];
```

Initializing Single Dimensional Array

```
int array[4] = { 100, 200, 300, 400 };
```

If the array size is omitted, the compiler creates an array just large enough to hold the initialization values.

```
int array[] = { 100, 200, 300, 400 };
```

It can be however, include too few initialization values, as in this example:

```
int array[10] = { 1, 2, 3 };
```

If an array element is not explicitly initialized, it is not sure that what value it holds when the program runs.

A program to read and display the elements

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

```
main()
```

```
{
```

```
int element[10],ctr;
```

```
printf("\n Reading the elements\n");
```

```
for(ctr=0;ctr<10;ctr++){
```

```
    printf("\n element %d is :\t",ctr+1);
```

```
    scanf("%d",&element[ctr]);
```

```
}
```

```
printf("\n Printing the elements\n");
```

```
for(ctr=0;ctr<10;ctr++)
```

```
    printf("%8d",element[ctr]);
```

```
}
```

Arrays as Strings

- String is an array of characters terminated by null character.
- There are two different ways to initialize the array.
- One is comma-delimited list of characters enclosed in braces,
- You can specify a string literal enclosed in double quotation marks.

Here are some examples:

```
char color[10];  
char color[10] = {'y', 'e', 'l', 'l', 'o', 'w', '\0'}; //Note NULL character has to be included at  
the end  
char color[10] = "orange";  
char color[] = {'g', 'r', 'a', 'y', '\0'};  
char color[] = "salmon";
```

```
/* reverse: reverse string s in place */
```

```
void reverse(char s[])
```

```
{
```

```
    int c, i, j;
```

```
    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
```

```
        c = s[i];
```

```
        s[i] = s[j];
```

```
        s[j] = c;
```

```
    }
```

```
    s[i] = '\0';
```

```
}
```

```
main()
```

```
{
```

```
    char name[]="TATA ELXSI";
```

```
    reverse(name);
```

```
    puts(name);
```

```
}
```

```
/* atoi: convert s to integer */  
int atoi(char s[])  
{  
    int i, n;  
    n = 0;  
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)  
        n = 10 * n + (s[i] - '0');  
    return n;  
}
```

```

/* itoa: convert n to characters in s */
void itoa(int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0) /* record sign */
        n = -n; /* make n positive */
    i = 0;
    do { /* generate digits in reverse order */
        s[i++] = n % 10 + '0'; /* get next digit */
    } while ((n /= 10) > 0); /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

```

Multidimensional Arrays

- Multidimensional arrays, or “arrays of arrays”.
- this can be done by adding an extra set of brackets and array lengths for every additional dimension you want your array to have.
- For example, here is a declaration for a two-dimensional array that holds five elements in each dimension.
- A two-element array consisting of five-element arrays

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

Adding matrix elements

```
#include<stdio.h>
main()
{
int m1[2][2],m2[2][2],m3[2][2];
int i,j,r,c;

printf("Enter matrix size\n");
scanf("%d%d",&r,&c);

printf("enter the elements into matrix m1\n");
for(i=0;i<c;i++)
    for(j=0;j<c;j++)
        scanf("%d",&m1[i][j]);
```

Adding matrix elements

```
printf("enter the elements into matrix m2\n");
for(i=0;i<r;i++)
    for(j=0;j<c;j++)
        scanf("%d",&m2[i][j]);

for(i=0;i<r;i++)
    for(j=0;j<c;j++)
        m3[i][j]=m1[i][j]+m2[i][j];

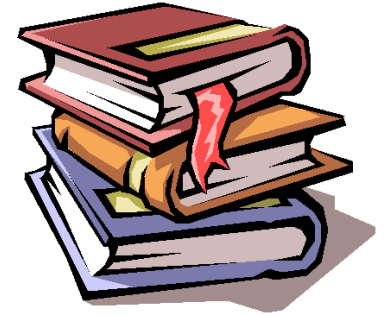
printf("Resultant matrix is ..... \n");
for(i=0;i<r;i++){
    for(j=0;j<c;j++)
        printf("%d  ",m3[i][j]);
}
}
```


Exercise

- Matrix multiplication
- Transpose of Matrix
- Trace and norm of matrix

references

- The GNU C Reference Manual :
<https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html>
- The GNU C Reference Manual :
<http://www.cprogramming.com/>
- C Programming Language
by Brian W. Kernighan , Dennis Ritchie
- C: The Complete Reference
by Herbert Schildt



Thank you



TATA ELXSI

ITPB Road Whitefield
Bangalore 560 048 India
Tel +91 80 2297 9123
Fax +91 80 2841 1474
e-mail info@tataelxsi.com

www.tataelxsi.com