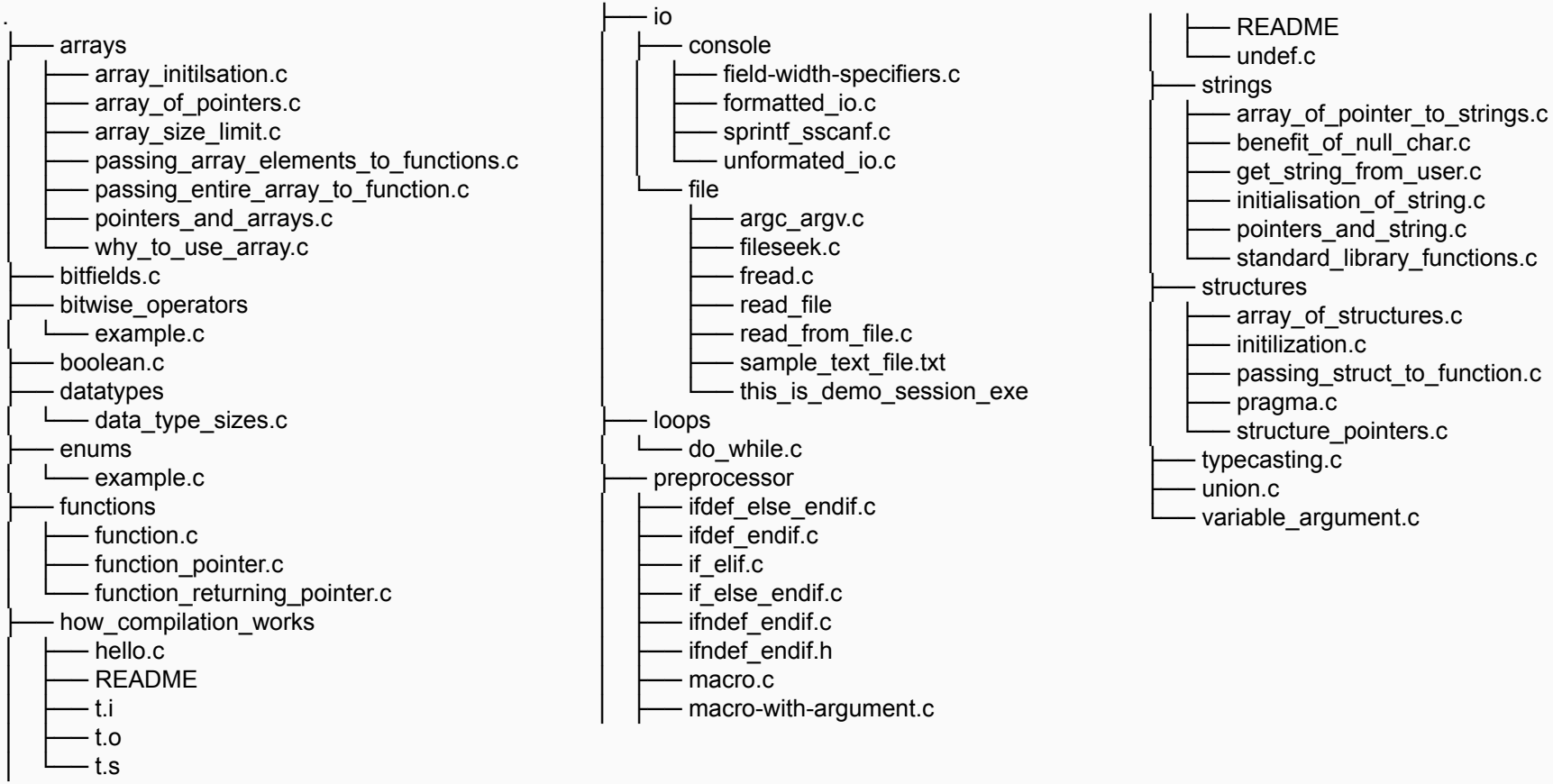
C Programming

Basics of Computing World !

lynxbee.com

Modular C Programs

lynxbee.com



What is C ?

- C was developed by Dennis Ritchie in 1972 at AT&T Bell Laboratories, USA.
- C is basic of all languages, to make fundamentals stronger it's advised to learn C first.
- C is base language for Embedded system.
- C is best language for performance (speed of execution), produces less code size hence good for embedded devices which have processor power & memory size restrictions.

Very Basic C program


```
$ vim minimum-c-program.c

main() {
}
```

Above is the very basic framework for Any C program.

- main, () and {}

```
$ gcc -o minimum-c-program minimum-c-program.c


$ ./minimum-c-program
```

```
#include <stdio.h>

main() {
        printf("Hello World\n");
}
```

- What happens when you didn't written #include

Constants, Variables, Keywords

Constants - Entity that do not change

Integer constants -
1, 423, +700, -345 etc

Float Constants -
426.9, -367.3 etc

Character constants -
‘A’ , ‘n’, ‘Z’, ‘3’ etc
Maximum length => 1 Char in single inverted

Variables - entity that may change

Names given to memory location which can change.

int age; float salary; char yes_no;

Keywords

- Words whose meaning is already decided for the compilers.
- Keywords can't be used as variable names.

auto	double	int	struct	break	else	long	switch
case	enum	register	typedef	char	extern	return	union
const	float	short	unsigned	continue	for	signed	void
default	goto	sizeof	volatile	do	if	static	while

First Program


```
#include <stdio.h>

void main() {
        printf("hello world");
}
```

- Header
- main
- Body

- #include is - preprocessor.

- Printf is declared in header stdio.h and defined in a library.

- Always ends with ;

Second Program


```
#include <stdio.h>

int main(void) {
        /* Printing Hello World */
        printf("hello world");
        return 0;
}
```

- Comments should be included in /* */
- Returns integer
- Void

Third Program


```
#include <stdio.h>

int main(void) {
        int p, n; /* declaration */
        float r, si;
        p = 1000; /* definition */
        n = 3;
        r = 8.5;

        si = p*n*r/100;
        printf ("Simple Interest is : %f \n", si);
}
```

- Printf => printf("<format string>" , <list of variables>);
- %f - float, %d - Int, %c - Char.
- \n -> New line

Compilation and Execution

- **Create a file using editor**
  vim simple_interest.c

- **Compile using gcc (GNU C Compiler for Linux)**
  gcc -o simple_interest simple_interest.c

- **./simple_interest**

Receiving Inputs from User using scanf


```
#include <stdio.h>

int main(void) {
        int p,n;
        float r,si;
        printf("Enter values of p, n, r");
        scanf("%d, %d, %f", &p, &n, &r);
        si = p*n*r/100;
        printf("Simple Interest = %f\n", si);
        return 0;
}
```

& - Address of operator, which is actually a memory location

Operators

Priority	Operators	Description
First	* / %	Multiplication, division, Modular
Second	+ -	Addition, Subtraction
Third	=	Assignment

Associativity of Operators


```
#include <stdio.h>

int main (void){

        /* Associativity of operators :1) Left to Right 2) Right to Left */
        /* Multiplication & Division has "Left to Right Associativity" */
        /* for Division: / , left side is 3 & right side is 2*5, indicating left side is unambiguous */
        /* for Multiplication: * , left side is 3/2 & right side is 5, indicating right side is unambiguous */
        /* Since both * & / has same priority & "Left to Right" Associativity, In below example */
        /* Division will get executed first and them multiplication */
        /* Hence result will be 5 */

        int a = 3/2*5;
        printf("a=%d\n", a);

        /*Above Statement will print result as 5 since 3/2 is equals to 1 due to integer division*/
        return 0;
}
```

Decision Making Statements – If, If-else, Switch

If Statement

Conditions - x == y, x != y, x < y, x > y, x <= y, x >= y

Simple Program

```
#include <stdio.h>

int main (void) {
        int x = 9;
        if ( x < 10 )
                printf(" you have assigned x less than 10\n");
        return 0;
}
```

If - else Statement


```
#include <stdio.h>

int main() {
        Int x = 11;
        If (x<10)
                printf("you have set x less than 10\n");
        else
                printf("you have set x greater than 10\n");
        return 0;
}
```

Nested If else statements are possible

Logical Operators

AND (&&), OR (||) & NOT (!)

Different than bitwise operators

Simple Example

Else - If Statement

If the first condition is satisfied, other conditions are not checked.
Reduces the multiple indentation required by “if-else” statements.

Simple Example -

The Not ! Operator

- This operator reverses the result of the expression it operates on.
- Final result will be either true or false i.e. 1 or 0
- !(y<10) => this means “not y less than 10”. In other words, if y is less than 10, the expression will be false, since (y<10) is true.

The Conditional Operator

? : also called as ternary operators.

Expression 1 ? expression 2 : expression 3

Loops

- While
- For
- Do-while

While Loop

```
#include <stdio.h>

int main() {
        int count = 1;
        while ( count < =5 ) {
                printf ( “count is %d”, count);
                count = count + 1;
        }
        printf(“now count is more than 5, exiting program\n”);
        return 0;
}
```

Increment and Decrement Operators

I++, i-- and ++i, --i

i+=1, i-=1

Example :

```
#include <stdio.h>

int main() {
        int count = 1;
        while ( count < =5 ) {
                printf ( “count is %d”, count);
                count++; // this is same as count+=1 and count = count + 1;
        }
        printf(“now count is more than 5, exiting program\n”);
        return 0;
}
```

The For Loop

- Most used
- All steps at one statement

```
for ( initialize; test; increment ) {
}
```

Example :

```
#include <stdio.h>

int main() {
int count;
for ( count = 1; count < =5; count++ ) {
   Printf ( “count is %d”, count);
}
printf(“now count is more than 5, exiting program\n”);
return 0;
}
```

Do – while loop

- Used when you are not sure how many times the loop needs to be executed.
- Loop is executed atleast once.

Example -

```
Int main () {
Char condition;
Int num=0;
Do {
        printf(“You are inside for the %d time\n”, num++);
        printf(“do you want to remain in loop ( y/n ) : ”);
        scanf(“%c”, &condition);
}while (condition == ‘y’)

printf(“you entered: %c, hence out of loop\n”, condition)
return 0;
}
```

Break & Continue

break - is used to move out of loop immediately upon meeting certain condition
continue - is used to move back to loop by skipping steps beyond this.

Example :

```
Int main() {
        Int i, j=5;
        for(i=1;i<=10;i++) {
                If ( i == j)
                        break;
                printf("i=%d\n",i);
        }
        Return 0;
}
```

Continue Example

```
Int main() {
        Int i, j=5;
        for(i=1;i<=10;i++) {
                If ( i == j)
                        continue;
                printf("i=%d\n",i);
        }
        Return 0;
}
```

Switch Case

- Used when we need to make a choice between number of things.
- Integer expression can be constants or something which evaluates to integer.
- “Case” will be followed by “int” or “char” constant.

```
Switch ( integer expression )
{
        Case constant 1 :
                “Do this”
        Case constant 2:
                “Do this”
        Default :
                “If no match”
}
```

```
#include <stdio.h>
int main() {
int i = 2
Switch ( i ) {
        case 1 :
                printf(“1”);
        case 2:
                printf(“2”);
        Case 3:
                printf(“3”);
        Default :
                “in default”
}
return 0;
}
```

```
#include <stdio.h>
Int main() {
Int i = 2
Switch ( i ) {
        case 1 :
                printf(“1”);
                break;
        case 2:
                printf(“2”);
                break;
        Case 3:
                printf(“3”);
                break;
        Default :
                “in default”
}
Return 0;
}
```

Goto Keyword : better avoid


```
Int main() {
        Int i; j=2;
        For (i=0; i<5; i++) {
                printf("i is %d\n", i);
                If (i==j)
                        goto at_end;
        }
        printf("normal completion of loop\n");
        return 0;

at_end: printf("reached directly at end due to i==j\n");
        Return 0;
}
```

```
Int main() {
        Int i; j=6;
        For (i=0; i<5; i++) {
                printf("i is %d\n", i);
                If (i==j)
                        goto at_end;
        }
        printf("normal completion of
loop\n");
        return 0;

at_end: printf("reached directly at end
due to i==j\n");
        return 0;
}
```

Functions

- Function is a block of statement which performs some task.
- Functions need three things, declaration, call and definition

```
#include <stdio.h>

/* declaration */
Void function_name();

int main () {
        /* function call */
        function_name();
        printf(“inside the main \n”);
        return 0;
}

/* function definition */
Void function_name() {
        printf(“inside the function \n”);
}
```

- Start main
- Control passes to function while main is suspended
- Function is executed
- Control returns to main
- Main resumes execution
- Main is completed

Types of Functions

- Library Function ex. main, printf
- User defined functions ex. myfunc()

Benefits of Functions -

- Avoid rewriting of same code
- Modular functions make program design easier for development and debugging

Passing Values to Functions - adds communication between calling & called function

```
Void sum(int);
Int main(void) {
        int i = 10;
        add(i);
        return 0;
}

void add (int j) {
        printf("sum = %d", j+5);
}
```

Type, order and number of arguments from calling and called function should be same.

Return Type & Variable from a Function


```
int sum(int);
int main(){
        Int i=10, ret;
        ret = sum(i);
        printf("sum = %d\n",sum);
        Return 0;
}

int sum(int j) {
        Int add;
        Add = j+5;
        Return add;
}
```

- Only "return" with no value will return garbage.
- "return" returns only one value.

Scope in Functions

- The scope of a variable is local to the function in which it is defined.
- Local variables defined inside a function are created in “stack” memory.

Calling Convention - C has “right to left” calling convention

```
#include <stdio.h>

Int main(){
        Int i = 1;
        printf(“%d, %d, %d \n”, i, ++i, i++);
        Return 0;
}
```

This program will return as 3, 3, 1

Function calls – 1. Call by Value 2. Call by reference

1. Call by Value
   - We pass “values” of variables to the called function.

```
#include <stdio.h>
Int main(void) {
        Int j=10;
        function(j);
        Return 0;
}

Void function (int k) {
        printf(“%d ”, k);
}
```

Call by reference - will need understanding of pointers

Pointers

Int i = 5;
This does 3 things,

- Reserve memory
- Give a Name
- Store Value in it

```
Int main() {
        Int i = 5;
        printf("address = %u \n", &i);
        printf("value = %d \n", i);
        Return 0;
}
```

& - is called "address of" operator

* - is called "value at address" operator

```
Int main () {
        Int i = 5;
        Printf ("address = %u \n", &i);
        Printf ("value = %d \n", i);
        Printf (" value = %d \n", *(&i) );
        Return 0;
}
```

Pointers

Address can also be collected in a variable.
Definition => j = &i ; declaration => int *j ;

This means, J is a variable which is used to store address of integer or value at the address contained in j is integer.

```
Int main() {
        Int i=5, *j;
        j = &i;
        printf("Address of i = %u", &i);
        printf("value of i = %d", i);
        printf("value inside j =%d", *j);
        return 0;
}
```

Back to Function Call by Reference

Example : swapping of two integers

```
Int main() {
        Int i=10, j=20;
        swap(i,j);
        printf("i= %d, j=%d", i, j);
        Return 0;
}

Void swap(int x, int y) {
        Int t;
        t = x;
        X = y;
        Y = t;
        printf("x = %d, y=%d", x,y);
}
```

We can't use return since it can return only one value.

```
Int main() {
        Int i=10, j=20;
        swap( &i, &j );
        printf("i= %d, j=%d", i, j);
        Return 0;
}

Void swap( int *x, int *y) {
        Int t;
        T = *x;
        *x = *y;
        *y = t
}
```

Above program will actually exchange i & j

- Useful to return more than one variables to main.

Recursion

Recursion : when statement in a function calls the same function

Example : Calculate Factorial 4! = 4*3*2*1

```
Int factorial (int);
Int main() {
        Int num, ret;
        printf("Enter the number = ");
        scanf("%d", &num);
        Ret = factorial ( num);
        printf("calculated factorial = %d", ret );
        Return 0;
}

Int factorial(int x) {
        Int f = 1, i;
        For (i=x; i>=1; i--)
                F = f*i;
Return f;
}
```

Using recursion

```
Int factorial( int x ) {
        Int f;
        If (x == 1)
                Return 1;
        Else
                F = x * factorial (x-1);
        Return f;
}
```

Data Types

- char, int, float
- Signed & unsigned
- long & short

char	Signed, unsigned	
int	Signed, unsigned	Long, short
float		Float, double, long double

Use of sizeof


```
#include <stdio.h>

int main(void) {
    char c;
    signed char c1;
    unsigned char c2;

    int a;
    short int a1;
    long int a2;
    signed int a3;
    unsigned int a4;
    short signed int a5;
    short unsigned int a6;
    long signed int a7;
    long unsigned int a8;

    float b;
    double b1;
    long double b2;

    printf ("sizeof (char) = %d \n", sizeof(c));
    printf ("sizeof (signed char) = %d \n", sizeof(c1));
    printf ("sizeof (unsigned char) = %d \n", sizeof(c2));
    printf ("sizeof (int) = %d \n", sizeof(a));
    printf ("sizeof (short int) = %d \n", sizeof(a1));
    printf ("sizeof (long int) = %d \n", sizeof(a2));
    printf ("sizeof (signed int) = %d \n", sizeof(a3));
    printf ("sizeof (unsigned int) = %d \n", sizeof(a4));
    printf ("sizeof (short signed int) = %d \n", sizeof(a5));
    printf ("sizeof (short unsigned int) = %d \n", sizeof(a6));
    printf ("sizeof (long signed int) = %d \n", sizeof(a7));
    printf ("sizeof (long unsigned int) = %d \n", sizeof(a8));

    printf ("sizeof (float) = %d \n", sizeof(b));
    printf ("sizeof (double) = %d \n", sizeof(b1));
    printf ("sizeof (long double) = %d \n", sizeof(b2));

    return 0;
}
```

```
$ ./a.out
sizeof (char) = 1
sizeof (signed char) = 1
sizeof (unsigned char) = 1
sizeof (int) = 4
sizeof (short int) = 2
sizeof (long int) = 4
sizeof (signed int) = 4
sizeof (unsigned int) = 4
sizeof (short signed int) = 2
sizeof (short unsigned int) = 2
sizeof (long signed int) = 4
sizeof (long unsigned int) = 4
sizeof (float) = 4
sizeof (double) = 8
sizeof (long double) = 12
```

Storage Classes in C

- Where the variable would be stored
- Decides the initial value of variable when it's not specifically initialised.
- Scope of variable
- Life of variable

	Storage	Default Value	Scope	Life
Automatic	Memory	Garbage	Local to block	Till last statement in block
Register	CPU Registers	Garbage	Local to Block	Till last statement in block
Static	Memory	Zero	Local to Block	Persists between different function calls
External	Memory	Zero	Global	Till Program is running

Storage Classes in C ...

Automatic Storage Class

```
#include <stdio.h>
void main() {
        auto int i, j;
        printf("i = %d, j=%d \n", i, j);
}
```

Register Storage Class

```
#include <stdio.h>
void main() {
        register int i;
        For (i=1; i<10; i++ )
                printf("i = %d \n", i);
}
```

Static Storage Class

```
#include <stdio.h>

Void counter();

Int main() {
        counter();
        counter();
        counter();
}

void counter() {
        Static int i=1;
        Printf ("counter is = %d\n", i);
        I++;
}
```

Extern Storage Class


```
#include <stdio.h>

Int i = 2;

Int main() {
        I = i + 3;
        Printf ( “ i is set to %d \n”, i);
        function();
        Printf (“ i is set to %d \n”, i);
}

function() {
        I = i + 10;
}
```

Another Example :

```
#include <stdio.h>

Int x = 5;

Void main () {
        Extern int y;
        Printf (“x = %d, y = %d \n”, x, y);
}

Int y = 7;
```

C Preprocessors

- C program before compilation gets through C preprocessor
- Preprocessing is first step of building executable.
- Preprocessor begins with #

Macro Expansion :

```
#include <stdio.h>

#define MACRO_NAME 23

int main(void) {
        int a = 10;
        a = a + MACRO_NAME;
        printf("Adding a macro to a : %d\n", a);
        return 0;
}
```

- During compilation preprocessor replaced every occurrence of “MACRO_NAME” with assigned value i.e. 23 in our case.
- You need to make only one change.
- Compiler generates faster and more compact code for constants than variables.
-

Macro with Arguments


```
#include <stdio.h>

#define SQUARE(x) (x*x)

int main(void) {
        printf("SQUARE(9) is : %d\n", SQUARE(9));
        printf("SQUARE(25) is : %d\n", SQUARE(25));
        return 0;
}
```

Why Macros, Not functions ?

- Preprocessor replaces the Macro with the actual code / initialisation of macro during compilation which generates more code size but fast in run time.
- Function call, passes the control from one the other, adding delay in execution hence slowing run time execution but reduces the size of program.

Macros to Include File.

Two ways -

- #include “filename.h”
- #include <filename.h>

Difference only in search path.

Conditional Compilation using Macros


```
#include <stdio.h>

// below program will not print anything if ENABLE is not defined.
// way to enable this code is, just define like below,

#define ENABLE

// to disable this, just comment above line

int main(void) {

#ifdef ENABLE
        printf("This code is enabled upon condition as above\n");
#endif
        return 0;
}
```

“Ifdef” can be used to make programs portable.

#ifdef - #else - #endif


```
#include <stdio.h>

// below program will print second message because ENABLE
// is not defined.

int main(void) {

#ifdef ENABLE
        printf("This code is enabled upon condition as above\n");
#else
        printf("When disabled, you want to execute this code\n");
#endif
        return 0;
}
```

#ifndef

#ifndef - Important during header file definitions to avoid multiple inclusion.

Header - ifndef_endif.h

```
#ifndef __IFNDEF_ENDIF_H
#define __IFNDEF_ENDIF_H

#define MYNAME "somename"

#endif
```

Program - ifndef_endif.c

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

int main(void) {
        printf("%s\n", MYNAME);
        return 0;
}
```

Header - ifndef_endif.h

```
#define MYNAME "somename"
```

Program - ifndef_endif.c

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

int main(void) {
        printf("%s\n", MYNAME);
        return 0;
}
```

Example of using #if - #else #endif


```
#include <stdio.h>

#define MACRO 4

int main(void) {
	int a = 5;

// Note: here you can't compare a macro with variable
// hence #if MACRO == a
// this statement will not work.

#if MACRO == 5
	printf("Macro value matches with a\n");
#else
	printf("Macro value doesnt match with a\n");
#endif

	return 0;
}
```

Example of #if – #elif – #endif


```
#include <stdio.h>

#define MACRO 3

int main(void) {

#if MACRO == 5
        printf("Macro is currently set to 5\n");
#elif MACRO == 4
        printf("Macro is currently set to 4\n");
#else
        printf("Macro is something else\n");
#endif

        return 0;
}
```

#undef Example


```
#include <stdio.h>

#define TEST

int main(void) {
        printf("starting program execution inside main\n");

#ifdef TEST
        printf("TEST is defined, so do something here\n");
#endif

#undef TEST

#ifdef TEST
        printf("TEST is defined, so do something here\n");
#else
        printf("TEST is undefined, did you do that ?\n");
#endif

        return 0;
}
```

Arrays - set of similar data types

Ordinary Variables are capable of holding only one value at a time.

```
#include <stdio.h>
Int main (void) {
        Int x;
        X = 4;
        X = 10;
        Printf (“x=%d\n”, x);
        Return 0;
}
```

Age = {21, 18, 27, 31, 45};

- In array counting of elements begins at 0

How to declare above array ?

```
Int age[5];
```

Using Arrays


```
#include <stdio.h>

int main(void) {
        int x[2];
        x[0] = 4;
        x[1] = 10;
        printf("x[0] = %d\n", x[0]);
        printf("x[1] = %d\n", x[1]);
        return 0;
}
```

- You can take values from user using scanf

```
for (i=0; i<2; i++) {
        printf(“%d”,&x[i]);
}
```

- Number inside square bracket is called array dimension e.g. in x[2] , 2 is array dimension
- Accessing elements from array

```
printf("x[0] = %d\n", x[0]);
```

You can use the for loop like below,

```
for (i=0; i<2; i++) {
        printf(“x[%d] = %d \n”, i, x[i]);
}
```

Initialisation of Arrays


```
Int x[2] = {4, 10};
Int x[] = {4, 10};
```

- Till the array elements are not given specific values, they contains garbage.
- If array is initialized where it is declared, mentioning of array dimension is optional.
- All array elements gets stored in contiguous memory.
- There will be no error message if you are going beyond array size like below,

```
#include <stdio.h>
Int main() {
        Int x[3], i;
        For (i=0; i<10; i++) {
                X [i] = i;
        }
        Return 0;
}
```

Array Boundary Checking


```
#include <stdio.h>

int main(void) {
        int x [5], i;
        printf("Enter elements of array: \n");
        for (i = 0; i < 10; i++)
                scanf("%d", &x[i]);

        printf("Printing output of above array :");
        for (i = 0; i < 10; i++)
                printf("x[%d] = %d \n", i, x[i]);

        return 0;
}
```

```
$ ./a.out
Enter elements of array:
1
3
14
34
23
56
99
2
7
89
Printing output of above array :x[0] = 1
x[1] = 3
x[2] = 14
x[3] = 34
x[4] = 23
x[5] = 56
x[6] = 99
x[7] = 2
x[8] = 7
x[9] = 89
*** stack smashing detected ***: ./a.out terminated
Aborted (core dumped)
```

Passing Array elements to function


```
#include <stdio.h>

void function (int y);

int main(void) {
        int i, x[2] = {4, 10};

        for (i=0; i<2; i++)
                function(x[i]);

        return 0;
}

void function (int y) {
        printf("%d\n", y);
}
```

```
#include <stdio.h>

void function (int *y);

int main(void) {
        int i, x[2] = {4, 10};

        for (i=0; i<2; i++)
                function(&x[i]);

        return 0;
}

void function (int *y) {
        printf("%d\n", *y);
}
```

Incrementing & Decrementing Pointers


```
#include <stdio.h>

int main(void) {
        int x[2] = {4, 10};
        int *y;
        y = &x[0];

        printf("x[0]=%d, *y=%d\n", x[0], *y);

        // now lets increment the pointer
        printf("Incrementing pointer as y++\n");
        y = y + 1; // here you can increment as many till y+(n-1) if "n" is element size
        printf("*y=%d\n", *y);
        // this printed 10, which means incrementing pointer reaches to next element.

        printf("Decrementing pointer as y--\n");
        y--;
        printf("*y=%d\n", *y);

        return 0;
}
```

Passing entire array to function


```
#include <stdio.h>

void function (int *, int);

int main(void) {
        int x[2] = {4, 10};
        function(&x[0], 2);
        // above call also can be made as,
        // function(x, 2);
        return 0;
}

void function (int *y, int j) {
        int i;
        for (i = 0; i < j; i++)
                printf("%d\n", *(y+i));
}
```

Array of pointer


```
#include <stdio.h>

int main(void) {
        int x[2] = {4, 10};
        int *p[] = {x, x+1};

        printf("%d, %d\n", **p, **(p+1));
        return 0;
}
```

Strings ...

- String is a one dimensional array of characters terminated by null (‘\0’)
  Char message = {‘H’, ‘E’, ‘L’, ‘L’, ‘O’, ‘\0’};
  Each character occupies one byte and also the last character ‘\0’ hence total size of string will be always one more than actual characters in string.
- Null character ‘\0’ is way to decide when the string ends.

Initialisation of Strings

```
#include <stdio.h>

int main(void) {
        char message[] = {'H', 'E', 'L', 'L', 'O', '\0'};
        printf("string is : %s\n", message);

        // Another way of initialising same string is as below,
        // char message[] = "HELLO";
        // note here, we dont need to mention "null" character '\0'
        // and this is most standard way to initialize strings

        return 0;
}
```

Another use of null character


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

int main(void) {
	char message[] = "How will you calculated the characters in this sentence ?";
	int length = 0;

	// so this is where null character helps us.
	while (message[length] != '\0') {
		length++;
	}

	printf("Number of characters in message = %d\n", length);

	// can we get this length instead of using while, yes, using strlen library function
	// but for that, we have to include string.h header
	// printf("Number of characters in message = %d\n", strlen(message));

	return 0;
}
```

Get string from Users ...


```
#include <stdio.h>

// if you want to run the program to test user input
// using scanf, just enable below line.
// #define TAKE_STRING_USING_SCANF

int main(void) {
        char name[20];
#ifdef TAKE_STRING_USING_SCANF
        printf("Enter Your Name: \n");
        scanf("%s", name);
        printf("You entered your name as: %s\n\n", name);

        printf("Did you want to enter your surname along with name? Try
Again...");
        scanf("%s", name);
        printf("You entered your name & surname is: %s\n\n", name);

        printf("Did you observed that, we couldn't print your surname which\n");
        printf("was separated by space i.e. ' ' \n");
        printf("So, thats the limitation of accepting string using scanf \n");
        printf("You can't accept the \"Multi Word\" string.\n\n");

        printf("Ok, got it.. So, whats the solution ?\n");
        printf("Thats where, fgets & puts will help us..\n");

#else
        printf("Now, Try to enter your name & surname\n");

        if(fgets (name, 20, stdin) != NULL) {
        /* writing content to stdout */
        puts("You entered your name and surname as\n");
        puts(name);
        }
#endif

        return 0;
}
```

Pointers and Strings


```
#include <stdio.h>

int main(void) {
        char *message = "hello";
        char *p;

        p = message;
        printf("string in p : %s\n", p);

        // Now we will try to update the string in p
        p = "world";
        printf("updated string in p : %s\n", p);
        // same you can't do with static string which can be defined as

        // char message[] = "hello";
        // char p[10];
        // p = message;
        // this will give error;
        // or p = "world" is not possible.

        // hence we need to use pointers;

        return 0;
}
```

- you can point address of one string to another pointer
- update the string inside pointer.

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

int main(void) {
        char source[] = "hello";
        char destination[50];
        int length;

        length = strlen(source);
        printf("length of source string = %d\n", length);

        strcpy(destination, source);
        printf("destination string after copying source is now: %s\n", destination);

        strcat(destination, " world!");
        printf("destination string after appending another string is now: %s\n", destination);

        if (!strcmp(source, "test string")) {
        printf("source and \"hello\" string is same\n");
        } else {
        printf("source is certainly not same as \"test string\"\n");
        }

        return 0;
}
```

Library functions for string

- Strlen, strcpy, strcat and strcmp are most widely used string library functions
- Check string.h for other supported functions.

Array of Pointer to Strings


```
#include <stdio.h>


int main(void) {
        char *names[5];
        int i;

        for (i=0; i<5; i++) {
                printf("Enter Name: ");
                scanf("%s", names[i]);
        }

        return 0;
}
```

This program will end up with run time error, since we are not initializing the memory locations of the strings in the array.

Solution ...


```
int main(void) {
        char *names[5];
        int i, len;
        char single_name[20];
        char *p;

        for (i=0; i<5; i++) {
        printf("Enter Name: ");
        scanf("%s", single_name);
        len = strlen(single_name);
        p = (char *)malloc(sizeof(char *) * len);
        strcpy(p, single_name);
        names [i] = p;
        }

        printf("\n Entered Names are as below: \n");
        for (i=0; i<5; i++) {
        printf("%s\n", names[i]);
        }

        return 0;
}
```

Here we allocate memory in runtime for the new string and save this address to the array of pointers.

New things -

- Malloc
- Typedef (char *) , actually malloc returns void pointer hence we need to typedef it to character.
- Malloc memory needs to freed by ourself after using.

Structures

Variables are single data type, int, float, char
Arrays are collection of single data types
Structures are collection of different data types.

For example - person has “name”, “age”, “salary” etc.

Declaration -

```
Struct person {
        Char name[10];
        Int age;
        Float salary;
};
```

```
#include <stdio.h>

int main(void) {
        struct person {
                char name[20];
                int age;
                float salary;
        };

        struct person p1 = {"person_name", 20, 20000.78};

        printf("name = %s, age = %d, salary =
                 %f\n", p1.name, p1.age, p1.salary);
        return 0;
}
```

Check for

- Struct Variable declaration
- Struct initialization
- Structure element access
- we can also initialise elements using scanf and &p1.name

We can also create the struct variable as,

```
Struct person {
        Char name[20];
        Int age;
        Float salary;
} p1;
```

Points to Note:

- Closing bracket of struct declaration must follow semicolon
- Struct declaration does not tell compiler to reserve any space in memory
- Normally structures are declared at the top of the program, even before main or mostly inside a separate header which can be included in the program.
- If a struct variable is initialized with {0} , then all its elements will be initialized to 0.

Array of Structures


```
struct person {
        char name[20];
        int age;
        float salary;
};

struct person p[5];
```

Passing Structure to Function


```
#include <stdio.h>

struct person {
        char name[20];
        int age;
        float salary;
};

void function(struct person);

int main(void) {

        struct person p1 = {"person_name", 20, 20000.723};

        function(p1);

        return 0;
}

void function(struct person p) {

        printf("name = %s, age = %d, salary = %f\n", p.name, p.age, p.salary);
//      printf("name = %s, age = %d, salary = %.3f\n", p.name, p.age, p.salary);
}
```

- We had to move structure declaration outside of main so that it will be available to function.
- Entire structure can be passed to function

Structure Pointers


```
#include <stdio.h>

int main(void) {
        struct person {
        char name[20];
        int age;
        float salary;
        };

        struct person p1 = {"person_name", 20, 20000.78};
        struct person *ptr;

        ptr = &p1;

        printf("name = %s, age = %d, salary = %f\n", ptr->name, ptr->age, ptr->salary);

        return 0;
}
```

Note :

- Structure Pointers use -> operator to access element

```
#include <stdio.h>

// To enable pragma, just uncomment below line.
// #pragma pack (1)

struct t {
        char c;
        int i;
        float f;
};

int main(void) {
        struct t t1;

        printf("Address of character: %u\n", &t1.c);
        printf("Address of int: %u\n", &t1.i);
        printf("Address of float: %u\n", &t1.f);

        return 0;
}
```

Check how this program works with commenting and uncommenting #pragma pack (1)

Console Input / Output

- Formatted => printf, scanf
- Unformatted => getch, getche, getchar, putch, putchar, gets, puts

printf(“format string”, list of variables);

Format string contains -
- Characters that are printed as they are
- Conversion specification that begins with %
- Escape sequence that begin with \

Example -> printf(“Age : %d \n”, n);

Here, Age is printed as is, %d is conversion specification and \n is Escape sequence.

%d , %u, %ld, %lu, %x, %o - Integer conversion specifiers
%f, %lf - Float conversion specifiers
%c - character and %s - string conversion specifiers

Field Width Specifiers

- Can be used for printf to properly format output or restrict the digits after decimal

```
#include <stdio.h>

int main(void) {
        float s = 2789.742;
        int age = 23;

        printf("s = %10d years\n", age); //print in 10 columns with right justified like: s =         23 years
        printf("s = %-10d years\n", age); //print in 10 columns with left justified like  s = 23      years

        // print default float
        printf("s = %f\n", s);
        // print only 2 digits after decimal
        printf("s = %.2f\n", s);

        return 0;
}
```

Escape Sequence

\n , \t, \' , \" => these are mostly used.

Sprintf & sscanf

```
#include <stdio.h>

struct t {
        int i;
        char c;
        float f;
};

int main(void) {
        char str[20];
        struct t t1;

sprintf(str, "%d %c %f", 10, 'c', 23.45);

        printf("string: %s\n", str);

        printf("reading back from string to
struct\n");
        sscanf(str, "%d %c %f", &t1.i, &t1.c, &t1.f);
        printf("%d, %c, %f \n", t1.i, t1.c, t1.f);

        return 0;
}
```

Unformatted I/O


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

int main(void) {
        int i = 0, j = 0;
        char name[20];
        char ch, temp[20];

        printf("\nEnter person's name: \n");

        while ((ch = getchar()) != '\n') {
        temp[j] = ch;
        j++;
        }
        temp[j] = '\0';

        strcpy(name, temp);

        printf("===== You Entered =====\n");
        printf("Name: %s ", name);
        return 0;
}
```

Scanf has a limitation of accepting strings with %s.

File Operations

- Creating new file
- Opening existing file
- Reading from a file
- Writing to a file
- Moving to specific location in file (seek)
- Closing a file

```
int main(void) {
        FILE *fp;
        char ch;

        fp = fopen(FILENAME, "r");

        while((ch = fgetc(fp)) != EOF) {
                printf("%c", ch);
        }

        return 0;
}
```

```
// verify with NULL

int main(void) {
        FILE *fp;
        char ch;

        fp = fopen(FILENAME, "r");
        if (fp == NULL) {
                printf("file %s is not present, please
check\n", FILENAME);
                return -1;
        }

        while((ch = fgetc(fp)) != EOF) {
                printf("%c", ch);
        }

        return 0;
}
```

File opening Modes

- R - read
- W - write
- A - append
- R+ - read, write, modify
- W+ - write, read back, modify
- A+ - read, append

fprintf, & fscanf => same as printf & scanf but operates on files.

fread & fwrite => reads and writes in one go.

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

size_t fwrite(const void *ptr, size_t size, size_t nmemb,
        FILE *stream);
```

- The function fread() reads nmemb items of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.

- The function fwrite() writes nmemb items of data, each size bytes long, to the stream pointed to by stream, obtaining them from the location given by ptr.

Read from file and save to string


```
#include <stdio.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

// This program reads a text file into character buffer
// and then prints as string.

#define FILENAME "sample_text_file.txt"

int file_length(char *f)
{
        struct stat st;
        stat(f, &st);
        return st.st_size;
}

int main(void) {
   FILE *fp;
   char *buf;
   int filelength, r;

   fp = fopen(FILENAME, "rb");
   if (fp == NULL) {
        printf("file %s is not present, please check\n", FILENAME);
        return -1;
   }

   filelength = file_length(FILENAME);
   printf("filelength = %d\n", filelength);
   buf = (char *) malloc(filelength * sizeof(char) + 1);
   if (buf == NULL) {
        printf("Can't allocate memory\n");
        return -1;
   }
   memset(buf, 0, filelength);

   r = fread(buf, 1, filelength, fp);
   printf("read %d characters into buffer\n", r);
   buf[filelength+1] = '\0';

   printf("%s", buf);
   return 0;
}
```

File Seek Operations


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

#define FILENAME "sample_text_file.txt"

int main (void) {
   FILE *fp;
   int pos;
   char buf[20];

   fp = fopen(FILENAME, "rw+");
   if (fp == NULL) {
            printf("unable to open file: %s\n", FILENAME);
            return -1;
   }

   pos = ftell(fp);
   printf("When file opened, pointer is at : %d\n", pos);

   printf("Now lets go 7 positions ahead, and print 15 characters\n");
   fseek(fp, 7, SEEK_CUR); //SEEK_CUR is current pointer position

   pos = ftell(fp);
   printf("We are reached now at pointer position : %d\n", pos);

            fread(buf, 15, 1, fp);
   buf[15] = '\0'; //terminate to make string

   printf("Buffer of length: %d contained : %s\n", strlen(buf), buf);

   printf("Now we were at : %d, lets go to end of file\n", ftell(fp));
   printf("By going to end of file, will also tell us length of file\n");

   fseek(fp, 0, SEEK_END);
   pos = ftell(fp);
   printf("We reached to end: position = filelength : %d\n", pos);

   printf("Now lets rewind ourself to start of file\n");
   rewind(fp);

   pos = ftell(fp);
   printf("We are now at position : %d\n", pos);

   return 0;
}
```

Argc & argv

- Using static string for filenames etc, requires program to compile every time
- Argv, argv allows to run same program with different inputs from command line without compiling program

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    // suppose we decided we want to take only
    // two arguments as input to the executable
    // then argc i.e. argument count will be 3
    // 2 for actual arguments and one for exe name

    if (argc != 3) {
            printf("incorrect number of command line arguments\n");
            return -1;
    }

    // now we will just print what will get copied
    // into argv which is array of character strings

    printf("argv[0] = %s\n", argv[0]);
    printf("argv[1] = %s\n", argv[1]);
    printf("argv[2] = %s\n", argv[2]);

    // so if we compile this program and run, then it will print as.

    // $ ./a.out file1 file2
    //   argv[0] = ./a.out
    //   argv[1] = file1
    //   argv[2] = file2

    // Now its upto you to decide how we want to use this arguments.
    return 0;
}
```

Bitwise Operators

~	One's Complement	~12
>>	Right Shift	12 >> 3
<<	Left Shift	12 << 4
&	Bitwise AND	12 & 123
|	Bitwise OR	12 | 123
^	Bitwise XOR	12 ^ 123

Example of Bitwise Operators


```
#include <stdio.h>

void printbits(int z) {
   int size = 8 *sizeof(int);
   int j;

   for (j = (size - 1); j >=0; j--) {
            (0x1 << j) & z ? printf("1") : printf("0");
   }
   printf("\n");
}

int main(int argc, char **argv) {
   int x = 12;
   int y = 123;

   printf("%d results to bits as: => ", x);
   printbits(x);

   printf("%d results to bits as: => ", y);
   printbits(y);

   printf("One's complement of %d is : %d => ",x, ~x);
   printbits(~x);

   printf("bitwise AND : %d & % d is %d => ", x, y, x&y);
   printbits(x&y);

   printf("bitwise OR : %d | % d is %d => ", x, y, x|y);
   printbits(x|y);

   printf("bitwise XOR : %d ^ % d is %d => ", x, y, x^y);
   printbits(x^y);

   printf("Right shift %d by %d results : %d => ", x, 4, x>>4);
   printbits(x>>4);

   printf("Left shift %d by %d results : %d => ", x, 4, x<<4);
   printbits(x<<4);

   return 0;
}
```

Enums

Enums gives a opportunity to define our own data type.

- Makes program more readable
- Good for multi-developer scenario
- Internally compilers treat the enums as integers starting with 0 as first element.
- We can also initialize first element something other than 0, then all next elements will be 1 more than previous
- We can also initialize all elements with different integer values.

Scope of enums can be global if declared outside main or local if declared inside function. This is major difference between macro and enum.

```
Enum engineering {
        Entc,
        Computer,
        IT,
        Mechanical
};

Enum engineering trade;
```

Enum Example


```
include <stdio.h>

enum engineering {
        entc,
        computer,
        it,
        mechanical
};

int main (int argc, char **argv) {
        enum engineering trade;
        int mytrade;

        printf("Enum initialized values to : entc = %d,
computer = %d, it = %d, \
 mechanical = %d \n", entc, computer, it, mechanical);

        // here just for simulation, we will take input in
integer from user
        printf("Enter your trade in number as seen: ");
        scanf("%d", &mytrade);

        trade = mytrade;

        if (mytrade == entc)
        printf("You are Electronics & Telecomm Engineer\n");
        else if (mytrade == computer)
        printf("You are Computer Engineer\n");
        else if (mytrade == it)
        printf("You are Information and Technology
Engineer\n");
        else if (mytrade == mechanical)
        printf("You are Mechanical Engineer\n");
        // Note: this can also be done using switch

        return 0;
}
```

Visit

lynxbee.com