Introduction To programming Using
C language
Lesson -1

output – input
functions

Lesson -1

C Program Structure

printf() function

Header files

Escape Sequences

Variable Definitions

format codes

field-width specifier

C Keywords – Reserved Words

scanf() function

identifiers

Function Definition

All C programs are divided into units called 'functions’.
Every C program consists of one or more functions; this program has only one.
No matter how many functions there are in a C program, **main**() is the one to which control is passed from the operating system when the program is run; it's the first Function executed.

built-in library functions:

output (display information from computer to user)
printf()
input (enter data from user to computer)
scanf()

C Program Structure

```c
#include <stdio.h>
void main(void)
{
        printf("Hello Youssef");
        printf("\n");
        printf("Good Bye !!!");

}
```

```
Hello Youssef
Good Bye !!!
Process returned 12 (0xC)   execution time : 1.710 s
Press any key to continue.
```

The word "void" preceding "main" specifies that the function main() will not return a value.
The second "void," in parentheses, specifies that the function takes no arguments.

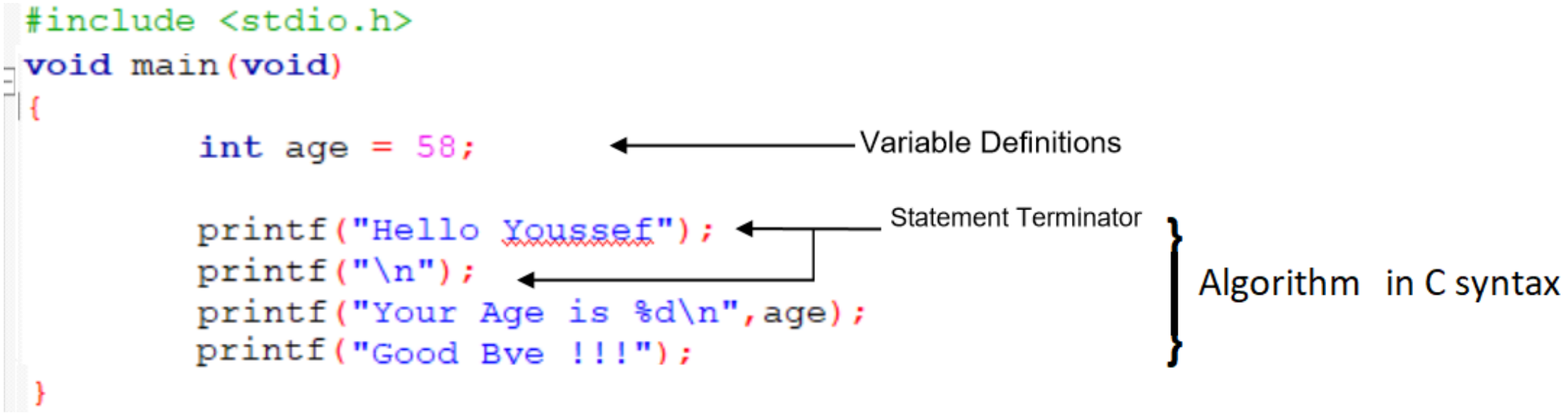
Variable Definitions

The statement

int num;

Defining a variable tells the compiler the type of the variable and the name of variable.
Specifying the name at the beginning of the program enables the compiler to recognize the variable as "approved" when you use it later in the program.

Complete program structure



The printf() function

The printf() function accepts a wide variety of ***format codes***, as shown in the next table

Code	Format
%c	Character
%d	Signed decimal integers
%e	Scientific notation (lowercase e)
%E	Scientific notation (uppercase E)
%f	Decimal Floating Point
%g	Use %E or %f which ever is shorter (if%e, uses lowercase e)
%G	Use %E or %f which ever is shorter (if%e, uses uppercase E)
%o	Unsigned Octal
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal (lowercase letters)
%p	Displays a Pointer
%%	Prints A % sign

Field-Width Specifiers

The printf() function gives the programmer considerable power to format the printed output.

```
void main(void)
{
     printf("%8.1f%8.1f%8.1f\n", 3.0, 12.5,523.3);
     printf("%8.1f%8.1f%8.1f\n", 300.0, 1200.5,5300.3);
}
```

The output will be lined up on the left side of the fields. like this:

```
   3.0     12.5    523.3
 300.0   1200.5   5300.3
```

A minus sign preceding the field-width specifier will put the output on the left side of the field instead of the right. For instance, let's insert minus signs in the field-width specifiers.

Escape Sequences

The new line character is an example of something called an '"escape sequence" so called because the backslash symbol (\) is considered an "escape" character. It causes an escape from the normal interpretation of a string, so that the next character is recognized as having a special meaning.

```
Void main(void)
{
     printf("Each \t word \t is \n Tabbed \t over \t
     once");
}
```

Here's the output:

```
Each        word        is
Tabbed      over        once
```

Escape Sequences

The tab “\t” and new line “\n” are probably the most often used escape sequences, but there are others as well. The following list shows the common escape sequences.

Control Character	Meaning
\n	New line
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\'	Single quote
\"	Double quote
\\	Backslash
\xdd	ASCII code in hexadecimal notation (each d represents a digit)
\ddd	ASCII code in octal notation (each d represents a digit)

Case 3: Discuss the output of the following program

```
void main(void)
{
    char c; /* 0..255 1 Byte     */
    int i; /* -32768..32767  2 Byte  */
    float f; /* 10^-38..10^38  4 Byte  */
    double d; /* 10"-308.. 1 0"308 8 Byte  */

    c='A';
    i=321;
    f=12.57;
    d=12.456789768;

    printf("\n\n\n");
    printf(" c = %c\n",c);
    printf(" i = %9d\n",i);
    printf(" i = %-9d\n",i);
    printf(" f = %f\n",f);
    printf(" d = %lf\n",d);

    printf("\n\n\n");
    printf(" c = %d\n",c);
    printf(" i = %c\n",i);
    printf(" f = %12.9f\n", f);
    printf(" d = %12.9lf\n",d);
}
```

```
 c = A
 i =       321
 i = 321
 f = 12.570000
 d = 12.456790



 c = 65
 i = A
 f = 12.569999695
 d = 12.456789768

Process returned 18 (0x12)   execution time : 1.299 s
Press any key to continue.
```

scanf() Function

The general purpose console input routine is scanf(). It can read all the built-in data types and automatically convert numbers into the proper internal format. It is much like the reverse of printf(). The prototype for scanf() is

int scanf(char *control_string,...);

The prototype for scanf() is in stodio.h the scanf() function returns the number of data items successfully assigned a value. If an error occurs, scanf() retuns EOF.

The Address Operator (&)

The scanf() function uses a new symbol, the ampersand (&) preceding the variable names used as arguments.

```
scanf("%f", &years);
scanf("%d %c %f", &event, &heat, &time);
```

What is its purpose? It would seem more reasonable to use the name of the variable without the ampersand, as we did in printf() statements in the pervious programs.
the C compiler requires the arguments to scanf() to be the addresses of variables, rather than the variables themselves.

scanf() - Format Specifiers

The input format specifier are preceded by a percent sign and tell scanf() what type of data is to be read next. These codes are listed in the next table

Code	Meaning
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating-point number
%f	Read a floating-point number
%o	Read an octal number
%s	Read a string
%x	Read a hexadecimal number
%p	Read a pointer
%n	Receives an integer value equal to the number of character read so far

C Keywords – Reserved Words

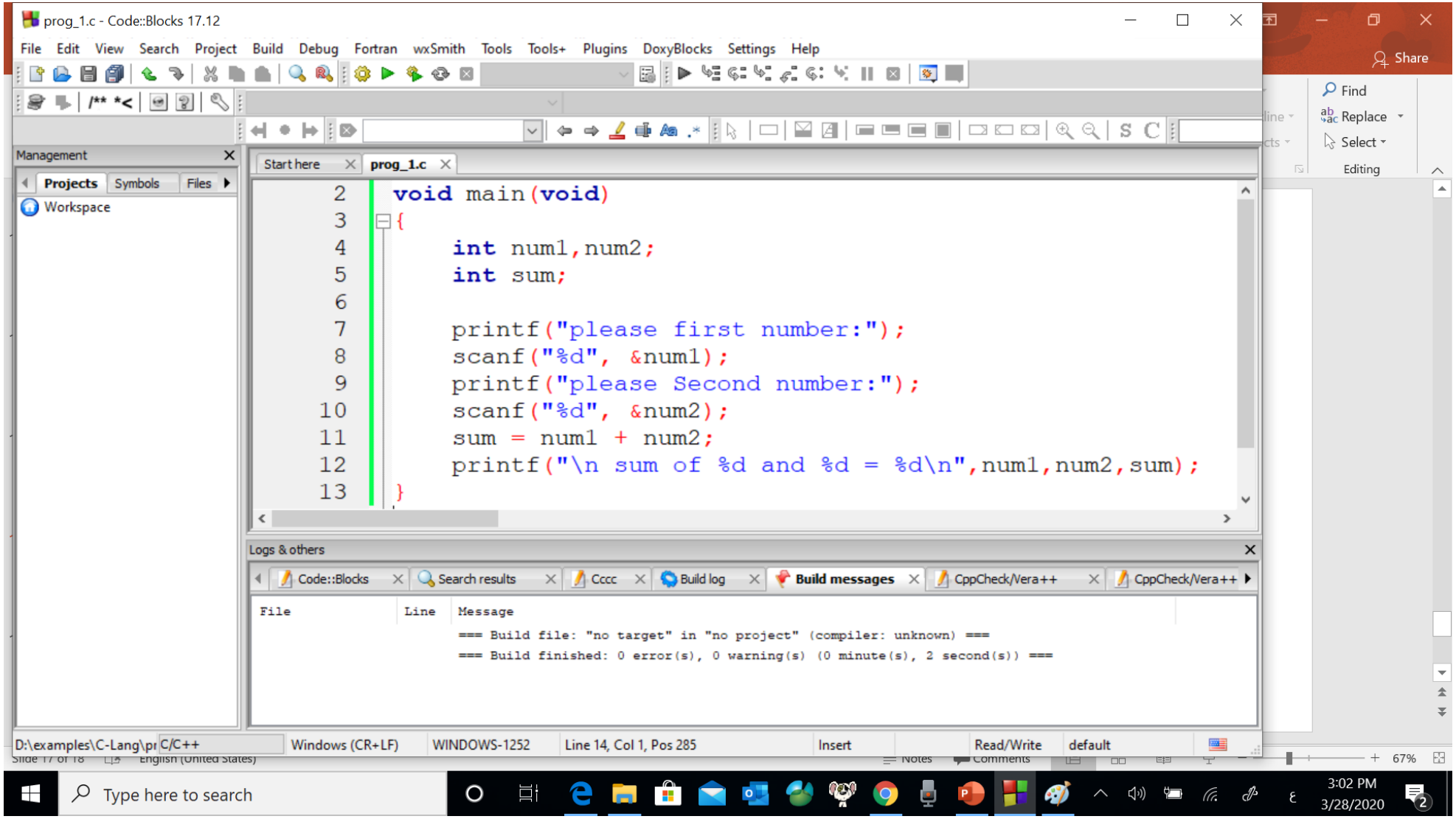
In C, we have 32 keywords, which have their predefined meaning and cannot be used as a variable name or as identifier in you program . These words are also known as “reserved words”. It is good practice to avoid using these keywords as identifiers.

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

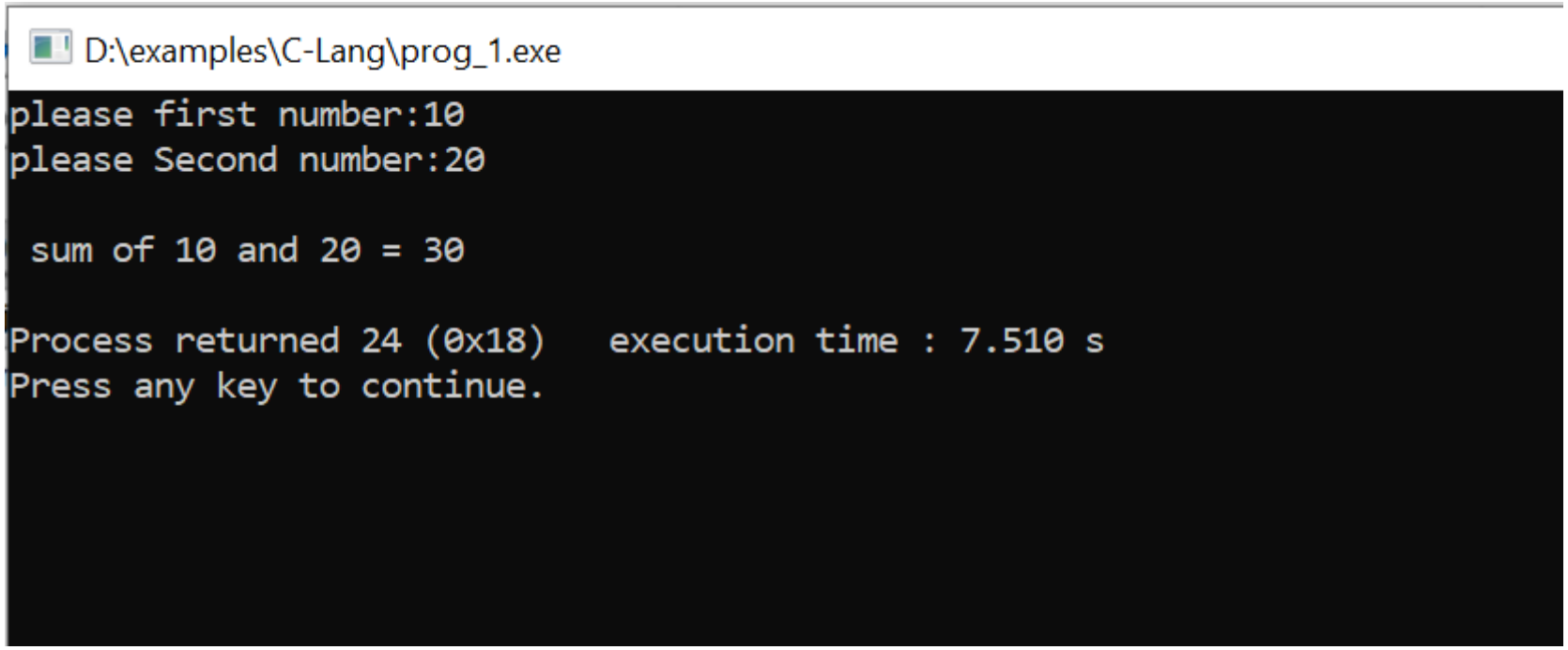
Case 1:
program to add two numbers

```
#include <stdio.h>
void main(void)
{
      int num1,num2;
      int sum;
      printf("please first number:");
      scanf("%d", &num1);
      printf("please Second number:");
      scanf("%d", &num2);
      sum = num1 + num2;
      printf("\n sum of %d and %d = %d\n",num1,num2,sum);
}
```

Case 1:program to add two numbers



Case 1:program to add two numbers (output)



identifiers

An **identifier** is a string of alphanumeric characters that begins with an alphabetic character or an underscore character that are used to represent various programming elements such as variables, functions, …etc. Actually, an **identifier** is a user-defined word.

Introduction To programming Using C language

Lesson -2

Variable Definitions

Lesson -2

identifiers

Variable Definitions

Variable types

Variable type modifier

C Basic Data Types

sizeof() Operator

identifiers

An **identifier** is a string of alphanumeric characters that begins with an alphabetic character or an underscore character that are used to represent various programming elements such as variables, functions, …etc. Actually, an **identifier** is a user-defined word.

Explicit – implicit declaration

An explicit declaration is when declare the variable explicitly before use it in any place in the program and some language restrict it to give it initialize value before use it.

An implicit declaration is when no need to declare the variable before use it so when you use it for the first type computer declare it implicitly depend on its assigned value

C language is explicit declaration language in most cases.

Variable Definitions

The statement

int num;

Defining a variable tells the compiler the type of the variable and the name of variable.
Specifying the name at the beginning of the program enables the compiler to recognize the variable as "approved" when you use it later in the program.

Variable types

We've been discussing variable definitions, which specify the name and type of a variable, and also set aside memory space for the variable.

Most variable types are numeric; but there is one that isn't, the character type. You have already met character constants. You know that they consist of a letter or other character surrounded by single quotes.

char ch1, ch2;
Ch1='A';

There are also several different kinds of numerical variables. We'll look at them briefly now. Later we'll learn more about them when we use them in actual programs.

You already know about integers (type int).
int num1;

Variable types

There are also two kinds of floating point variables.
Floating point numbers are used to represent values that are measured, like the length of a room (which might have a value of 145.25 inches) as opposed to integers.

Floating point variables can be very large or small. One floating point variable, type float, occupies four bytes and can hold numbers from about 10^38 to 10^-38 with between six and seven digits of precision.

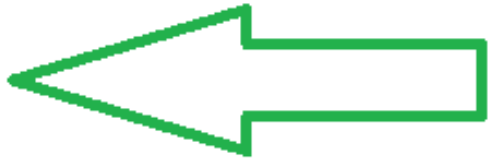
float fvar ;

A double-precision floating point variable, type double, occupies eight bytes and can hold number from about 10^308 to 10^-308 with about 15 digits of precision .

double dvar;

preferred place to Declare Variable

```c
#include <stdio.h>

void main(void)
{

    char ch;
    int number;
    unsigned short number_2;
    float salary;
    double disatance;


    ch='A';
    printf("char ch =%c\n",ch);
    ch=128;
    printf("char ch =%c   %d\n",ch,ch);

    number=2147483647;
    printf("int size =%d bytes \n",sizeof(number));
    printf("int number =%d\n",number);

}
```



Declear variable in the begining of main body (prefered place)

Variable type modifier

The C language provides the four basic arithmetic type specifiers *char*, *int*, *float* and *double*, and the modifiers ***signed***, ***unsigned***, ***short***, and ***long***.

type modifier	Data Types
signed	*char*
unsigned	*int*
short	*float*
long	*double*

Data representation

3 Bits

0	1	1	3
0	1	0	2
0	0	1	1
0	0	0	0
1	0	0	-4
1	0	1	-3
1	1	0	-2
1	1	1	-1

8 Bits

Range : -128..127 min: -2^7 max: $2^7 - 1$

Char ⇦ One Byte

	0	1	1	1	1	1	1	1	127
+	0	0	0	0	0	0	0	1	1
									128
=									⇩
	1	0	0	0	0	0	0	0	-128

C Basic Data Types

C Data Types

Type	Declaration Name	Range
Character	`char`	-128 to 127
Unsigned Character	`unsigned char`	0 to 255
Signed Character	`signed char`	-128 to 127
Integer	`int`	-32768 to 32767
Unsigned Integer	`unsigned int`	0 to 65535
Singed Integer	`signed int`	-32768 to 32767
Short Integer	`short int`	-32768 to 32767
Unsigned Short Integer	`unsigned short int`	0 to 65535
Signed Short Integer	`signed short int`	-32768 to 32767
Long Integer	`long` `long int`	-2147483648 to 2147483647
Signed Long Integer	`signed long int`	-2147483648 to 2147483647
Unsigned Long Integer	`unsigned long int`	0 to 4294967295
Floating-Point	`float`	-3.4E+38 to 3.4E+38
Double Precision Floating-Point	`double`	-1.7E+308 to 1.7E+308
Long Double Precision Floating-Point	`long double`	-1.2E+4932 to1.2E+4932

Some types gives you different ranges depend on the word size in your machine (16-bit,32-bit,64-bit) please check your machine ?

sizeof() Operator

Sizeof is a much used operator in the C. It is a compile time unary operator which can be used to compute the size of its operand. The result of sizeof is of unsigned integral type. sizeof can be applied to any data-type, including primitive types such as integer and floating-point types, pointer types, or compound datatypes.

Case2 : Program to determine data types size in byte using sizeof operator.

```
void main(void)
{
        printf("\nsize of char = %d\n", sizeof(char));
        printf("\nsize of int = %d\n", sizeof(int));
        printf("\nsize of unsigned int =    %d\n", sizeof(unsigned int));
        printf("\nsize of long int = %d\n",  sizeof(long int));
        printf("\nsize of unsigned long int =    %d\n", sizeof(unsigned long int));
        printf("\nsize of float = %d\n", sizeof(float));
        printf("\nsize of double = %d\n", sizeof(double));
        printf("\nsize of long double = %d\n",  sizeof(long double));
}
```

Find range of data types in C

		number of bytes	number of bits	min value	max value
	unsigned char	1	8	0	255
	char		7	-128	127
	unsined short int	2	16	0	65535
	short int		15	-32768	32767
	unsigned int	4	32	0	4294967295
long	int		31	-2147483648	2147483647

Find range of data types using C library

In C programming minimum and maximum constants are defined under two header files – limits.h and float.h.

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main()
{
    printf("Range of char %d to %d\n", SCHAR_MIN, SCHAR_MAX);
    printf("Range of unsigned char 0 to %d\n\n", UCHAR_MAX);
    printf("Range of signed short int %d to %d\n", SHRT_MIN, SHRT_MAX);
    printf("Range of unsigned short int 0 to %d\n\n", USHRT_MAX);
    printf("Range of signed int %d to %d\n", INT_MIN, INT_MAX);
    printf("Range of unsigned int 0 to %lu\n\n", UINT_MAX);
    printf("Range of signed long int %ld to %ld\n", LONG_MIN, LONG_MAX);
    printf("Range of unsigned long int 0 to %lu\n\n", ULONG_MAX);
    printf("Range of float %e to %e\n", FLT_MIN, FLT_MAX);
    printf("Range of double %e to %e\n", DBL_MIN, DBL_MAX);
    return 0;
}
```

Local – Global variables

Global variables are declared outside any function, and they can be accessed (used) on any function in the program.
Local variables are declared inside a function, and can be used only inside that function.

```c
#include <stdio.h>

int number_of_employee;

void main(void)
{

    int number;
    float salary;

    number=5;
    printf("number =%d\n",number);
    salary=7000.67;
    printf("salary =%9.2f\n",salary);
    number_of_employee=100;
    printf("number of employees =%d\n",number_of_employee);
}
```

Global variable to any place in program

default value for numeric zero

Local variables to the function main

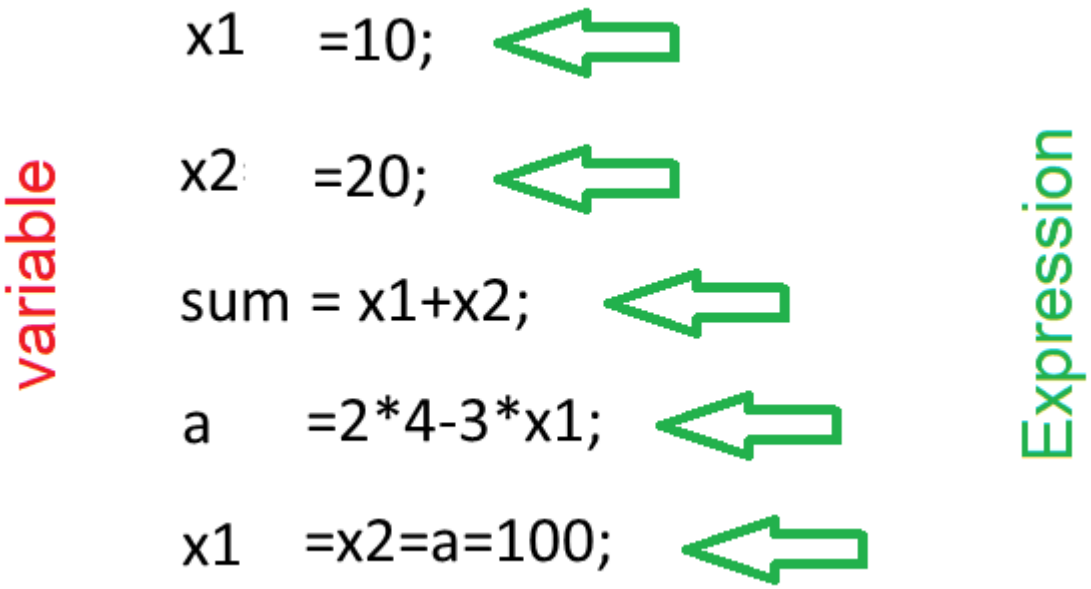
no default value (garbage value)

Introduction To programming Using C language Lesson -3

assignment operator-literals- comments

Assignment Operator =

Assignment operators are used to assigning value to a variable.



Assignment Operator

=

Assignment operators are used to assigning value to a variable.
The left side operand of the assignment operator is a variable and right side operand of the assignment operator is a value or expression .
The value on the right side must be of the same data-type or compatible of the variable on the left side otherwise the compiler will raise an error.

```
int a ;
int x1,x2;
int sum;

a= 10;
x1=10;   x2=20;
sum = x1+x2;

printf("Value of a is %d\n", a);
printf(“Sum = %d\n", sum );
```

Initializing Variables

It's possible to combine a variable definition with an assignment operator so that a variable is given a value at the same time it's defined.

```
void main(void)
 {
        int event = 5;
        char heat = 'c’;
        float time = 271.25;

        printf("The winning time in heat %c ", heat);
        printf(" of event %d was %f. ", event, time);

}
```

Literals in C

The constant values assigned to each constant variables or variable are referred to as the **literals**. Generally, both terms, constants and literals are used interchangeably. For eg,

int max= 5;

the value 5 is referred to as constant integer literal. There are four types of literals in C

Integer Literals

Integer literals are expressed in two types:

Prefixes
Suffixes

Prefixes Integer Literals

The Prefix of the integer literal indicates the base in which it is to be read.

Decimal-literal(base 10): A non-zero decimal digit followed by zero or more decimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9).
Octal-literal(base 8): a 0 followed by zero or more octal digits(0, 1, 2, 3, 4, 5, 6, 7).
Hex-literal(base 16): 0x or 0X followed by one or more hexadecimal digits(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C, d, D, e, E, f, F).
Binary-literal(base 2): 0b or 0B followed by one or more binary digits(0, 1).

Examples:

```
Int X;

X=56;
X=056;
X=0x46A;
X=ob10101;
```

Suffixes Integer Literals

The **Suffixes** of the integer literal indicates the type in which it is to be read.

These are represented in many ways according to their data types.

int: No suffix is required because integer constant is by default assigned as an int data type.

unsigned int: character u or U at the end of an integer constant.

long int: character l or L at the end of an integer constant.

unsigned long int: character ul or UL at the end of an integer constant.

long long int: character ll or LL at the end of an integer constant.

unsigned long long int: character ull or ULL at the end of integer constant.

Examples

```
int i=12l;
```

Defining Constants

There are two simple ways in C to define constants :
Using **#define** preprocessor.
Using **const** keyword.

```
#include <stdio.h>

int main() {
   const int  LENGTH = 10;
   const int  WIDTH = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}

output:
value of area : 50
```

```
#include <stdio.h>

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'
int main()
{
   int area;

   area = LENGTH * WIDTH;
   printf("value of area : %d", area);
   printf("%c", NEWLINE);
   return 0;
}

output:
value of area : 50
```

Renaming Data Types with typedef

Its use in clearer programs because the name of a type can be shortened and made more meaningful.

For example, consider the following statement in which the type unsigned char is redefined to be of type BYTE:

typedef unsigned char BYTE;

Now we can declare variables of type unsigned char by writing

BYTE var1, var2; instead of unsigned char var1, var2;

Comments

It's helpful to be able to put comments into the source code file that can be read by humans but are invisible to the compiler.

A comment begins with the two-character symbol slash-asterisk (/*) and ends with an asterisk-slash (*/).

It is usually illegal to nest comments. That is, you can't say

/* Outer comment /* inner comment * / more outer comment * /

Comments

```
#include <stdio.h>
 /* calculates age in days */

void main(void)
{
        /* initialize variables */
        float years, days;
        /*print prompt */
        printf("Please type age in years: ");

        /*get age in years from user */
        scanf("%f", &years);
        /* calculate age in days */
        days = years * 365;
        /* print result */
        printf(" You are %.1f days old.", days);
}
```

Introduction To programming Using
C language
Lesson -4

Arithmetic Operators- Expressions

Operators

Operators are words or symbols that cause a program to do something to variables or literals. For instance, the arithmetic operators (+) and (-) cause a program to add or subtract two numbers. There are many different kinds of operators;

Arithmetic Operators

C uses the four arithmetic operators that are common in most programming languages, and one, the remainder operator, which is not so common.

C math operators are symbols used for addition, subtraction, multiplication, and division, as well as other similar operations.

Symbol	Meaning
*	Multiplication
/	Division and integer division
%	Modulus or remainder
+	Addition
-	Subtraction

Formula	Result
4*2	8
64/4	16
80 - 15	65
12 + 9	21

Division and Modulus

The forward slash (/) always divides. However, it produces an integer division if integer values (constants, variables, or a combination of both) appear on both sides of the /. If there is a remainder, C discards it.

The percent sign (%) produces a modulus, or a remainder, of an integer division. This operator requires integers on both sides and will not work otherwise.

The Order of Precedence

You also must understand the order of precedence. This order (sometimes called the math hierarchy or order of operators) determines exactly how C computes formulas. The precedence of operators is the same as that used in high school algebra courses

try to determine the result of the following simple calculation:

2+3 *2

This is exactly the answer “8” that C computes!

The order of precedence for the primary operators:

Order	Operators
First	Multiplication, division, modulus remainder (*, /, %)
Second	Addition, subtraction (+, -)

Parentheses ()

Using Parentheses

If you want to override the order of precedence, put parentheses in your calculations. In other words any calculation in parentheses - whether it is addition, subtraction, division, or sometimes else- is always performed before the rest of the line.

Arithmetic expression

An arithmetic expression is a syntactically correct combination of numbers, operators, parenthesis, and variables.
(10 + 2) * (4 - 2) ➔ ?

Introduction To programming Using C language

Lesson -5

Assignment operator

Assignment operator

In C the assignment operator, =, is used more extensively than in other languages. So far, you have seen this operator used for the simple assignment of values to variables.

In C, the assignment , Multiple Assignments and compound assignment. Next discussion illustrates these additional uses.

```
x=14;
y=3+4*4;
a=2+3*x+y;
```

Multiple Assignments

If more than equal sign appears in an expression, each = performs an assignment. This multiple use introduces a new aspect of the precedence order. Consider the following expression:

```
a = b= c = d = o = 100;
```

Compound Assignment

If you compare a C program with a program with a similar purpose written in another language, you may well find that the C source file is shorter. One reason for this is that C has several operators that can compress often used programming statements. Consider the following statement:

total = total + number;
Here the value in number is added to the value in total, and the result is placed in total. In C this statement can be rewritten:

total += number;

So :

total = total + number; ➔ total += number;

Compound Assignment

Operator	Meaning	Example	Equivalent
+=	Addition assignment operator	bonus +=500;	bonus=bonus+500
- =	Subtraction assignment operator	budget -=50;	budget=budget-50
*=	Multiplication assignment operator	salary *=1.2;	salary=salary*1.2
/=	Division assignment operator	factor /=.50;	factor=factor/.50;
%=	Remainder assignment operator	daynum %=7;	daynum=daynum%7;

Introduction To programming Using
C language
Lesson -6

Increment and Decrement Operators

Increment Operator ++

C uses another operator that is not common in other languages, the increment operator.

The effect of num (++) is exactly the same as that of the statement

num = num + 1;

However, “num++” is far more compact to write. It also compiles into more efficient code.

Decrement Operator --

The effect of (num--) is exactly the same as that of the statement

num = num - 1;

Example:

```
int num1=3,num2=4,res1,res2;

res1 = num1 ++;
res2= --num2;
```

Result

res1←3 , num1←4
res2←3, num2←3

Introduction To programming Using C language

Lesson -7

Relational Operators

Relational operators

Relational operators are the vocabulary the program uses to ask questions about variables. Let's look at an example of a relational operator, in this case the "less than" (<) operator.

```
void main( void)
{
     int age;
     age = 15;
     printf("Is age less than 21? %d\n", age < 21 );
     age = 30;
     printf("Is age less than 21? %d\n", age < 21 );
}
```

The output from this program looks like this:

```
Is age less than 21? 1
Is age less than 21? 0
```

Relational operators

The relational operators in C look much like those in other languages. There are six of them:

Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

The equality operators

==	equal to
!=	Not equal to

Relational operators

Here's an example using the equal to (==) operator (sometimes called the "equal-equal" operator).

```
void main(void)
{
      int speed;
      speed = 75;
      printf("Is speed equal to 55? %d\n", speed == 55);
      speed = 55;
      printf("Is speed equal to 55? %d\n", speed == 55);
}
```

Here's the output of the equalto.c program:

```
Is speed equal to 55? 0
Is speed equal to 55? 1
```

Relational operators (examples)

Evaluates to either a false (0) or a true (1) value.

Assume that a program initializes four variables in this way:

```
int a = 5;
int b = 10;
int c = 15;
int d = 5;
```

The following statements are then true:

a is equal to d	so	a ==d.
b is less than c	so	b < c.
c is greater than a	so	c > a.
b is greater than or equal to a	so	b >=a.
d is less than or equal to b	so	d <=b.
b is not equal to c	so	b !=c.

Introduction To programming Using
C language
Lesson -8

Logical Operators

Logical Operators

There may be times when you need to test more than one set of variables. You can combine more than one relational test into a compound relational test by using C's logical operators shown below

Operator	Meaning
&&	AND
||	OR
!	NOT

Logical Operators

The && AND Truth table

True	AND	True	= True
True	AND	False	= False
False	AND	True	= False
False	AND	false	= False

The || OR Truth table,

True	OR	True	= True
True	OR	False	= True
False	OR	True	= True
False	OR	False	= False

The ! NOT Truth table.

NOT	True	= false
NOT	False	= True

Compound condition

Mark=77;

Res=(mark>60) && (mark<=80)
Res --------→1 (true)

Res = (mark<=50) || (mark>=85)
Res---------→0 (false)

Introduction To programming Using
C language
Lesson -9

The (?:) Conditional operator

Conditional operator

The conditional operator “?:” is C's only ternary operator.

A ternary operator requires three operands (instead of the single and double operands of unary and binary operators).
The conditional operator is used to replace if - else logic in some situations.
The format of the conditional operator is:

conditional_expression? expression1: expression2;

Conditional operator

Where conditional_expression is any expression in C that results in a True (nonzero) or False (zero) answer.

If the result of the conditional_expression is True expression1 executes.
If the result of the conditional_expression is False expression2 executes.

For Example if

```
x= 120;
y= 30;
z= ( x > y) ? 100 : 200;
```

Then

z←100

Introduction To programming Using
C language
Lesson -10

Bitwise Operators

Bitwise Operators

They operate on internal representations of data, not just "Values in variables" as the other operators do.

Bitwise operators require an understanding of binary numbering system and of your PC's memory.

enable you to operate at a level deeper than many programming languages allow.

Bitwise Operators

There are four bitwise logical operators shown in the next table. Because these operators work on the binary representation of integer data, systems programmers can manipulate internal bits in memory and variables.

The bitwise logical operators

Operator	Meaning
&	Bitwise AND
|	Bitwise inclusive OR
^	Bitwise exclusive OR
~	Bitwise 1's complement or Bitwise NOT

Bitwise Operators

The bitwise & (and) truth table

First bit	AND	Second bit	Result
1	&	1	1
1	&	0	0
0	&	1	0
0	&	0	0

The Bitwise | (OR) truth table

First bit	OR	Second bit	Result
1	|	1	1
1	|	0	1
0	|	1	1
0	|	0	0

The Bitwise ^ (exclusive OR) operator

First bit	XOR	Second bit	Result
1	^	1	0
1	^	0	1
0	^	1	1
0	^	0	0

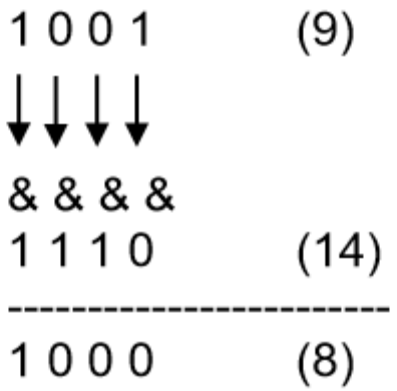
The Bitwise ~ (1's complement) operator

1's complement	bit	Result
~	1	0
~	0	1

Bitwise Operators

Examples:

If you apply the bitwise & operator to the numbers 9 and 14, you get a result of 8.The next figure show why this is so. When the binary values of 9(1001) and 14(1110) are operated on with a bitwise &, the resulting bit pattern is 8(1000)



Compound Bitwise Operators.

As with most of the mathematical operators, you can combine the bitwise operators with the equal sign (=) to form compound bitwise operators. The next table describes the compound bitwise operators.

The compound bitwise operators

Operator	Description
&=	Compound bitwise AND assignment
|=	Compound bitwise inclusive OR assignment
~=	Compound bitwise exclusive OR assignment.

The Bitwise Shift Operators

The Bitwise Shift Operators

Operator	Description
<<	bitwise left shift
>>	bitwise right shift

Example:

```
int num1 = 8;
int res;

res = num1 >> 1;

res ← 4;
```

Introduction To programming Using C language

Lesson -11

operator precedence- Associativity

Operator precedence

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

Operator associativity

If multiple operators with the same precedence are used in an expression then associativity rule tells the compiler which one has to execute first and which one has to execute next that is from left to right or right to left.

Operator precedence- associativity

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
2	++ --	Prefix increment and decrement	Right-to-left
2	+ -	Unary plus and minus	Right-to-left
2	! ~	Logical NOT and bitwise NOT	Right-to-left
2	sizeof	Size-of	Right-to-left
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+,-	Addition and subtraction	Left-to-right
5	<< >>	Bitwise left shift and right shift	Left-to-right
6	< ,<=, >, >=	For relational operators	Left-to-right
7	== !=	For relational = and ≠	Left-to-right
8	&	Bitwise AND	Left-to-right
9	^	Bitwise XOR (exclusive or)	Left-to-right
10	|	Bitwise OR (inclusive or)	Left-to-right
11	&&	Logical AND	Left-to-right
12	||	Logical OR	Left-to-right
13	?:	Ternary conditional	Right-to-left
16	=	Assignment	Right-to-left

Introduction To programming Using C language

Lesson -12

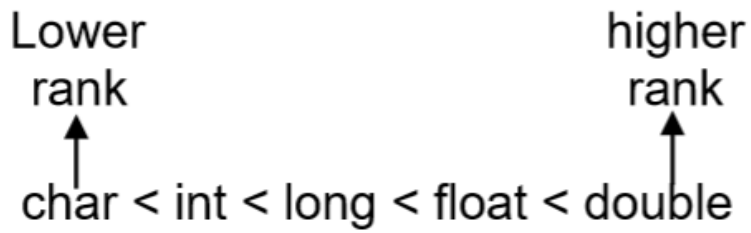
Type Conversion and Casting

Type Conversion

When data types are mixed in an expression, the compiler converts the variables to compatible types before carrying out the intended operation.

In these conversions, variables of lower rank are converted to the rank of the higher-ranking operand.

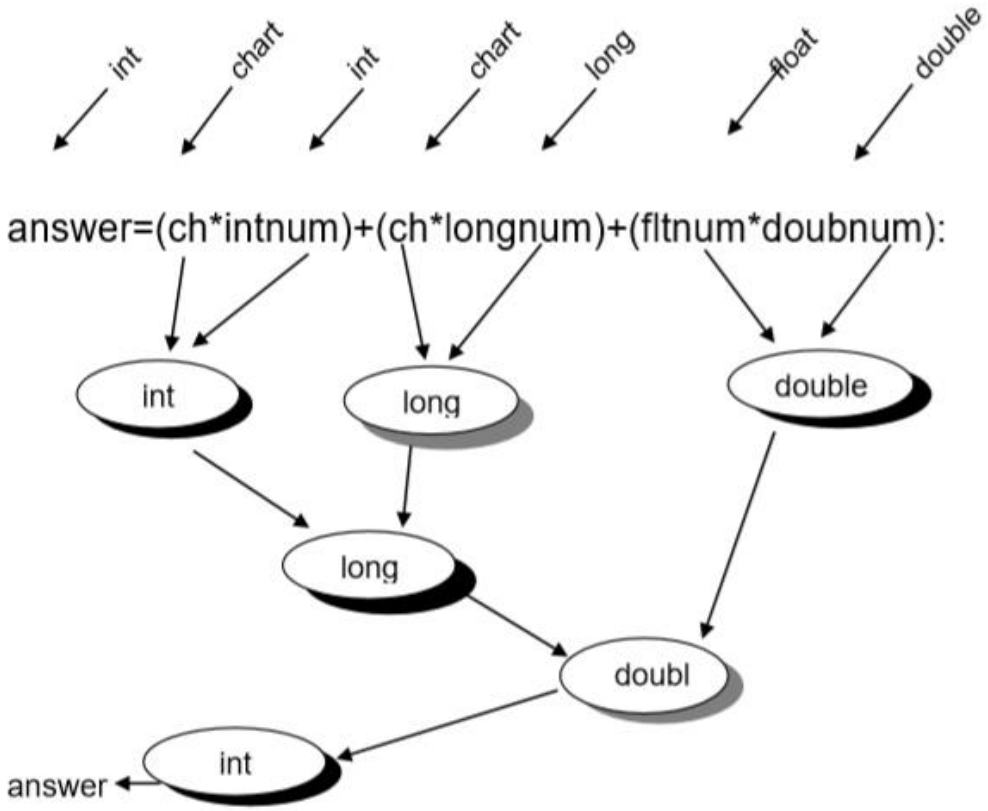
The ranking corresponds roughly to how many bytes each type occupies in memory.



Type Conversion - example

Before each operator is applied to the appropriate pair of variables, the variable with the lower rank is converted to the rank of the higher-ranking variable. This process is shown in the next figure.

```
void main(void)
{

    char ch;
    int intnum;
    long longnum;
    float fltnum;
    double doubnum;

    int answer;

    answer = (ch * intnum) + (ch * longnum) *  (fltnum * doubnum);

}
```



Some data types like *char* , *short int* take less number of bytes than *int*, these data types are automatically promoted to *int* or *unsigned int* when an operation is performed on them.

Promotion - Demotion

Promotion or moving from a lower rank to a higher rank, usually doesn't cause problems. Demotion, however, can easily result in a loss of precision or even yield a completely incorrect result. Next table details what happens when demotion occurs.

Data Type Demotion

Demotion	Result
int to char	low-order byte is used
long to char	low-order byte is used
float to char	float converted to long and low-order byte used
Double to char	double converted to float, float to long, and low-order byte used
long to int	low-order word (two bytes) used
float to int	float converted to long and low-order word used
Double to int	double converted to float, float to long, and low-order word used
float to long	truncate at decimal point; if result is too large for long, it is undefined
Double to long	truncate at decimal point; if result is too large for long, it is undefined

Promotion - Demotion

Generally, if a number is too large to fit into the type to which it is being demoted, its value will be corrupted.

Some loss of precision can even occur in the promotion of long to float, since these two types both use four bytes.

The moral is, avoid type conversions unless there's a good reason for them, and be especially careful when demoting a variable, since you may lose data.

Example

Case 1: example for type conversion

```
/* Arithmetic Operators */
/* an example for expression type conversion in mixed
data type * /
#include <stdio.h>
void main()
{
     int i1 =3 ,i2=2;
     float f,f1;

     f= i1/i2;
     printf("\n\n%12.5f\n",f); /* 1.00000 */

     f1= i2;
     f= i1/f1; /* expression type conversion */
     printf("\n\n%12.5f\n",f);/*1.50000*/
}
```

Typecasting

Typecasting provides a way to force a variable to be of a particular type. This overrides the normal type conversions we just described

```
answer = (int) fltnum;
```

the typecast will cause the value of fltnum to be converted to an integer value before being assigned to answer, no matter what type fltnum is.

Typecasting is often used to ensure that a value passed to a function is of the type the function is expecting. For instance, suppose we have a variable intnum of type int and we want to take its square root. However, the square root function requires a variable of type long. We could use the expression

```
sqroot((long) intnum)
```

Example

case 2: example for type casting

```
/* Arithmetic Operators */
/* an example for assignment type conversion in mixed
data type * /
/* and type casting * /
#include <stdio.h>
void main()
{
     int i=300;
     int ch;
     char c;

     c=i;
     printf("\n\n%d\n",c );

     c= (char)i; /*no need for type casting since there*/
                 /*is assignment type conversion       */
     printf("\n\n%d\n" ,c);

     ch = (char)i; /* you must use type casting since  */
                   /* there is not assignment type     */
                   /* conversion                       */
     printf("\n\n%d\n" ,ch);
}
```

Introduction To programming Using
C language
Lesson -13

Branching

Branching

Computer languages, must be able to perform different sets of actions depending on the circumstances.

C has three major decision-making structures:
the if statement, the if-else statement, and the switch statement.
A fourth, somewhat less important structure, is the conditional operator.

If Statement

C uses the keyword **if** to introduce the basic decision making statement. Here's a simple example:

```
if (expression)
    statement;
```

```
void main(void)
{
        char ch;

        ch = getche();
        if(ch == 'y')
        printf("\nYou typed y. ");
}
```

```
int main()
{
    int x = 20;
    int y = 22;
    if (x<y)
      printf("Variable x is less than y");
    return 0;
}
```

Multiple Statements With if

The body of the if statement may consist of either a single statement terminated by a semicolon or by a number of statements enclosed in braces. Here's an example of such a compound statement:

```
if (expression)
  {
    statement;
    statement;
    statement;
     ....
     ....
  }
```

```
void main(void)
{
        char ch;
        ch = getche();
        if(ch == 'y')
                {
                        printf("\nYou typed y. ");
                        printf("\nNot some other letter. ");
                }
}
```

```
int main()
{
   int n;
   printf("Enter a number:");
   scanf("%d",&n);
   if(n<10)
   {
      printf("%d is less than 10\n",n);
      printf("Square = %d\n",n*n);
   }
   return 0;
}
```

if-else Statement

If the expression is true, statement1 is executed; if not, statement2 is executed. Each statement can be a single statement or several in braces.

```
if (expression)
    statement1
else
    statement2
```

example:

```
void main(void)
{
        char ch;

        ch = getche();
        if(ch == 'y')
                printf("\nYou typed y. ");
        else
                printf("\nYou didn't type y. ");
}
```

Example 2: C program to find if a number is odd or even.

```
#include<stdio.h>
int main()
{
   int n;
   printf("Enter a number:");
   scanf("%d",&n);
   if(n%2 == 0)
      printf("%d is even",n);
   else
      printf("%d is odd",n);
   return 0;
}
```

Nested If

Nested If Statements means if statements can be nested in any part on if statement

Example : C program to check if a number is less than 100 or not. If it is less than 100 then check if it is odd or even.

```
#include<stdio.h>
int main()
{
   int n;
   printf("Enter a number:");
   scanf("%d",&n);
   if(n<100)
   {
      printf("%d is less than 100\n",n);
      if(n%2 == 0)
         printf("%d is even",n);
      else
         printf("%d is odd",n);
   }
   else
      printf("%d is equal to or greater than 100",n);
   return 0;
}
```

Nested If

Consider the following program, which looks as if it would respond with an appropriate comment when the user types in the temperature (in degrees).

```
void main(void)
{
        int temp;

        printf("Please type in the temperature: ");
        scanf("%d", &temp);
        if(temp < 30)
                if(temp >25)
                        printf("Nice day! ");
        else
                printf("Sure is hot! ");
}
```

The problem is that the else is actually associated with the if immediately preceding it, not the first if, as the indentation would lead you to believe.

```
void main(void)
{
        int temp;

        printf("Please type in the temperature: ");
        scanf("%d", &temp);
        if(temp < 30)
            if(temp >25)
               printf("Nice day! ");
            else
               printf("Sure is hot! ");
}
```

Nested If

If you want to ensure that an else goes with an earlier if than the one it would ordinarily be matched with, you can use braces to surround the intervening if structure.

```
void main(void)
{
       int temp;

       printf("Please type in the temperature: ");
       scanf("%d", &temp);
       if(temp < 30)
           if(temp >25)
              printf("Nice day! ");
           else
              printf("Sure is hot! ");
}
```

```
void main(void)
{
       int temp;

       printf("Please type in the temperature: ");
       scanf("%d", &temp);
       if(temp < 30)
        {
           if(temp >25)
              printf("Nice day! ");
        }
       else
           printf("Sure is hot! ");
}
```

The Else-If Construct

We've seen how if-else statements can be nested. Let's look at a more complex example of this arrangement:

```c
void main(void)
{
    float num1=1.0, num2=1.0;
    char op;

    printf("Type number operator number :");
    scanf("%f %c %f", &num1, &op, &num2);
    if (op == '+')
        printf(" solution = %f", num1 + num2);
       else
        if (op == '-')
            printf(" solution = %f", num1 - num2);
        else
            if(op == '*')
                printf(" solution = %f", num1 * num2);
            else
                if(op == '/')
                    printf(" solution = %f", num1 / num2);
    printf(" \n\n ");
}
```

```c
void main(void)
{
    float num1=1.0, num2=1.0;
    char op;

    printf("Type number operator number :");
    scanf("%f %c %f", &num1, &op, &num2);
    if (op == '+')
            printf(" solution = %f", num1 + num2);
    else if (op == '-')
            printf(" solution = %f", num1 - num2);
    else if(op == '*')
            printf(" solution = %f", num1 * num2);
    else if(op == '/')
            printf(" solution = %f", num1 / num2);

    printf(" \n\n ");
}
```

Introduction To programming Using C language

Lesson -14

The switch Statement

switch case statement

The switch statement is similar to the if else-if construct but has more flexibility and a clearer format.
Break statement is essential in the switch statement.

The **expression** used in a **switch** statement must have an integral or enumerated type

```
void main(void)
{
        float num1=1.0, num2=1.0;
        char op;

        printf("Type number operator number:") ;
        scanf("%f %c %f", &num1, &op, &num2);

        switch (op)
        {
                case '+':
                        printf(" =%f", num1+ num2);
                        break;
                case '-':
                        printf(" =%f", num1- num2);
                        break;
                case '*':
                        printf(" =%f", num1 * num2);
                        break;
                case '/':
                        printf(" =%f", num1 / num2);
                        break;
                default:
                        printf("Unknown operator ");
        }
        printf("\n\n");
}
```

switch case statement

```c
#include    <stdio.h>
void main(void)
{
    int number;

    printf("Please number >= 1 and <= 10 :");
    scanf("%d",&number);
    switch(number)
    {
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
        printf(" Odd Number ");
        break;
    case 2:
    case 4:
    case 6:
    case 8:
    case 10:
        printf(" Even Number ");
        break;
    default:
        printf(" Invalid Input ");

    }
}
```

Introduction To programming Using
C language
Lesson -15

for statement

For statement

It is often the case in programming that you want to do something a fixed number of times. Perhaps you want to calculate the paychecks for 120 employees or print out the squares of all the numbers from 1 to 50. The for loop is ideally suited for such cases.

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

For statement

It is often the case in programming that you want to do something a fixed number of times. Perhaps you want to calculate the paychecks for 120 employees or print out the squares of all the numbers from 1 to 50. The for loop is ideally suited for such cases.

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

How for loop works?

- The initialization statement is executed only once.
- Then, the test expression is evaluated. If the test expression is evaluated to false, the `for` loop is terminated.
- However, if the test expression is evaluated to true, statements inside the body of `for` loop are executed, and the update expression is updated.
- Again the test expression is evaluated.

For statement

```
for (initializationStatement; testExpression; updateStatement)
{
    // statements inside the body of loop
}
```

The Initialize Expression

The first expression, the initialize expression, initializes the control variable. The initialize expression is always executed as soon as the loop is entered.

The Test Expression

The second expression, If the test expression is true the body of the loop will be executed. If the expression becomes false the loop will be terminated and control will pass to the statements following the for .

The update Expression

The third expression, change control variable each time the loop is executed. .

The Body of For Loop

Following the keyword for and the loop expression is the body of the loop. That is, the statement (or statements) that will be executed each time round the loop.

for statement

Let's look at a simple example of a for loop:

```
/* for loop */
/* prints numbers from 0 to 9 */
void main(void)
{
     int count ;

     for (count=0; count<10; count++)
          printf("count=%d\n ", count);
}
```

Type in the program and compile it. When executed, it will generate the following output:

```
count=0
count=1
count=2
count=3
count=4
count=5
count=6
count=7
count=8
count=9
```

Multiple Statements in Loops

{the opening brace, the statements, and the closing brace} - is treated as a single C statement, with the semicolon understood. This is often called a "compound statement" or "block."

```c
#include <stdio.h>
void main(void)
{
    /* sum 10 numbers */
    int count,value,  total=0;
    for (count=1; count<=10; count++)
    {
        printf("Please value [%d]: ",count);
        scanf("%d",&value);
        total = total + value;
    }

    printf("Sum of 10 numbers is %d\n",total);

}
```

```
Please value [1]: 1
Please value [2]: 2
Please value [3]: 3
Please value [4]: 4
Please value [5]: 5
Please value [6]: 6
Please value [7]: 7
Please value [8]: 8
Please value [9]: 9
Please value [10]: 15
Sum of 10 numbers is 60

Process returned 24 (0x18)   execution time : 24.297 s
Press any key to continue.
```

Nested for

C programming allows to use one loop inside another loop. The following section shows a example to illustrate the nested for concept.

```
for ( init; condition; increment )
{
   optional statement
   for ( init; condition; increment )
   {
      statement(s);
   }
   optional statement
}
```

outer loop

inner loop

Multiplication table

```c
#include <stdio.h>
void main(void)
{
    /* multiplication table */
    int row,col;

    for (row=1; row<=10; row++)
        {
            for (col=1; col<=10; col++)
                printf("%4d",row*col);
            printf("\n");
        }
}
```

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100

Process returned 10 (0xA)   execution time : 1.800 s
Press any key to continue.
```

The Break And Continue Statements

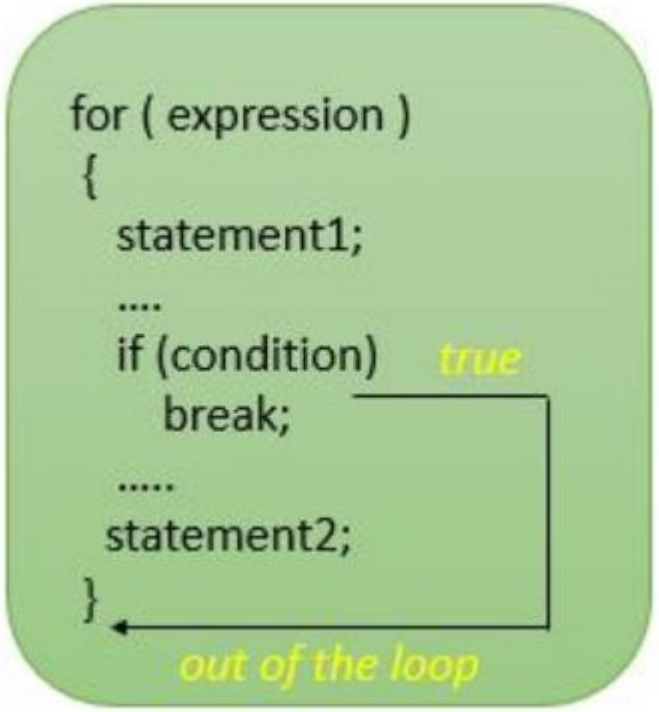
C has two statements which can be used with any of the loops described above: break and continue.

The ***break*** statement bails you out of a loop as soon as it's executed. It's often used when you want to leave the loop.

The ***continue*** statement is inserted in the body of the loop, and, when executed, takes you back to the beginning of the loop, bypassing any statements not yet executed.

The break statement

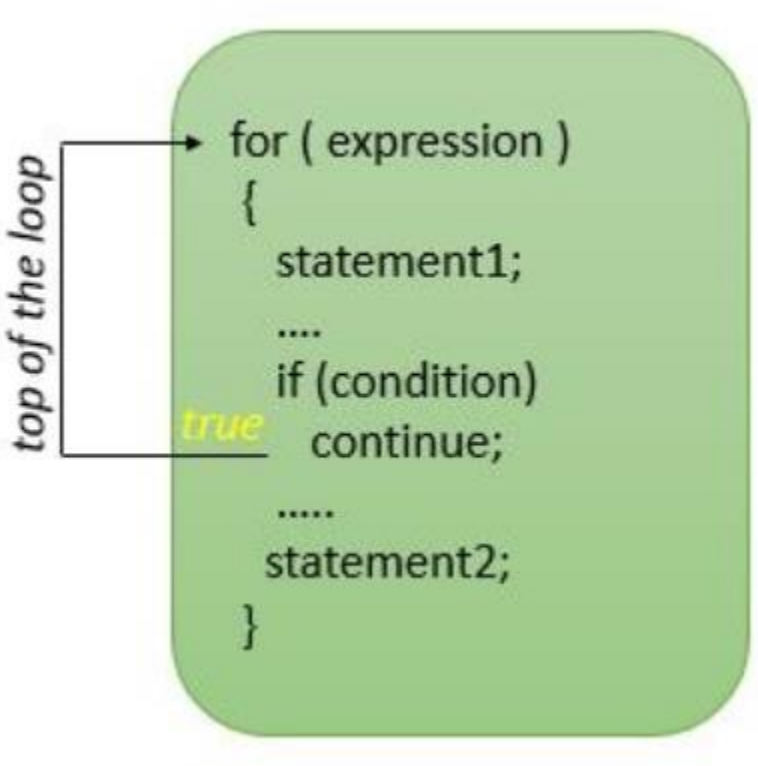
In any loop break is used to jump out of loop skipping the code below it without caring about the test condition.




```c
#include <stdio.h>
void main(void)
{
    /* sum 10 numbers or stop sum if enter 0 */
    int count,value,  total=0;
    for (count=1; count<=10; count++)
        {
            printf("Please value to stop enter zero [%d] : ",count);
            scanf("%d",&value);
            if (value==0) break;
            total = total + value;
        }

    printf("Sum of 10 numbers is %d\n",total);

}
```

The **continue** statement

Like a break statement, continue statement is also used with if condition inside the loop to alter the flow of control.it skips the remaining statements in the body of that loop and performs the next iteration of the loop.




```c
#include <stdio.h>
void main (void)
{
 int i,sum = 0;
 for (i = 0; i <= 10; i++)
 {

   if (i%2==0)
      continue;
   sum = sum + i;

  }
 printf("sum of odd numbers = %d",sum);

}
```

Introduction To programming Using
C language
Lesson -16

while Statement

While statement

A loop is used for executing a block of statements repeatedly until a given condition returns false.

```
while (condition test)
{
      //Statements to be executed repeatedly

}
```

```
#include <stdio.h>
int main()
{
   int count=1;
   while (count <= 4)
   {
        printf("%d ", count);
        count++;
   }
   return 0;
}
```

Output:

```
1 2 3 4
```

While statement

```c
#include <stdio.h>
void main (void)
{
 char ch;

 ch=getch();
 while(ch!=27)
 {
     printf("You press %c with Ascii code %d character\n",ch,ch);
     ch=getch();
 }


}
```

Assignment Expressions As Values :

The most radical aspect of the some program is the use in the inner while loop test expression of a complete assignment expression as a value:

```c
while((ch=getch())!=27)
{
    printf("You press %c with Ascii code %d character\n",ch,ch);
}
```

```
You press a with Ascii code 97 character
You press s with Ascii code 115 character
You press d with Ascii code 100 character
You press f with Ascii code 102 character
You press   with Ascii code 0 character
You press ; with Ascii code 59 character
You press   with Ascii code 0 character
You press < with Ascii code 60 character
You press   with Ascii code 0 character
You press = with Ascii code 61 character
You press α with Ascii code -32 character
You press H with Ascii code 72 character
You press α with Ascii code -32 character
You press P with Ascii code 80 character
You press α with Ascii code -32 character
You press M with Ascii code 77 character
You press α with Ascii code -32 character
You press K with Ascii code 75 character
```

While statement

```c
#include <stdio.h>
void main(void)
{
    int n;
    float num,sum;
    int i;
    printf("\nplease enter values terminated by 0 to stop \n");

    i=1;sum=0;
    printf("value number: %2d: ", i);
    scanf("%f", &num);
    while (num!=0)
        {
            sum +=num;
            i++;
            printf("value number: %2d: ", i);
            scanf("%f", &num);
        }

    printf("\n Sum of %d previous numbers = %.2f\n",--i ,sum);
}
```

```
please enter values terminated by 0 to stop
value number:  1: 5
value number:  2: 7
value number:  3: 8
value number:  4: 0

 Sum of 3 previous numbers = 20.00

Process returned 36 (0x24)   execution time : 8.873 s
Press any key to continue.
```

Introduction To programming Using
C language
Lesson -17

The Do While Loop

Do While

The last of the three loops in C is the do while loop. This loop is very similar to the while loop the difference is that in the do loop the test condition is evaluated after the loop is executed, rather than before.

The do loop, unlike the other loops we've examined, has two keywords: do and while. The do keyword marks the beginning of the loop; it has no other function. The while keyword marks the end of the loop and contains the loop expression.

Do While

When would you use a do loop? Any time you want to be sure the loop body is executed at least once.

```
do
{
//code that is to be executed
} while(condition);
```

```c
#include <stdio.h>
int main()
{
        int j=0;
        do
        {
                printf("Value of variable j is: %d\n", j);
                j++;
        }while (j<=3);
        return 0;
}
```

```
Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
```

Fibonacci Series

Fibonacci series is nothing but a Series of Numbers which are found by adding the two preceding(previous) Numbers. For Example,
0,1,1,2,3,5,8,13,21 is a Fibonacci Series of length 9.

```c
#include <stdio.h>
void main(void)
{
    int n,f,f1,f2;

    printf(" Enter The Number Of Terms:");
    scanf("%d",&n);

    f1=-1;
    f2=1;
    do
    {
        f=f1+f2;
        f1=f2;
        f2=f;
        printf("%d\n",f);
        n--;
    } while(n>0);

}
```

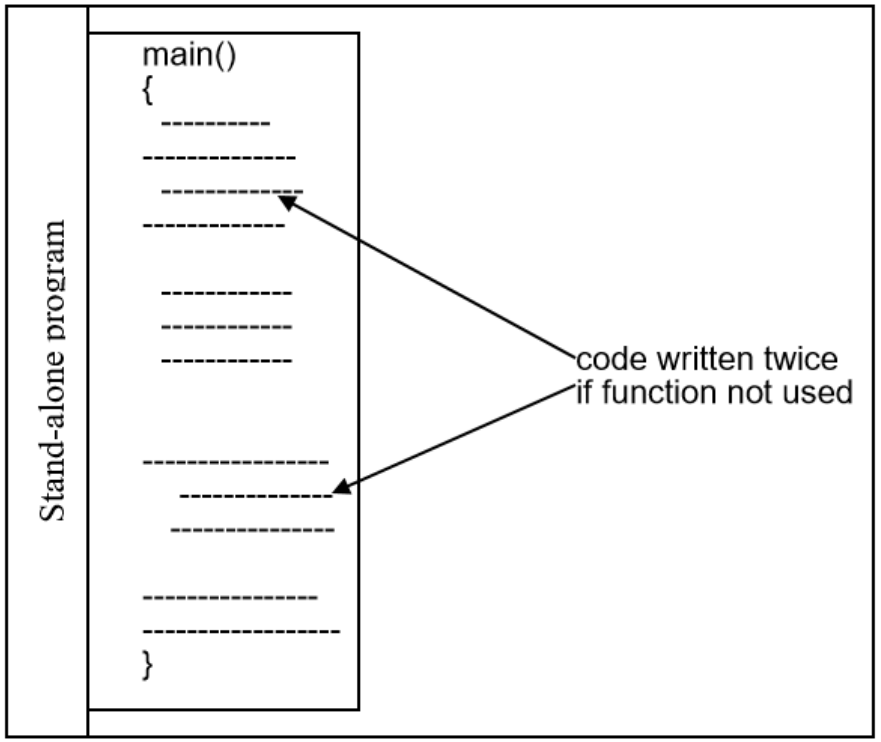
```
 Enter The Number Of Terms:10
0
1
1
2
3
5
8
13
21
34

Process returned 0 (0x0)   execut
Press any key to continue.
```

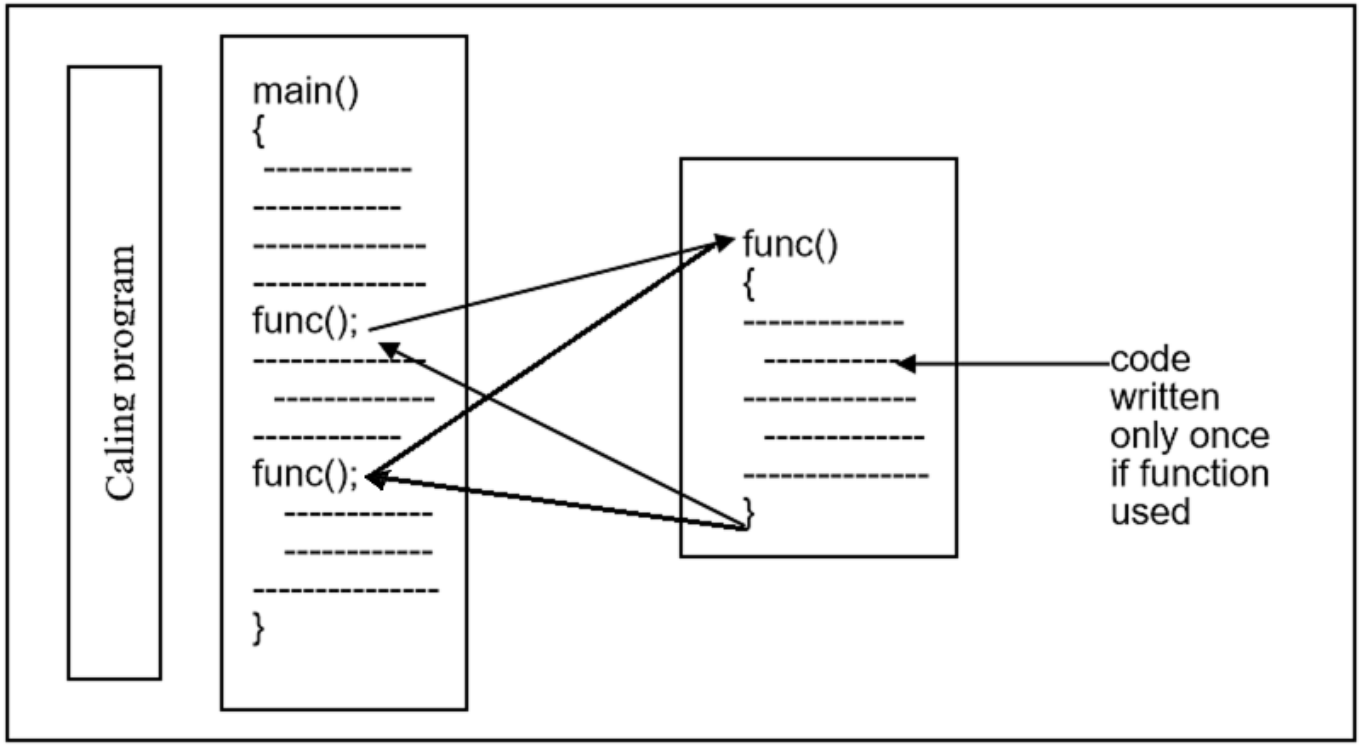
Introduction To programming Using
C language
Lesson -18

Functions

Functions

Probably the original reason functions for were invented was to avoid having to write the same code over and over.

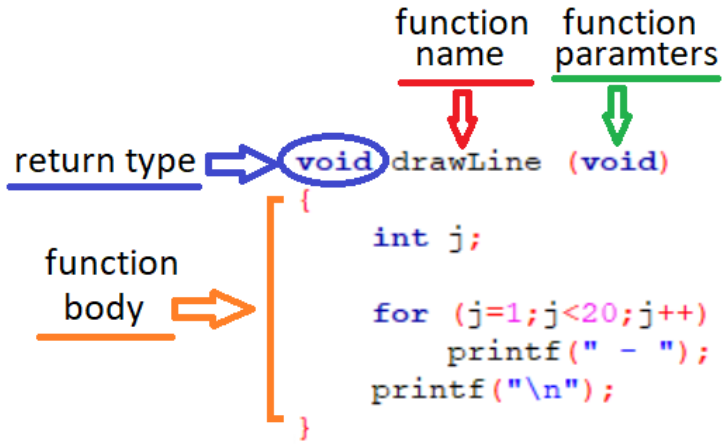


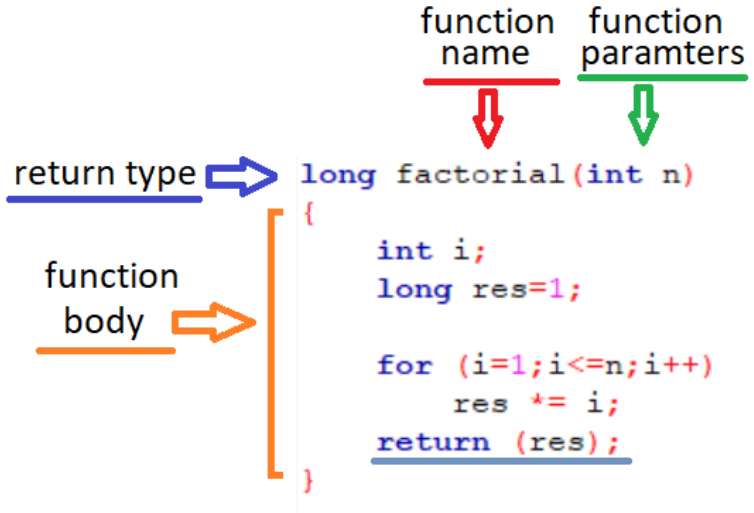


The Function Definition

The function itself is referred to as the function definition:

1. The definition starts with a data type that is being returned and if the function does not return any data please specify “void”
2. followed by the function name it is preferable to give it name illustrate its function task
3. followed by two brackets ()
4. inside this two brackets my be you have a parameters which is the symbolic name for "data" that goes into a function with specified data type if the function does not have any parameters please specify “void”
5. The function definition continues with the body of the function by two braces {} and with some empty lines for implementation its tasks
6. inside this braces the program statements that do the work.
7. if the function return any type so before leave the body of function you must return a value with this type .





function prototype

a function prototype or function interface is a declaration of a function that specifies the function's name and type signature (data types of parameters, and return type) .

In a prototype, parameter names are optional however, the type is necessary along with all modifiers.

Function prototype must be known before function call.

There's a third function-related element normally:

- A prototype declares a function
- A function definition is the function itself
- A function call executes a function

```
#include <stdio.h>
void drawLine (void);
long factorial(int n);

void main(void)
{
    printf("factorial 5 = %ld\n",factorial(5));
    drawLine();
    printf("factorial 6 = %ld\n",factorial(6));
    drawLine();
    printf("factorial 7 = %ld\n",factorial(7));
    drawLine();
}

void drawLine (void)
{
    int j;

    for (j=1;j<20;j++)
        printf(" - ");
    printf("\n");
}
long factorial(int n)
{
    int i;
    long res=1;

    for (i=1;i<=n;i++)
        res *= i;
    return (res);
}
```

function prototype (decleartion)

function call

function defination

Calling The Function

As with the C library functions we've seen, such as printf() and scanf(), our user-written function drawLine() is called from main() simply by using its name, including the parentheses following the name ends with a semicolon.

If the function required argument you must send it to the function between parentheses to call it correctly .

If the function does not return any value “void” this function act as procedure and must call as speared statement like the example :

drawLine();

If the function return a value of any type so the function act as function and must call as apart of expression like the examples:

Answer = factorial (4);
Answer = 2* factorial(3)+2;
printf (“factorial 3= %d\n”,factorial(3));

call the return value of function

call the function

```
void main(void)
{
    printf("factorial 5 = %ld\n",factorial(5));
    drawLine();
    printf("factorial 6 = %ld\n",factorial(6));
    drawLine();
    printf("factorial 7 = %ld\n",factorial(7));
    drawLine();
}
```

Advantages Of Prototyping

Their major advantage is that the data types of a function's arguments are clearly specified at the beginning of a program. A common was to call a function using the wrong data type for an argument; If the function call and the function definition were in different files, this lead to program failure in a way that was very difficult to debug, because there was no warning from the compiler.

When a prototype is used, however, the compiler knows what data types to expect as arguments for the function, and is always able to flag a mismatch as an error. Also as in declaring variables-the prototype clarifies for the programmer and anyone else looking at the listing what each function is and what its arguments should be.

Argument names can be used in the prototype as a reminder of what the function arguments are. So far we've shown prototypes without these names. However, a human being reading the source code may be confused as to which argument is which, because they both have the same data type. To clarify the situation, the prototype can be changed to

Local Variables

The variable j used in the drawLine() function is known only to drawLine(); it is invisible to the main() function. If we added this statement to main() (without decelerating a variable j there):

printf(" %d ", j);

We could declare another variable, also called j, in the main() function; it would be a completely separate variable, known to main() but not to drawLine(). This is a key point in the writing of C functions: variables used in a function are unknown outside the function.

A local variable will be visible to the function it is defined in, but not to others.

A local variable used in this way in a function is known in C as an "automatic" variable, because it is automatically created when a function is called and destroyed when the function returns.

```
void drawLine (void)
{
    int j;
    for (j=1;j<20;j++)
        printf(" - ");
    printf("\n");
}
```



```
long factorial(int n)
{
    int i;
    long res=1;
    for (i=1;i<=n;i++)
        res *= i;
    return (res);
}
```

The Return Statement

The return() statement has two purposes:

1. First, executing it immediately transfers control from the function back to the calling function.
2. Second, whatever is inside the parentheses following return is returned as a value to the calling function.

The return statement need not be at the end of the function. It can occur anywhere in the function; as soon as if s encountered, control will return to the calling function .

Limitation Of return():

- Return can only use it to return one value. If you want your function to return two or more values to the calling program, you need another mechanism.

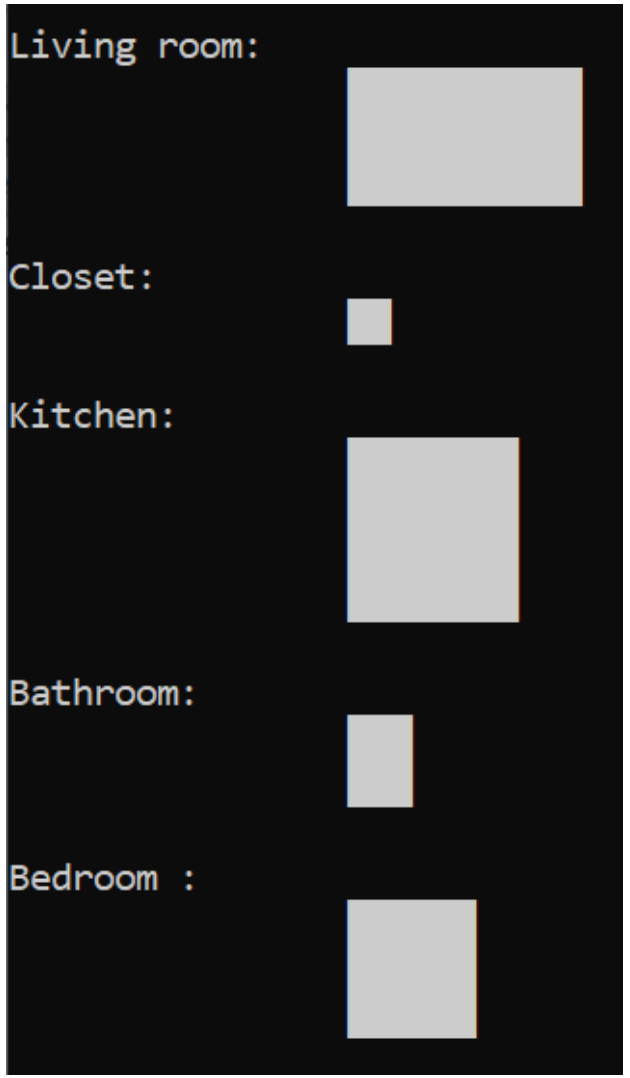
Pass Data To A Function

The mechanism used to send information to a function is the argument. You've already used arguments in printf() and scanf() functions.

Passing Arguments :

We can pass as many arguments as we like to a function. Here's an example of a program that passes two arguments to a function, drawRect(), whose purpose is to draw variously sized rectangles on the screen.

```
#include <stdio.h>
void drawRect(int length, int width);
void main(void)
{
    printf(" \nLiving room:\n ");  /* print room name */
    drawRect(22, 12);              /* draw room */
    printf("\nCloset:\n");        /* etc. */
    drawRect(4, 4);
    printf("\nKitchen:\n");
    drawRect(16, 16);
    printf("\nBathroom:\n");
    drawRect( 6,8);
    printf(" \nBedroom :\n");
    drawRect( 12, 12);
}
void drawRect(int length, int width)
 {
    int j, k;

    length /= 2;     /* horizontal scale factor */
    width  /= 4;      /*vertical scale factor */

    for (j=1;j<=width;j++)   /* number of lines */
    {
        printf("\t\t");      /* tab over */
        for (k= 1; k<=length; k++ )   /* line of rectangles */
            printf("\xDB");   /* print one rectangle */
        printf("\n");    /* next line */
    }
 }
```



call by Value

When we pass the actual parameters while calling a function then this is known as function call by value. In this case the values of actual parameters are copied to the formal parameters. Thus operations performed on the formal parameters don't reflect in the actual parameters.

```c
#include <stdio.h>
void duplicate(int a);
void main(void)
{
    int number=5;


    printf("initial value of number = %d\n",number);
    increment(number);
    printf("value of number after duplicate = %d\n",number);


}

void increment(int a)
{
    a++;
}
```

```
initial value of number = 5
value of number after duplicate = 5

Process returned 36 (0x24)   execution time : 1.656 s
Press any key to continue.
```

Introduction To programming Using
C language
Lesson -19

variable Scope or availability

Scope –availability

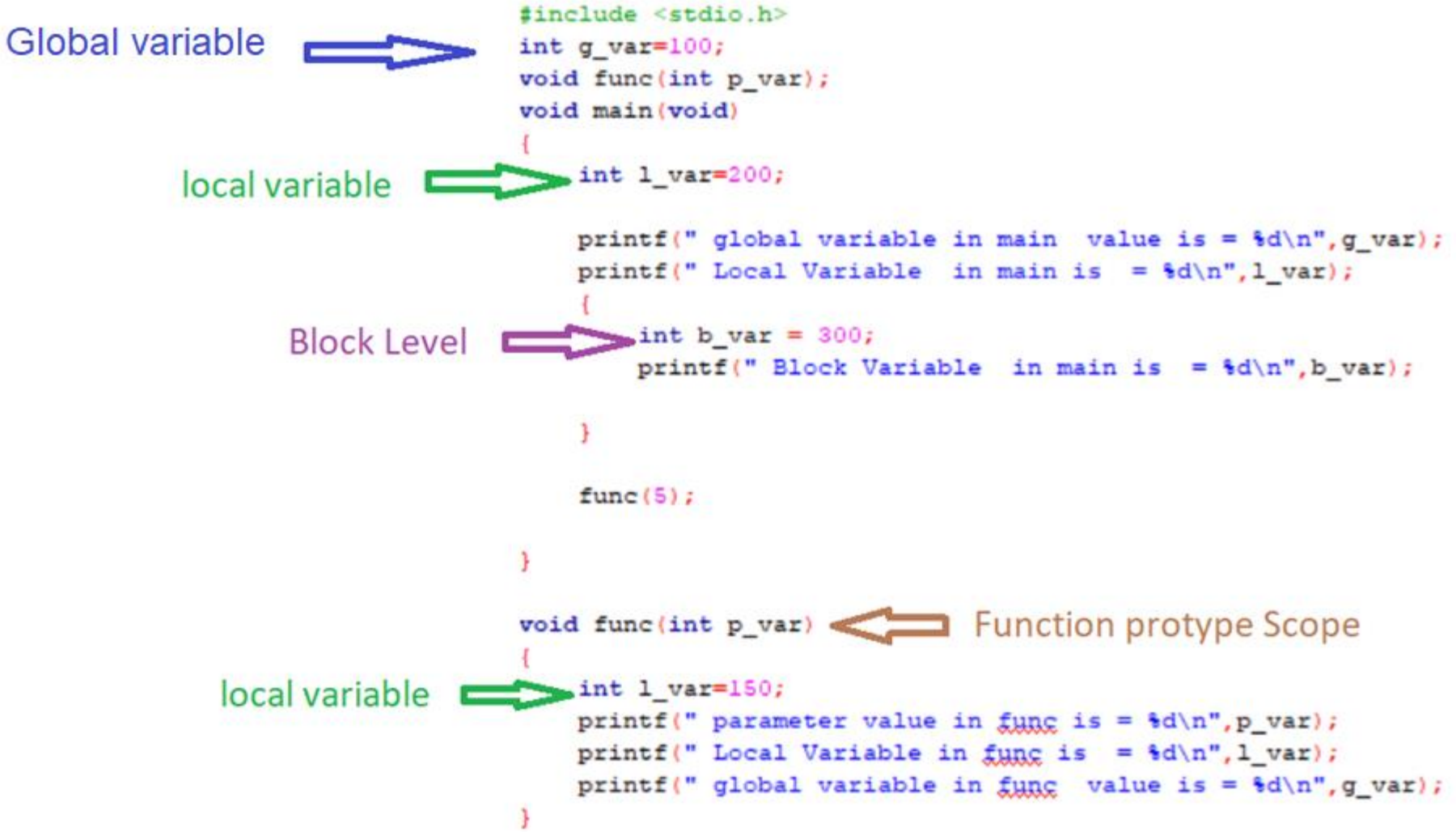
File or global Scope:
These variables are usually declared outside of all of the functions,
at the top of the program and can be accessed from any portion of the program.

Function Scope local scope:
A Function scope begins at the opening of the function and ends with the closing of it.

Function Prototype Scope:
These variables range includes within the function parameter list.
its similar to local variable.

Block Scope:
A Block is a set of statements enclosed within left and right braces i.e.
'{' and '}' respectively. Basically these are local to the blocks in
which the variables are defined and are not accessible outside.

Scope –availability



Introduction To programming Using
C language
Lesson -20

variable Storage Classes or Lifetime

Storage Classes

Every C variable possesses a characteristic called its storage class. The storage class defines two characteristics of the variable: its lifetime, and its visibility (or scope).

by using variables with the appropriate lifetime and visibility we can write programs that use memory more efficiently, run faster, and are less to programming errors. Correct use of storage class is especially important in large programs.

In terms of their lifetime, variables can be divided into two categories:
automatic variables have shorter lifetimes than do static and external variables.

Lifetime

In terms of their lifetime, variables can be divided into two categories:
automatic variables have ***shorter*** lifetimes than do static and external variables.

Automatic Variables:
Automatic variables are the most commonly used in C; They are created (that is, memory space is allocated to them) when the function containing them is called, and destroyed (their memory space is "deallocated") when the function terminates.

Automatic Variables

```
Func()
 {
        int alpha;
        auto int beta;
        register int gamma;
}
```

All three variables are created when the function func() is called and disappear when it has finished and control returns to the calling function.

Such variables are called "automatic" because they are created and destroyed automatically. (Automatic variables are stored on the ***stack***.)

Variables of type auto and register are created when the function containing them is called and destroyed when control returns to the calling program.

Static Variables

```
Func()
{
        static int delta;
}
```

If we want a variable to retain its value after the function that defines it is terminated, we have several choices. First. the variable can be defined to be of type static, as shown in this example

A static variable is known only to the function in which it is defined, but, unlike automatic variables, it does not disappear when the function terminates.

Instead it keeps its place in memory and therefore its value. In the example above, even after the function func() has terminated, delta will retain the value it was given in the function. If program control returns to the function again, the value of delta will be there for the function to use.

external Variables

program is to make its variable external, by placing the variable outside of any function Like static variables, external variables exist for the life of the program. The difference between them has to do with their visibility

Register Variables

Register Variables Another storage class specifier is register.
This specifier is like auto in that it indicates a variable with a visibility and a lifetime limited to the function in which it is defined.

The difference is that the compiler, if possible, will assign a register variable to one of the microprocessor's registers instead of storing it in memory. Registers can be accessed much faster than memory locations, so using the register class for variables that are frequently accessed can result in significantly increasing a program's speed.

Introduction To programming Using C language

Lesson -21

recursion

recursion

A recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem

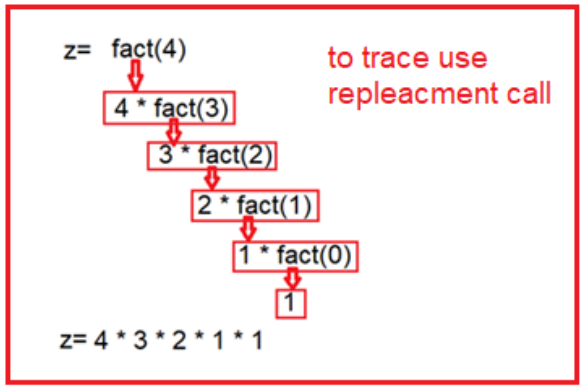
Most computer programming languages support recursion by allowing a function to call itself from within its own code.
for problems that can be solved easily by iteration, recursion is generally less efficient.

For example, the factorial function can be defined recursively by the equations:
$0! = 1$ and,
for all $n > 0$, $n! = n(n - 1)!$.

Recursion
with Return Value

```c
#include <stdio.h>
void main(void)
{
    int number1 ;
    int z;

    printf("please number to get factorial:");
    scanf("%d",&number1);

    z= fact(number1);
    printf(" factorial %d = %d\n",number1,z);

}

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

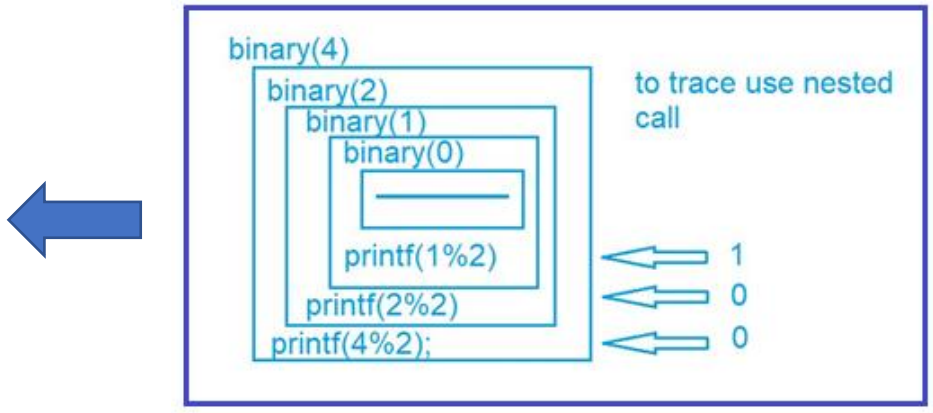
```
please number to get factorial:5
 factorial 5 = 120

Process returned 19 (0x13)   execution time : 3.910 s
Press any key to continue.
```



Recursion without Return Value

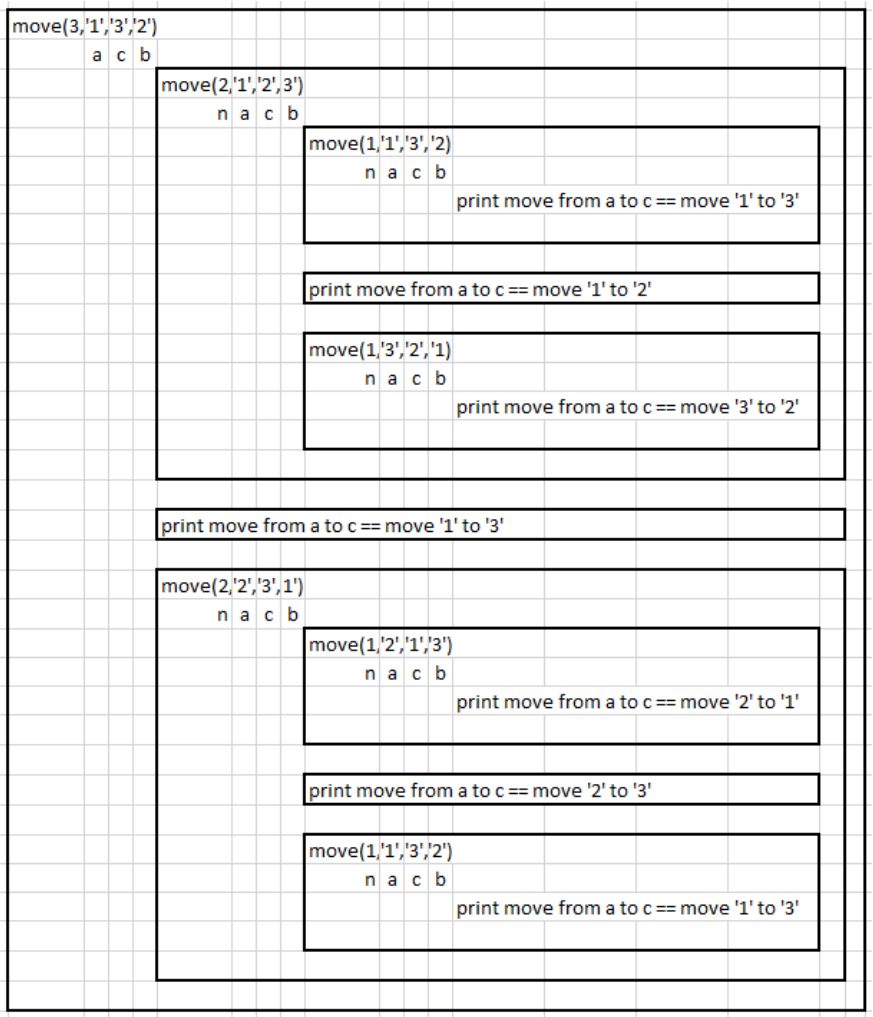
```c
#include <stdio.h>
void main(void)
{
    int number ;



    printf("please number to convert to binary:");
    scanf("%d",&number);
    printf(" binary of %d  is = ",number);
    binary(number);
}

void binary(int n)
{
    if (n>0)
    {
        binary(n/2);
        printf("%d",n%2);
    }
}
```

```
please number to convert to binary:4
 binary of 4  is = 100
Process returned 1 (0x1)   execution time : 4.496 s
Press any key to continue.
```



Two level Recursion

```c
#include <stdio.h>
void main(void)
{
    int n ;
    char a='1';
    char b='2';
    char c='3';


    printf("please number to Disks to move:");
    scanf("%d",&n);
    move(n,a,c,b);
}

void move(int n,char a,char c,char b )
{
    if (n==1)
        printf("Move Disk from %c to %c\n",a,c);
    else
    {
            move(n-1,a,b,c);
            printf("Move Disk from %c to %c\n",a,c);
            move(n-1,b,c,a);
    }
}
```




```
Move Disk from 1 to 3
Move Disk from 1 to 2
Move Disk from 3 to 2
Move Disk from 1 to 3
Move Disk from 2 to 1
Move Disk from 2 to 3
Move Disk from 1 to 3
```

How to think recursion

Introduction To programming Using C language

Lesson -22

Arrays

Needs of Arrays

If you have a collection of similar data elements, you may find it convenient to give each one a unique variable name. For instance, suppose you wanted to find the average temperature for a particular week. If each day's temperature had a unique variable name, you would end up reading in each value separately:

```c
#include<stdio.h>
void main(void)
{
    int day1 , day2 , day3 , day4 , day5 , day6 , day7 ;
    float avg;
    printf("Enter temperature Day 1: ");
    scanf("%d", & day1);
    printf("Enter temperature Day 2: ");
    scanf("%d", & day2);
    printf("Enter temperature Day 3: ");
    scanf("%d", & day3);
    printf("Enter temperature Day 4: ");
    scanf("%d", & day4);
    printf("Enter temperature Day 5: ");
    scanf("%d", & day5);
    printf("Enter temperature Day 6: ");
    scanf("%d", & day6);
    printf("Enter temperature Day 7: ");
    scanf("%d", & day7);

    avg=(day1+day2+day3+day4+day5+day6+day7)/7.0;
    printf("average is = %7.2f\n",avg);


}
```

```
Enter temperature Day 1: 33
Enter temperature Day 2: 35
Enter temperature Day 3: 29
Enter temperature Day 4: 30
Enter temperature Day 5: 44
Enter temperature Day 6: 41
Enter temperature Day 7: 40
average is =   36.00

Process returned 21 (0x15)   execution time : 15.527 s
Press any key to continue.
```

What is the case if you want to find the average temperature for a particular month ????

Array Definition

Array Definition An array is a collection of variables of a certain type, placed contiguously in memory. Like other variables, the array needs to be defined, so the compiler will know what kind of array, and how large an array, we want.

We do that in the example above with the line:

Instead of the following code :

int day1 , day2 , day3 , day4 , day5 , day6 , day7 ; ======➔ int day[7];

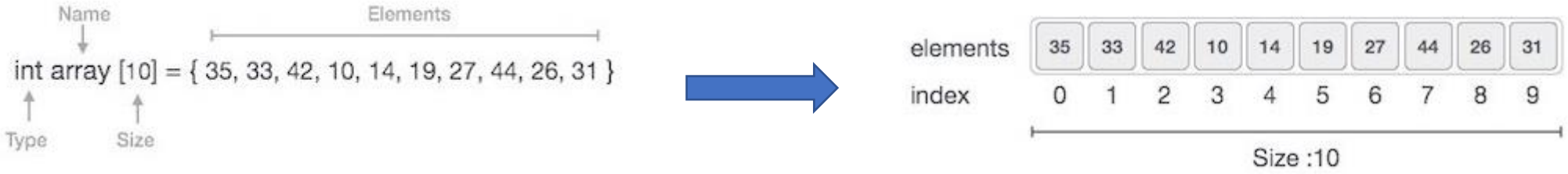
An array in is a collection of items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They are used to store similar type of elements as in the data type must be the same for all elements.

Array Representation

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

•Element – Each item stored in an array is called an element.
•Index – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation:
Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Reading In An Unknown Number Of Elements

So far we've worked with a fixed amount of input, requiring a data item for each of the days of the week. What if we don't know in advance how many items will be entered into the array? Here's a program that will accept any number of temperatures -up to 366 - and average them

```
/* averages arbitrary number of temperatures */
void main(void)
{
    float day[366];
    float sum =0.0;
    int num, i=0;
    do
    {
        printf("Enter temperature for day %d: " , i+1);
        scanf("%f", &day [i]);
    }while ( day[i++] >0 );
    num = i-1;
    for (i=0; i<num; i++)
        sum += day[i];
    printf("Average is %.2f", sum/num);
}
```

```
Enter temperature for day 1: 23
Enter temperature for day 2: 33
Enter temperature for day 3: 34
Enter temperature for day 4: 36
Enter temperature for day 5: 38
Enter temperature for day 6: 44
Enter temperature for day 7: 28
Enter temperature for day 8: 34
Enter temperature for day 9: 4
Enter temperature for day 10: 40
Enter temperature for day 11: 30
Enter temperature for day 12: 0
Average is 31.27
Process returned 16 (0x10)   execution time : 32.159 s
Press any key to continue.
```

Bounds Checking

suppose a user decided to enter data more that the array size ? As it turns out, there probably would be Big Trouble. The reason is that in C there is no check to see if the subscript used for an array exceeds the size of the array.

Data entered with too large a subscript will simply be placed in memory outside the array: probably on top of other data or the program itself. This will lead to unpredictable results, to say the least, and there will be no error message from the compiler to warn you that it's happening.

The solution, if there's the slightest reason to believe the user might enter too many items, is to check for this possibility in the program.

```
/* averages arbitrary number of temperatures */
#define MAX 5
void main(void)
{
    float day[MAX];
    float sum =0.0;
    int num, i=0;
    do
    {   if (i<MAX)
        {
            printf("Enter temperature for day %d: " , i+1);
            scanf("%f", &day [i]);
        }
        else
        {
            printf("index out of range !!!!\n");
            break;
        }

    }while ( day[i++] >0 );
    num = i-1;
    for (i=0; i<num; i++)
        sum += day[i];
    printf("Average is %.2f", sum/num);
}
```

Initializing Arrays

Initializing Arrays So far, we've shown arrays that started life with nothing in them. Suppose we want to compile our program with specific values already fixed in the array.
The list of values is enclosed by braces, and the values are separated by commas. The values are assigned in turn to the elements of the array, so that day[0] is 22, day[1] is 32, and so on, up to day[4], which is 45.

```c
/* averages arbitrary number of temperatures */
#include <stdio.h>
#define MAX 5
void main(void)
{
    float day[MAX]={22,32,34.6,44,45};
    float sum =0.0;
    int num, i=0;

    for (i=0; i<MAX; i++)
        sum += day[i];
    printf("Average is %.2f", sum/MAX);
}
```

Access Arrays elements using Array size

C does not provide a built-in way to get the size of an array. You have to do some work up front.
The simplest procedural way to get the value of the length of an array is by using the sizeof operator.

```c
#include <stdio.h>
void main(void)
{

    int data[]={3,4,6,8,14,17,22,33,88};


    int n = sizeof (data) / sizeof (data[0]);
    printf("size of data %d\n",n);

    printf("Min value = %d\n",data[0]);
    printf("Max value = %d\n",data[n-1]);
    printf("range value = %d\n",data[n-1]-data[0]);
    if (n%2==0)
        printf("median value = %d\n",data[n/2+1]);
    else
        printf("median value = %7.2f\n",(data[n/2]+data[n/2])/2.0);


}
```

Arrays As Arguments

We've seen examples of passing various kinds of variables as arguments to functions. Here's a program that uses a function called max() to find the element in an array with the largest value.

```
/* tells largest number in array typed in */
#define MAXSIZE 20
int max(int[],int);      /* prototype */

void main(void)
{
        int list[MAXSIZE];
        int size = 0;
        int num;
        do
                {
                printf("Type number: ");
                scanf("%d", &list[size]);
                }

          while ( list[size++] !=0);            /* exit loop on 0 */
          num= max(list,size-1);
        printf("Largest number is %d", num);
}

int max (int list[], int size)
{
        int dex, max;
        max = list[0];
        for (dex=1; dex<size; dex++)
                if(max < list[dex])
                        max = list[dex];
        return(max);
}
```

Addresses Of Things Versus Things

It's important to realize that passing the address of something is not the same thing as passing the something. When. a simple variable name is used as an argument passed to a function, the function takes the value corresponding to this variable name and installs it as a new variable in a new memory location created by the function for that purpose.

But what happens when the address of an array is passed to a function as an argument? Does the function create another array and move the values into it from the array in the calling program? No. Since arrays can be very large, the designers of C determined that it would be better to have only one copy of an array no matter how many functions wanted to access it. So instead of passing the values in the array, only the address of the array is passed. The function then uses the address to access the original array.

Function call by value or by reference

Before we discuss function call by value, lets understand the terminologies that we will use while explaining this:
Actual parameters: The parameters that appear in function calls.
Formal parameters: The parameters that appear in function declarations.

When we pass the actual parameters while calling a function then this is known as ***function call by value***. In this case the values of actual parameters are copied to the formal parameters. Thus operations performed on the formal parameters don't reflect in the actual parameters.

When we pass the address of an array while calling a function then this is called ***function call by reference***. When we pass an address as an argument, the function declaration should have a pointer as a parameter to receive the passed address.

int list[10];

Since list[0] is the first element of the array, it will have the same address as the array itself. And since & is the address operator, &list[0] gives the address of the first element of the array. In other words,
list == &list[0]

Introduction To programming Using C language

Lesson -23

Arrays Operations

Arrays Operations

- Traverse data
- Read data
- Insertion item
- Deletion item
- Searching item
- Sorting data

Printing Array

Traverse data:

You can traverse through an array using for loop . Using the for loop − Instead on printing element by element, you can iterate the index using for loop starting from 0 to length-1 of the array and access elements at each index.

```
for (i=0;i<n;i++)
    printf("%d\n",data[i]);
```

Reading Array

Read data:

You can traverse through an array using for loop . Using the for loop − Instead on reading element by element, you can iterate the index using for loop starting from 0 to length-1 of the array and read elements at each index from keyboard using scanf(“%type”,&array[i]) or read from other source like a file .

```
for (i=0;i<n;i++)
   {
      printf("value[%d]:",i+1);
      scanf("%d",&data[i]);
   }
```

Insert Element to Array

Insertion items :

Insertion operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array within the original size.

```
for(i=size; i>=pos; i--)
        arr[i+1] = arr[i];
    arr[pos] = num;
    size++;
```

```c
#include <stdio.h>
#define MAX_SIZE 100
void main(void)
{
    int arr[MAX_SIZE];
    int i, size, num, pos;

    printf("Enter size of the array<100 : ");
    scanf("%d", &size);

    printf("Enter elements in array : \n");
    for(i=0; i<size; i++)
        {
            printf("value[%d]:",i);
            scanf("%d", &arr[i]);
        }
    printf("Enter element to insert : ");
    scanf("%d", &num);
    printf("Enter the element position : ");
    scanf("%d", &pos);

    /* If position of element is not valid */
    if(pos > size || pos < 0)
        printf("Invalid position! Please enter position between 0 to %d", size);
    else
    {
        /* Make room for new array element by shifting to right */

        for(i=size; i>=pos; i--)
            arr[i+1] = arr[i];
        arr[pos] = num;
        size++;

        printf("Array elements after insertion : \n");
        for(i=0; i<size; i++)
        {
            printf("value[%d]:",i);
            printf("%d\n", arr[i]);
        }
    }
}
```

Delete Element From Array

Deletion items :

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

```
for(i=pos; i<size-1; i++)
        arr[i] = arr[i+1];
    size--;
```

```c
#include <stdio.h>
#define MAX_SIZE 100
void main(void)
{
    int arr[MAX_SIZE];
    int i, size, pos;

    printf("Enter size of the array<100 : ");
    scanf("%d", &size);

    printf("Enter elements in array : \n");
    for(i=0; i<size; i++)
        {
            printf("value[%d]:",i);
            scanf("%d", &arr[i]);
        }

    printf("Enter the element position to be delete : ");
    scanf("%d", &pos);

    /* If position of element is not valid */
    if(pos >= size || pos < 0)
        printf("Invalid position! Please enter position between 0 to %d", size-1);
    else
    {
        /* remove  room in this position by shifting to left */
        for(i=pos; i<size-1; i++)
            arr[i] = arr[i+1];
        size--;

        printf("Array elements after deletion : \n");
        for(i=0; i<size; i++)
        {
            printf("value[%d]:",i);
            printf("%d\n", arr[i]);
        }
    }
}
```

Search a specific value in Array

Searching item:

Finding a specific member of an array means searching the array until the member is found. It's possible that the member does not exist and the programmer must handle that possibility within the logic of his or her algorithm.

Unsorted data :

-Linear Search

sorted data:

-Binary Search

Sorting the Arrays Elements

Sorting data:

A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, and outputs a sorted array.

- Selection sort
- Bubble sort

Introduction To programming Using
C language
Lesson -24

Searching

Search a specific value in Array

Searching item:

Finding a specific member of an array means searching the array until the member is found. It's possible that the member does not exist and the programmer must handle that possibility within the logic of his or her algorithm.

Unsorted data :

-Linear Search

sorted data:

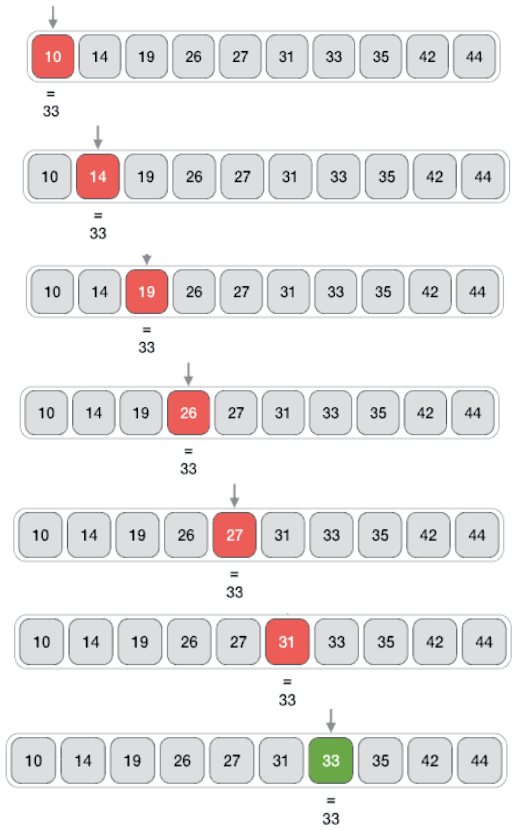
-Binary Search

Linear Search

Linear Search:

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular position is returned, otherwise the search continues till the end of the data collection.

Key=33

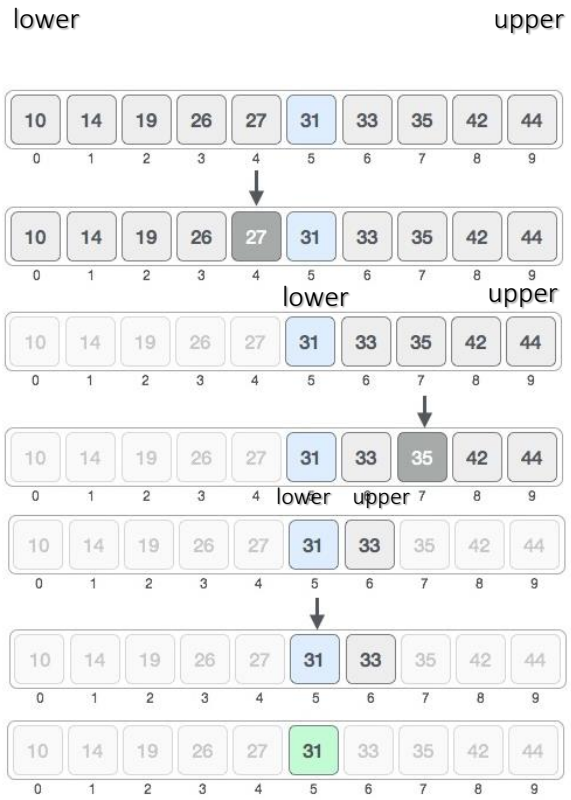



```c
#include <stdio.h>
void main(void)
{
    int data[100];
    int n,key,pos=-1;
    int i;

    printf("please size of data<100:");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("value[%d]:",i+1);
        scanf("%d",&data[i]);
    }

    printf("please key to be located in data :");
    scanf("%d",&key);
    for (i=0;i<n;i++)
    {
        if (data[i]==key)
        {
            pos=i;
            break;
        }
    }
    if (pos==-1)
    {
        printf("%d is not located !!!",key);
    }
    else
    {
        printf("%d is located at position %d\n",key,pos+1);
    }
}
```

Binary Search

Binary Search:

For this algorithm to work properly, the data collection should be in the sorted. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well Lower less than or equal upper index .



mid=(lower + upper)/2;

mid=(0 + 9)/2; ➔ mid=4

mid=(5+ 9)/2; ➔ mid=7

mid=(5+ 6)/2; ➔ mid=5

```c
#include <stdio.h>
void main(void)
{
    int data[100];
    int n,key,pos=-1;
    int lower,upper,mid;
    int i;

    printf("please size of data<100:");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("value[%d]:",i+1);
        scanf("%d",&data[i]);
    }
    printf("please key to be located in data :");
    scanf("%d",&key);

    lower=0; upper=n-1;
    do
    {
        mid=(lower+upper)/2;
        if (key==data[mid])
            {
                pos=mid+1;
                break;
            }
        else if (key>data[mid])
            lower=mid+1;
        else
            upper=mid-1;
    } while (lower <=upper);
    if (pos==-1)
        printf("%d is not located !!!",key);
    else
        printf("%d is located at position %d\n",key,pos);
}
```

Introduction To programming Using
C language
Lesson -25

Sorting

Sorting the Arrays Elements

Sorting data:

A sorting algorithm is an algorithm made up of a series of instructions that takes an array as input, performs specified operations on the array, and outputs a sorted array.

- Selection sort
- Bubble sort

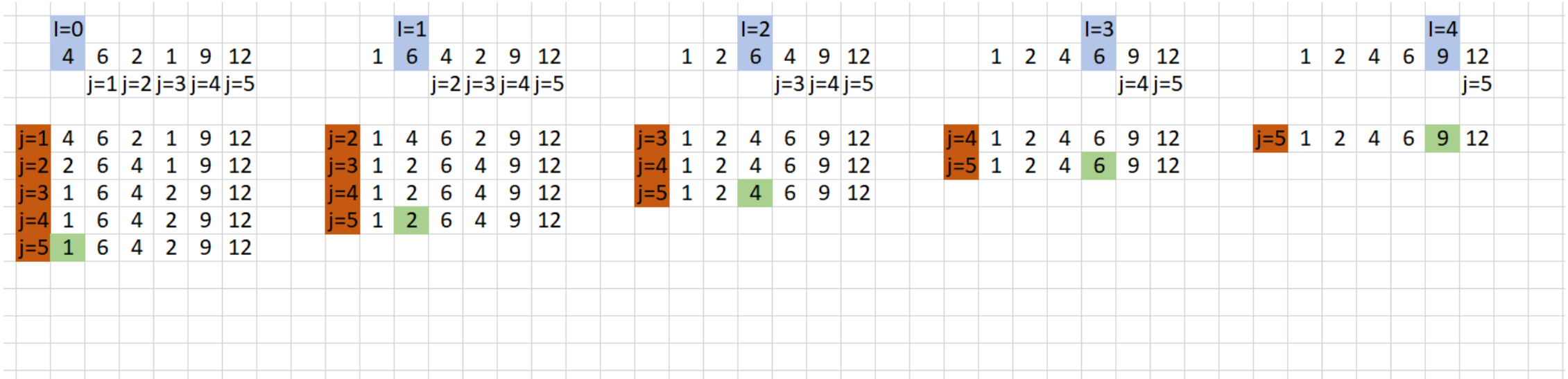
Selection sort

Selection sort :

Selection sort is a simple sorting algorithm. This sorting algorithm is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list

Before sort :

4	6	2	1	9	12



After sort :

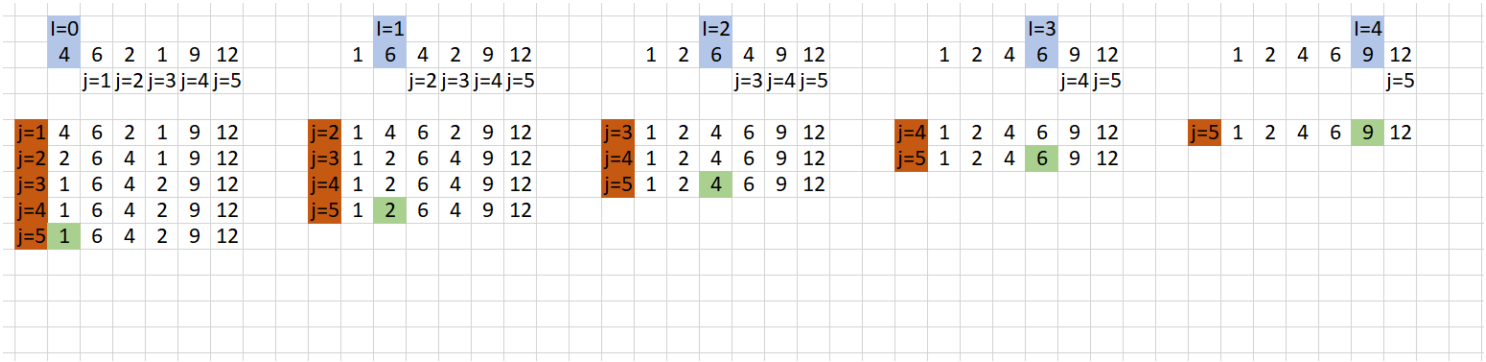
1	2	4	6	9	12

Selection sort

Selection sort :

Selection sort is a simple sorting algorithm. This sorting algorithm is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list

4	6	2	1	9	12



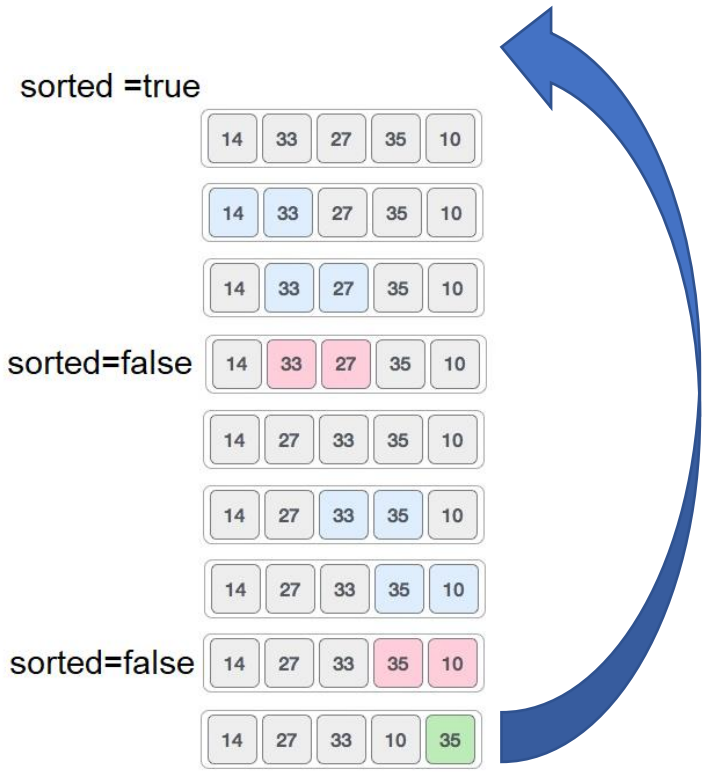
1	2	4	6	9	12

```c
#include <stdio.h>
void main(void)
{
    int data[100];
    int n;
    int i,j;

    printf("please size of data<100:");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("value[%d]:",i+1);
        scanf("%d",&data[i]);
    }


    for (i=0;i<n-1;i++)
        for (j=i+1;j<n;j++)
            if (data[i]>data[j])
                {
                    int temp;

                    temp=data[i];
                    data[i]=data[j];
                    data[j]=temp;
                }

   printf("Data after sort \n");
   printf("=============== \n");
   for (i=0;i<n;i++)
        printf("%d: %d\n",i+1,data[i]);
}
```

Bubble sort

Bubble sort :

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order and repeat this operation till start and end the inner loop without swap any adjacent elements .




```c
#include <stdio.h>
#define TRUE   1
#define FALSE  0
void main(void)
{
    int data[100];
    int n;
    int i;
    int sorted;

    printf("please size of data<100:");
    scanf("%d",&n);
    for (i=0;i<n;i++)
    {
        printf("value[%d]:",i+1);
        scanf("%d",&data[i]);
    }
    do
    {
        sorted = TRUE;
        for (i=0;i<n-1;i++)
            if (data[i]>data[i+1])
                {
                    int temp;
                    temp=data[i];
                    data[i]=data[i+1];
                    data[i+1]=temp;
                    sorted = FALSE;
                }
    } while (!sorted);

   printf("Data after sort \n");
   printf("=============== \n");
   for (i=0;i<n;i++)
        printf("%d: %d\n",i+1,data[i]);
}
```

Introduction To programming Using
C language
Lesson -26

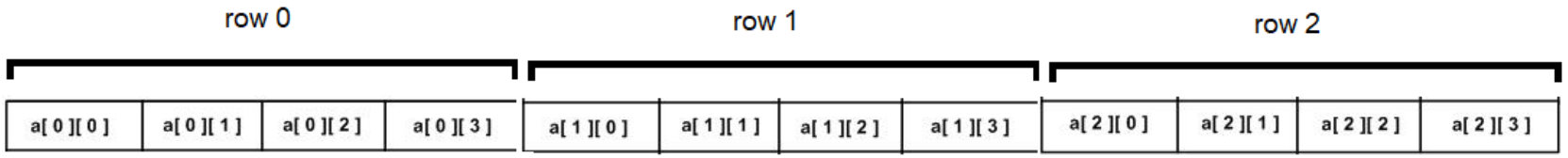
Two dimension Arrays

Two dimensional Arrays

Two dimensional Arrays
The simplest form of multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size [x][y], you would write something as follows −
type arrayName [x][y];

Where type can be any valid C data type and arrayName will be a valid C identifier. A two-dimensional array can be considered as a table which will have x number of rows and y number of columns. A two-dimensional array a, which contains three rows and four columns can be shown as follows
Int a[3][4];

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



Initializing Two-Dimensional Arrays

Initializing Two-Dimensional Arrays

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {  {0, 1, 2, 3} ,  {4, 5, 6, 7} ,  {8, 9, 10, 11}};
```

The nested braces, which indicate the intended row, are optional.

The following initialization is equivalent to the previous example −

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

store the elements of two dim Array

This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

Method 1

```c
#include<stdio.h>
void main(void )
{

   int a[2][3];

   int i, j;
   for(i=0; i<2; i++)
    {
      for(j=0;j<3;j++)
      {
         printf("Enter value for a[%d,%d]:", i, j);
         scanf("%d", &a[i][j]);
      }
    }

   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++)
   {
      for(j=0;j<3;j++)
         printf("%d ", a[i][j]);
      printf("\n");
   }

}
```

Method 2

```c
#include<stdio.h>
void main(void)
{

   int a[2][3];

   int i, j;
   for(i=0; i<2; i++)
    {
      printf("Enter value for row:%2d:", i);
      for(j=0;j<3;j++)
      {
         scanf("%d", &a[i][j]);
      }
    }

   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++)
   {
      for(j=0;j<3;j++)
         printf("%d ", a[i][j]);
      printf("\n");
   }

}
```

store the elements of two dim Array

This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

Method 1

```c
#include<stdio.h>
void main(void )
{

   int a[2][3];

   int i, j;
   for(i=0; i<2; i++)
    {
      for(j=0;j<3;j++)
      {
         printf("Enter value for a[%d,%d]:", i, j);
         scanf("%d", &a[i][j]);
      }
    }

   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++)
   {
      for(j=0;j<3;j++)
         printf("%d ", a[i][j]);
      printf("\n");
   }

}
```

Method 2

```c
#include<stdio.h>
void main(void)
{

   int a[2][3];

   int i, j;
   for(i=0; i<2; i++)
    {
      printf("Enter value for row:%2d:", i);
      for(j=0;j<3;j++)
      {
         scanf("%d", &a[i][j]);
      }
    }

   printf("Two Dimensional array elements:\n");
   for(i=0; i<2; i++)
   {
      for(j=0;j<3;j++)
         printf("%d ", a[i][j]);
      printf("\n");
   }

}
```

Magic square as two dim

A magic square is a square grid filled with distinct positive integers in the range such that each cell contains a different integer and the sum of the integers in each row, column and diagonal is equal.

```c
#include <stdio.h>
void main(void)
{
    int row ,col ,value;
    int size;

    printf("please size odd:");
    scanf("%d",&size);

    int magic[size+1][size+1];

    row=1;
    col=(size+1)/2;
    value =1;
    do
    {
        magic[row][col]=value;
        if (value%size==0)
            row++;
        else
            {
            row--;
            col--;
            }
        if (row==0) row=size;
        if (col==0) col=size;
        value++;
    } while (value <= size*size);

    for (row=1;row<=size;row++)
    {
        for (col=1;col<=size;col++)
            printf("%3d",magic[row][col]);
        printf("\n");
    }
}
```

```
please size odd:5
 15  8  1 24 17
 16 14  7  5 23
 22 20 13  6  4
  3 21 19 12 10
  9  2 25 18 11

Process returned 5 (0x5)   execution time : 0.983 s
Press any key to continue.
```

Magic square checks

Syntax for passing a 2D array to a function can be difficult to remember. One important thing for passing multidimensional arrays is, first array dimension does not have to be specified. The second (and any subsequent) dimensions must be given ,or you must send first the size of dimensions

```
int checkRow (int size ,int a[size][size] , int row )
{
    int r,c;
    int sum =(size*(size*size+1))/2;
    int sum_r=0;

    for (c=0;c<size;c++)
        sum_r += a[row][c];
    return ((sum==sum_r)?1:0);
}

int checkCol(int size ,int a[size][size] , int col )
{
    int r,c;
    int sum =(size*(size*size+1))/2;
    int sum_c=0;

    for (r=0;r<size;r++)
        sum_c += a[r][col];
    return (sum==sum_c?1:0);

}
```

```
int checkDiagonal(int size ,int a[size][size],char diag)
{
    int r,c;
    int sum =(size*(size*size+1))/2;
    int sum_d=0;
    switch (diag)
    {
    case 'L':
    case 'l':
        for (r=0;r<size;r++)
            for (c=0;c<size;c++)
                if (r==c)
                        sum_d += a[r][c];
        break;
    case 'R':
    case 'r':
        for (r=0;r<size;r++)
            for (c=0;c<size;c++)
                if (r==size-(c+1))
                        sum_d += a[r][c];
        break;
    }
    return (sum==sum_d?1:0);
}
```

Magic square checks

Syntax for passing a 2D array to a function can be difficult to remember. One important thing for passing multidimensional arrays is, first array dimension does not have to be specified. The second (and any subsequent) dimensions must be given ,or you must send first the size of dimensions

```
int checkMagic(int size ,int a[size][size])
{
    int r,c;
    int magic = 1;
    for (r=0;r<size;r++)
        if (checkRow (size ,a ,r)==0)
        {
            printf("row %d is %s\n",r+1,"Not magic");
            magic=0;
        }
    for (c=0;c<size;c++)
        if (checkCol(size ,a ,c)==0)
        {
          printf("col %d is %s\n",c+1,"Not magic");
          magic=0;
        }
    if (checkDiagonal(size ,a,'l')==0)
    {
        printf("diag L is %s\n","Not magic");
        magic=0;
    }
    if (checkDiagonal(size ,a,'r')==0)
    {
        printf("diag R is %s\n","Not magic");
        magic=0;
    }
    return (magic);
}
```

```
#include <stdio.h>
void main(void)
{
    /* n(n^2+1)/2 */
    char sel;

    int a[3][3]={{1,2,3},{4,5,6},{7,8,9}};
    int b[3][3]={{6,1,8},{7,5,3},{2,9,4}};
    int c[3][3]={{6,1,8},{7,5,4},{2,9,3}};

    if (checkMagic(3 ,a))
            printf (" A Magic Square !!!!\n");
                else
            printf (" A Not a Magic Square !!\n");
    printf("============================\n");
    if (checkMagic(3 ,b))
            printf (" B Magic Square !!!!\n");
                else
            printf (" B Not a Magic Square !!\n");
    printf("============================\n");
    if (checkMagic(3 ,c))
            printf (" C Magic Square !!!!\n");
                else
            printf (" C Not a Magic Square !!\n");
    printf("============================\n");
}
```

Sparse Matrix

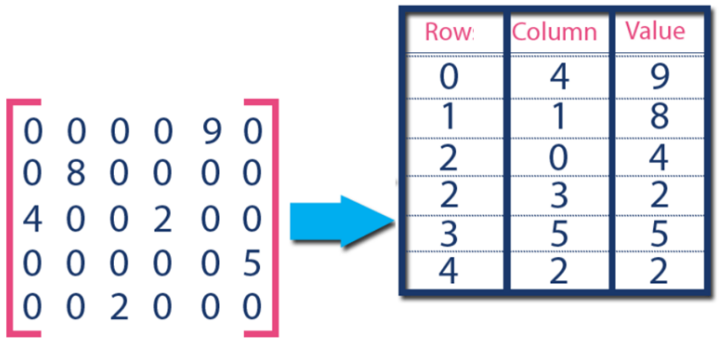
In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represent a m X n matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

When a sparse matrix is represented with a 2-dimensional array, we waste a lot of space to represent that matrix. For example, consider a matrix of size 100 X 100 containing only 10 non-zero elements. In this matrix, only 10 spaces are filled with non-zero values and remaining spaces of the matrix are filled with zero. That means, totally we allocate 100 X 100 X 4 = 40000 bytes of space to store this integer matrix. And to access these 10 non-zero elements we have to make scanning for 10000 times. To make it simple we use the following sparse matrix representation.

Sparse Matrix Representations

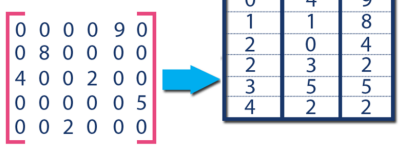
A sparse matrix can be represented by using TWO representations, those are as follows...

1. Triplet Representation (Array Representation)
2. Linked Representation



Triplet Representation (Array Representation)

In this representation, we consider only non-zero values along with their row and column index values. In this representation The first row is filled with 0, 4, & 9 which indicates the non-zero value 9 is at the 0th-row 4th column in the Sparse matrix. In the same way, the remaining non-zero values also follow a similar pattern.



Row	Column	Value
0	4	9
1	1	8
2	0	4
2	3	2
3	5	5
4	2	2

```c
#include<stdio.h>
void main(void)
{
    int index;
    int row,col;
    int matrix[5][6] =
    {
        {0 , 0 , 0 , 0 , 9, 0 },
        {0 , 8 , 0 , 0 , 0, 0 },
        {4 , 0 , 0 , 2 , 0, 0 },
        {0 , 0 , 0 , 0 , 0, 5 },
        {0 , 0 , 2 , 0 , 0, 0 }
    };
    // Finding total non-zero values in the sparse matrix
    int size = 0;
    for (int row = 0; row < 5; row++)
        for (int col = 0; col < 6; col++)
            if (matrix[row][col] != 0)
                size++;
    // Defining result Matrix
    int sparse[3][size];
    int k = 0;
    for (int row = 0; row < 5; row++)
        for (int col = 0; col < 6; col++)
            if (matrix[row][col] != 0)
            {
                sparse[0][k] = row;
                sparse[1][k] = col;
                sparse[2][k] = matrix[row][col];
                k++;
            }
```

```c
    // Displaying result matrix
    printf("Triplet Representation : \n");
    index=0;
    for ( row = 0; row < 5; row++)
    {
        for ( col = 0; col < 6; col++)
            if ((row==sparse[0][index]) && (col==sparse[1][index]))
            {
                printf("%3d",sparse[2][index]);
                index++;
            }
            else
                printf("%3d",0);
        printf("\n");
    }
    return 0;
}
```

D:\C_Lang_C\spares.exe

```
Triplet Representation :
  0  0  0  0  9  0
  0  8  0  0  0  0
  4  0  0  2  0  0
  0  0  0  0  0  5
  0  0  2  0  0  0
```

Introduction To programming Using
C language
Lesson -27

strings

String data type

String data type is not supported in C Programming. String means Collection of Characters to form particular word. String is useful whenever we accept name of the person, Address of the person, some descriptive information. We use array of type character to create String.

Strings are the form of data used in programming languages for storing and manipulating text, such as name of the person, Address of the person, some descriptive information. In C, a string is not a formal data type as it is in some languages. Instead, it is an array of type char.

Declaring String Or Character Array :

char String_Variable_name [SIZE] ;

String variable with laterals initialization

Each character occupies one byte of memory, and the last character of the string is the character '\0'. What character is that? It looks like two characters, but it's actually an escape sequence, like '\n'. It's called the "null character," and it stands for a character with a value of 0 (zero).

```c
#include<stdio.h>
void main(void)
  {

    char  name1[] = "youssef";
    char  name2[] ={'a','l','i'}; /* wrong way */

    printf("%s\n",name1);
    printf("%s\n",name2);

  }
```

```
youssef
aliyoussef
```

```c
#include<stdio.h>
void main(void)
  {

    char  name1[] = "youssef";
    char  name2[] ={'a','l','i','\0'}; /* correct way */

    printf("%s\n",name1);
    printf("%s\n",name2);

  }
```

```
youssef
ali
```

String variables

We've looked at a string constant, now let's see what a string variable looks like. Here's an example program that reads in a string from the keyboard, using scanf(), and prints it out as part of a longer phrase:

```c
#include<stdio.h>
void main(void)
  {

    char  name[5] ;


    printf("please your name:");
    scanf("%s",name);
    printf("Hello %s\n",name);

  }
```

```
please your name:youssef
Hello youssef
```

```c
#include<stdio.h>
void main(void)
  {

    char  name[20] ;


    printf("please your name:");
    scanf("%s",name);
    printf("Hello %s\n",name);

  }
```

```
please your name:youssef shawky
Hello youssef
```

You may have noticed something odd about the scanf() statement in the program. That's right; there's no address operator (&) preceding the name of the string we're going to print: scanf("%s ", name); This is because name is an address. We need to preface numerical and character variables with the & to change values into addresses, but name is already the name of an array, and therefore it's an address and does not need the &.

The String I/O Functions gets() And puts()

There are many C functions whose purpose is to manipulate strings. One of the most common is used to input a string from the keyboard. Why is it needed? Because our old friend, the scanf() function, has some limitations when it comes to handling strings.

```c
#include<stdio.h>
void main(void)
  {

    char  name[5] ;


    printf("please your name:");
    scanf("%s",name);
    printf("Hello %s\n",name);

  }
```

```
please your name:youssef
Hello youssef
```

```c
#include<stdio.h>
void main(void)
  {

    char  name[20] ;


    printf("please your name:");
    scanf("%s",name);
    printf("Hello %s\n",name);

  }
```

```
please your name:youssef shawky
Hello youssef
```

```c
#include<stdio.h>
void main(void)
  {

    char  name[20] ;

    printf("please your name:");
    gets(name);
    printf("Hello ");
    puts(name);

  }
```

```
please your name:youssef shawky
Hello youssef shawky
```

You may have noticed something odd about the scanf() statement in the program. That's right; there's no address operator (&) preceding the name of the string we're going to print: scanf("%s ", name); This is because name is an address. We need to preface numerical and character variables with the & to change values into addresses, but name is already the name of an array, and therefore it's an address and does not need the &.

String as function parameter

There are many C functions whose purpose is to manipulate strings. One of the most common is used to input a string from the keyboard. Why is it needed? Because our old friend, the scanf() function, has some limitations when it comes to handling strings.

```c
#include<stdio.h>
int str_len(char a[]);
int Palindrome(char a[]);
void main(void)
  {
    char  name[20] ;
    printf("please your name:");
    gets(name);
    printf("you name length is %d characters\n",str_len(name));
    if (Palindrome(name))
        printf("%s Palindrome word  ",name);
    else
        printf("%s not Palindrome word  ",name);
  }
int str_len(char a[])
{
    int i=0;
    while (a[i] !='\0') i++;
    return i;
}
int Palindrome(char a[])
{
    int res =1;
    int i;
    for (i=0;i<(str_len(a)/2);i++)
    {
            if (a[i]!=a[str_len(a)-1-i])
                {
                  res = 0;
                  break;
                }
    }
    return (res);
}
```

```
please your name:madam
you name length is 5 characters
madam Palindrome word
```

Note:

- While the individual characters were surrounded by single quotes, the string is surrounded by double quotes.
- We don't need to insert the null character '\0'. Using the string format causes this to happen automatically.

String functions <string.h>

In C, which thinks of strings as arrays, there are no special operators for dealing with them. However, C does have a large set of useful string-handling library functions. the strlen() function returns the length of the string whose address is given to it as an argument.

The library string.h (also referred to as cstring) has several common functions for dealing with strings stored in arrays of characters.

strlen() : Finding Length of String

While manipulating string or character array in c programming. We need to compute the length of the string , C library provides different string handling functions. In order to compute length of the string strlen() function is used.

Explanation and Summary :

Point	Explanation
No of Parameters	1
Parameter Taken	Character Array Or String
Return Type	Integer
Description	Compute the Length of the String
Header file	string.h

```c
#include<stdio.h>
#include<string.h>
void main(void)
  {
    char str[20];
    int length ;

    printf("Enter the String : ");
    gets(str);
    length = strlen(str);
    printf("Length of String : %d ", length);
  }
```

```
Enter the String : mehrivan
Length of String : 8
```

String functions <string.h>

The library string.h (also referred to as cstring) has several common functions for dealing with strings stored in arrays of characters.

Function name	Action
int strlen(string str)	It returns the length of the string without including end character (terminating char '\0').
int strcmp(string str1, string str2)	If str1 < str2 then it would result a negative value. If str1 > str2 then return positive value.If string1 == string2 then you would return (zero)
string strcat(string str1, string str2)	It concatenates two strings and returns the concatenated string.
string strcpy(string str1, string str2)	Copy second string into First
string strstr(string str1,string str2)	finds the first occurrence of the substring str2 in the string str1. This returns a string to the first occurrence of the string str2 in the string str1, or a null pointer if the sequence is not present in str1.
string strtok(string str, string delim)	The contents of this string are modified and broken into smaller strings (tokens)
string strupr(string str)	function converts a given string into uppercase. "MODIFY This String To Upper" is converted into upper case using strupr( ) function
string strlwr(string str)	function converts a given string into Lowercase."MODIFY This String To LOwer" is converted into lower case using strlwr( ) function

String functions Examples

In C, which thinks of strings as arrays, there are no special operators for dealing with them. However, C does have a large set of useful string-handling library functions. the strlen() function returns the length of the string whose address is given to it as an argument.

The library string.h (also referred to as cstring) has several common functions for dealing with strings stored in arrays of characters. This a very useful Example.

```c
#include<stdio.h>
#include<string.h>
void main(void)
  {
    char str1[20]="hello koko wawa";
    char str2[20]="koko";
    char str3[30];


    /* strlen() */
    printf("str1 = %s and it has a length= %d\n",str1,strlen(str1));
    printf("str2 = %s and it has a length= %d\n",str2,strlen(str2));
    printf("str3 = %s and it has a length= %d\n",str3,strlen(str3));
    /* strcmp() */
    if (strcmp(str1,str2) >0)
        printf("str1=%s > str2=%s\n",str1,str2 );
    else if (strcmp(str1,str2) <0)
        printf("str1=%s < str2=%s\n",str1,str2 );
    else
         printf("str1=%s = str2=%s\n",str1,str2 );
    /* strcat */
    printf("str1+str2 = %s\n",strcat(str1,str2));
    printf("str1 after strcat = %s\n",str1);
    printf("str2 after strcat = %s\n",str2);
    /* strcpy */
    strcpy(str1,"koko");
    strcpy(str2,"wawa");
    strcpy(str3,str1);
    strcpy(str3,strcat(str3,str2));

    printf("str1 = %s \n",str1);
    printf("str2 = %s \n",str2);
    printf("str3 = %s \n",str3);

  }
```

```
str1 = hello koko wawa and it has a length= 15
str2 = koko and it has a length= 4
str3 = @ and it has a length= 1
str1=hello koko wawa < str2=koko
str1+str2 = hello koko wawakoko
str1 after strcat = hello koko wawakoko
str2 after strcat = koko
str1 = koko
str2 = wawa
str3 = kokowawa
```

Sorting array of strings

In C, which thinks of strings as arrays, there are no special operators for dealing with them. However, C does have a large set of useful string-handling library functions. the strlen() function returns the length of the string whose address is given to it as an argument.

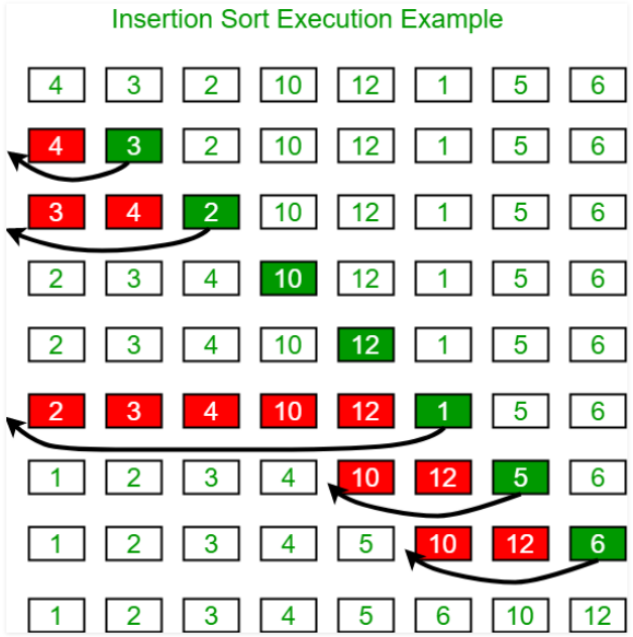
The library string.h (also referred to as cstring) has several common functions for dealing with strings stored in arrays of characters. This a very useful Example.

```c
#include<stdio.h>
#include<string.h>
void  main(void)
{
   int i,j,count;
   char names[100][25],temp[25];
   printf("How many strings u are going to enter?: ");
   scanf("%d",&count);
   fflush(stdin);
   puts("Enter Strings one by one: ");
   for(i=0;i<count;i++)
   {
        printf("name[%d]:",i+1);
        gets(names[i]);
   }

   for(i=0;i<count-1;i++)
      for(j=i+1;j<count;j++){
         if(strcmp(names[i],names[j])>0)
         {
            strcpy(temp,names[i]);
            strcpy(names[i],names[j]);
            strcpy(names[j],temp);
         }
      }
   printf("Order of Sorted Strings:\n");
   for(i=0;i<count;i++)
      puts(names[i]);

}
```

```
How many strings u are going to enter?: 6
Enter Strings one by one:
name[1]:koko
name[2]:lolo
name[3]:soso
name[4]:fofo
name[5]:toto
name[6]:ali
Order of Sorted Strings:
ali
fofo
koko
lolo
soso
toto
```

insertion sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.




```c
#include<stdio.h>
#include<string.h>
void  main(void)
{
   int i,j,count;
   char names[100][25],temp[25];
   printf("How many strings u are going to enter?: ");
   scanf("%d",&count);
   fflush(stdin);
   puts("Enter Strings one by one: ");
   for(i=0;i<count;i++)
   {
        printf("name[%d]:",i+1);
        gets(names[i]);
   }
   insertionSort(names,count);
   printf("Order of Sorted Strings:\n");
   for(i=0;i<count;i++)
      puts(names[i]);
}

void insertionSort(char arr[][25], int n)
{
    int i, j;
    char key[25];
    for (i = 1; i < n; i++)
    {
        strcpy(key,arr[i]);
        j = i - 1;
        while (j >= 0 && strcmp(arr[j],key)>0)
        {
            strcpy(arr[j + 1],arr[j]);
            j = j - 1;
        }
        strcpy(arr[j + 1],key);
    }
}
```

```
How many strings u are going to enter?: 6
Enter Strings one by one:
name[1]:koko
name[2]:ali
name[3]:mona
name[4]:dalia
name[5]:youssef
name[6]:mohab
Order of Sorted Strings:
ali
dalia
koko
mohab
mona
youssef
```

Introduction To programming Using
C language
Lesson -28

pointers

address of the variable - &

Pointers are regarded by most people as one of the most difficult topics in C. There are several reasons for this.
First, the concept behind pointers indirection may be a new one for many programmers.
And second, the symbols used for pointer notation in C are not as clear as they might be.
In other words, pointers may be difficult, but they aren't too difficult.

In a similar way, a program statement can refer to a variable indirectly, using the address of the variable as a sort of post office box.

```c
#include<stdio.h>
void main(void)
{
    int x = 10;
    printf("Value of x is : %d\n",x);
    printf("Value of &x or address of x is : %u\n",&x);
    printf("Value of &x or address of x is : %p\n",&x);
}
```

```
Value of x is : 10
Value of &x or address of x is : 6356732
Value of &x or address of x is : 0060FEFC
```

Pointers

A pointer provides a way of accessing a variable without referring to the variable directly. The mechanism used for this is the address of the variable. In effect, the address acts as an intermediary between the variable and the program accessing it.

Pointers:
in **C** language is a variable that stores/points the address of another variable.

```
int     *ptr;
int *    ptr;
```

```c
#include<stdio.h>
void main(void)
{
    int x = 10;
    int *ptr;

    ptr = &x;

        printf("Value of x      : %d\n",x);
        printf("Address of x    : %u\n",&x);
        printf("Value of ptr is : %u\n",ptr);
        printf("Value that ptr pointing to it  : %d\n",*ptr);
}
```

```
Value of x      : 10
Address of x    : 6356728
Value of ptr is : 6356728
Value that ptr pointing to it  : 10
```

Pointers data type

Important point to note is:

The data type of pointer and the variable must match, an int pointer can hold the address of int variable, similarly a pointer declared with float data type can hold the address of a float variable

```c
#include<stdio.h>
void main(void)
{
    int x = 10;
    float y=23.5f;
    int *   ptr_i;
    float * ptr_f;

    ptr_i = &x;
    ptr_f = &y;

    printf("size of pointer to x is %d bytes\n",sizeof(ptr_i));
    printf("size of pointer to y is %d bytes\n",sizeof(ptr_f));


}
```

```
size of pointer to x is 4 bytes
size of pointer to y is 4 bytes
```

Dereferencing Pointer operator (*)

When used with Pointer variable , it refers to variable being pointed to, this is called as Dereferencing of Pointers. Dereferencing Operation is performed to access or manipulate data contained in memory location pointed to by a pointer

Any Operation performed on the de-referenced pointer directly affects the value of variable it pointes to.

```c
#include<stdio.h>
void main(void)
{
    int x = 10;
    int * ptr;

    printf("Value of x is : %d\n",x);

    ptr = &x;
    *ptr = 20;

    printf("Value of x now is : %d\n",x);
}
```

```
Value of x is : 10
Value of x now is : 20
```

void Pointer

Void Pointer is special type of pointer which can store the address of any variable. It's a special type of pointer called void pointer or general purpose pointer.
Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables. Instead it is feasible to declare single pointer variable which can act as integer pointer, character and float pointer.

```c
#include<stdio.h>
void main(void)
{



int inum = 8;
float fnum = 67.7;
void *ptr;

ptr = &inum;
printf("\nThe value of inum = %d",*((int*)ptr));

ptr = &fnum;
printf("\nThe value of fnum = %f",*((float*)ptr));


}
```

Type of Address Stored in Void Pointer	Dereferencing Void Pointer
Integer	*((int*)ptr)
Charcter	*((char*)ptr)
Floating	*((float*)ptr)

```
The value of inum = 8
The value of fnum = 67.699997
```

pointer arithmetic

We can perform arithmetic operations on the pointers like addition, subtraction, etc. However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.

In pointer-from-pointer subtraction, the result will be an integer value. Following arithmetic operations are possible on the pointer in C language:

- Increment
- Decrement
- Addition
- Subtraction
- Comparison

Expression	**Result**
Address + Number	Address
Address – Number	Address
Address – Address	Number
Address + Address	Illegal

Incrementing Pointer

We can perform Incrementing Pointer :

1.Incrementing Pointer is generally used in array because we have contiguous memory in array and we know the contents of next memory location.

2.Incrementing Pointer Variable Depends Upon data type of the Pointer variable

3. Increment and Decrement Operations on pointer should be used when we have Continues memory (in Array).

```
new value =  current address +  size_of(data type)
```

```c
#include<stdio.h>
void main(void)
{

int inum = 8;
int  *ptr;

ptr = &inum;
printf("\nThe value of pointer ptr  = %u\n",ptr);

ptr++;
printf("\nThe value of pointer ptr  = %u\n",ptr);

}
```

```
The value of pointer ptr = 6356728

The value of pointer ptr = 6356732
```

```c
#include<stdio.h>

void main(void)
{

int arr[5]={1,2,3,4,5};
int *ptr;

ptr=&arr;
printf("Value of ptr : %u\n",ptr);

ptr++;
printf("Value of ptr : %u\n",ptr);

ptr++;
printf("Value of ptr : %u\n",ptr);
printf("Value of that ptr point now is : %d\n",(*ptr));

}
```

```
Value of ptr : 6356712
Value of ptr : 6356716
Value of ptr : 6356720
Value of that ptr point now is : 3
```

C difference between *ptr++ & ++*ptr

Are *ptr++ and ++*ptr are same ?
Ans : Absolutely Not

ptr++ means:
Increment the Pointer not Value Pointed by It

```c
#include<stdio.h>

void main(void)
{

    int arr[5]={1,2,3,4,5};
    int *ptr;
    int i;

    ptr=&arr;
    i=0;
    while (i <5)
    {
      printf("Value of ptr : %d\n",*ptr++);
      i++;
    }

    for (i=0;i<5;i++)
        printf("%d ",arr[i]);

}
```

```
Value of ptr : 1
Value of ptr : 2
Value of ptr : 3
Value of ptr : 4
Value of ptr : 5
1 2 3 4 5
```

++*ptr means
Increment the Value being Pointed to by ptr

```c
#include<stdio.h>

void main(void)
{

    int arr[5]={1,2,3,4,5};
    int *ptr;
    int i;

    ptr=&arr;
    i=0;
    while (i <5)
    {
      printf("Value of ptr : %d\n",++*ptr);
      i++;
    }

    for (i=0;i<5;i++)
        printf("%d ",arr[i]);

}
```

```
Value of ptr : 2
Value of ptr : 3
Value of ptr : 4
Value of ptr : 5
Value of ptr : 6
6 2 3 4 5
```

Double Pointer (Pointer to Pointer)

Double Pointer (Pointer to Pointer) in C. A pointer is used to store the address of variables. So, when we define a pointer to pointer, the first pointer is used to store the address of the second pointer

*int **ptr; /* declaring double pointers */*

```c
#include <stdio.h>
/* C program to demonstrate pointer to pointer*/
void main(void)
{
    int var = 123;
    int *ptr2;
    int **ptr1;

    ptr2 = &var;
    ptr1 = &ptr2;

    printf("Value of var = %d\n", var );
    printf("Value of var using single pointer = %d\n", *ptr2 );
    printf("Value of var using double pointer = %d\n", **ptr1);

}
```

```
Value of var = 123
Value of var using single pointer = 123
Value of var using double pointer = 123

Process returned 40 (0x28)   execution time : 0.150 s
Press any key to continue.
```

NULL Pointer

We have already studied the different type of pointer in C. In this tutorial we will be learning about NULL Pointer in C Programming

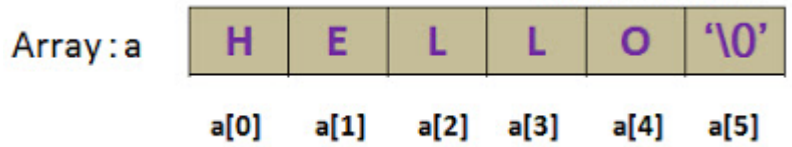
```
int  *ptr = NULL;
```

Difference between char *a and char a[]

We char * and char [] both are used to access character array, Though functionally both are same , they are syntactically different. See how both works in order to access string.

char a[] = "HELLO";

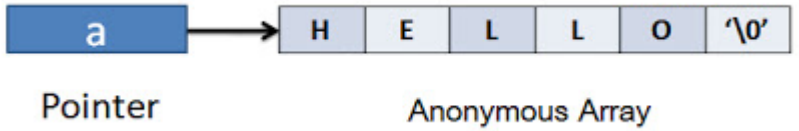
In the example – String “Hello” is stored in Character Array ‘a’. Character array is used to store characters in Contiguous Memory Location. It will take Following Form after Initialization. We have not specified Array Size in this example.
Each array location will get following values :




```
a[0] = 'H'
a[1] = 'E'
a[2] = 'L'
a[3] = 'L'
a[4] = 'O'
a[5] = '\0'
```

char *a = "HELLO";

String “Hello” will be stored at any Anonymous location in the form of array. We even don’t know the location where we have stored string, However String will have its starting address.



Constant Pointers

A constant pointer in C cannot change the address of the variable to which it is pointing, i.e., the address will remain constant. Therefore, we can say that if a constant pointer is pointing to some variable, then it cannot point to any other variable.

Syntax Declaration :

```
data_type * const Pointer_name = Initial_Address;
```

Sample Code Snippet :

```
int num = 20;
 int * const ptr = &num ; // Declare Constant Pointer

 *ptr = 20 ;              // Valid Statement
 ptr ++ ;                 // Invalid Statement
```

Pointer to Constant

pointer to constant in C a pointer through which one cannot change the value of variable it points is known as a pointer to constant. These type of pointers can change the address they point to but cannot change the value kept at those address.

```c
#include<stdio.h>
void main(void)
{
    int var1 = 100;
    int var2 = 200;
    /* const data_type * pointer_name = initial_address */
    const int* ptr = &var1;

    printf("%d\n", *ptr);
    ptr = &var2;
    printf("%d\n", *ptr);
    return 0;
}
```

Pointer to function

Function pointer : A pointer which keeps address of a function is known as function pointer or pointer to function.

We should follow following 3 steps to use pointer to call function :
1.Declare Pointer which is capable of storing address of function. **Return_type *(*ptr)(parameter type)**
2.Initialize Pointer Variable
3.Call function using Pointer Variable.

```c
#include<stdio.h>
void display(int n);

void main(void)
{
     void (*ptr)(int);

     ptr = &display; /* or  ptr=display */
     (*ptr)(3);      /* or  ptr(3)      */
}

void display(int n)
{
    int i;
    for (i=1;i<=n;i++)
        printf("Hello Youssef\n");
}
```

Pointer to function

In C, we can use function pointers to avoid code redundancy.

```c
#include<stdio.h>
int add(int ,int );
int subtrct(int ,int );
int mutipl(int ,int );
int div(int ,int );

void main(void)
{
      int num1,num2,op;

      int (*calc[4])(int,int)={add,subtrct,mutipl,div};

      printf("First Number:");
      scanf("%d",&num1);
      printf("Second Number:");
      scanf("%d",&num2);
      printf("select your operation : 1-addition ,2-subtraction ,3-multiplication,4-division : ");
      scanf("%d",&op);
      printf("result = %d\n",calc[op-1](num1,num2));

}

int add(int a,int b) { return (a+b);}
int subtrct(int a,int b) { return (a-b);}
int mutipl(int a,int b) { return (a*b);}
int div(int a,int b) { return (a/b);}
```

Pointer to array element

Pointer and array , both are closely related and We know that array name without subscript points to first element in an array so

int x[10];

Here x , x[0] both are identical

1.x is an array ,
2.i is subscript variable

x [i] = * (x + i)
= * (i + x)
= i [x]

```c
#include<stdio.h>

void main(void)
  {

    int x[]={100,200,300,400,500};

    printf("%d\n",x[3]);
    printf("%d\n",*(x+3));
    printf("%d\n",*(3+x));
    printf("%d\n",3[x]);

  }
```

```
400
400
400
400
```

Pointer to Character = string

Pointer can also be used to create strings.
Pointer variables of char type are treated as string.

char * name = “youssef";

printf(“%s\n”,name);

```c
#include<stdio.h>
void main(void)
  {

    char * name = "youssef";

    printf("%s\n",name);
    while (*name !='\0')
    {
            printf("%c\n",*name++);

    }
  }
```

```
youssef
y
o
u
s
s
e
f
```

Pointer to array of string

A pointer which pointing to an array which content is string, is known as pointer to array of strings.

char *arr[4] = {"C","C++","Java",“Python"};
char *(*ptr)[4] = &arr;

```c
#include<stdio.h>
void main(void)
{
int i;

char *arr[4] = {"C","C++","Java","VBA"};
char *(*ptr)[4] = &arr;

for(i=0;i<4;i++)
    printf("Address of String %d : %u  %s\n",i+1,(*ptr)[i],(*ptr)[i]);


}
```

```
Address of String 1 : 4206628  C
Address of String 2 : 4206630  C++
Address of String 3 : 4206634  Java
Address of String 4 : 4206639  VBA
```

Pointer usages

Pointer Use
Some reasons to use pointers are:

1. passing address to function (call by references)
2. Return more than one value from a function
3. Pass arrays and strings more conveniently from one function to another
4. Manipulate arrays more easily by moving pointers them (or to parts of them),instead of moving the arrays themselves
5. Create complex data structures, such as linked lists and binary trees, where one data structure must contain references to other data structures

passing address to function (call by references)

The call by reference method of passing arguments to a function copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. It means the changes made to the parameter affect the passed argument.

```c
#include<stdio.h>
void main(void)
{

    int x =10;
    int y =20;

    printf("x=%d and y=%d\n",x,y);
    swap_1(x,y);
    printf("x=%d and y=%d\n",x,y);
    swap_2(&x,&y);
    printf("x=%d and y=%d\n",x,y);


}

void swap_1(int a,int b)
    {

        int temp;

        temp=a;
        a=b;
        b=temp;
    }

void swap_2(int * a,int * b)
    {

        int temp;

        temp=*a;
        *a=*b;
        *b=temp;
    }
```

```
x=10 and y=20
x=10 and y=20
x=20 and y=10
```

Return more than one value from a function

We can return more than one values from a function by using the method called “call by address”, or “call by reference”. In the invoker function, we will use two variables to store the results, and the function will take pointer type data. So we have to pass the address of the data.
In this example, we will see how to define a function that can return quotient and remainder after dividing two numbers from one single function.

```c
#include<stdio.h>

void main(void)
{
   int a = 76, b = 10;
   int q, r;
   div(a, b, &q, &r);
   printf("Quotient is: %d\nRemainder is: %d\n", q, r);
}

void div(int a, int b, int *quotient, int *remainder)
{
   *quotient = a / b;
   *remainder = a % b;
}
```

```
Quotient is: 7
Remainder is: 6
```

Pass arrays and strings more conveniently return array as pointer

To pass an entire array to a function, only the name of the array is passed as an argument.

To pass multidimensional arrays to a function, only the name of the array is passed to the function(similar to one-dimensional arrays).

In C programming, you can pass arrays to functions, however, you cannot return arrays from functions.

```c
#include<stdio.h>
int * sum(int a[3], int b[3]);
void main(void)
{
   int a[3]={1,2,3};
   int b[3]={4,5,6};
   int * c;
   int i;


   c= sum(a,b);
   for (i=0;i<3;i++)
        printf("%d \t ",*(c+i));
}

int * sum(int a[3], int b[3])
{
   int i;
   static int c[3];
   for (i=0;i<3;i++)
        c[i]=a[i]+b[i];
   return (c);
}
```

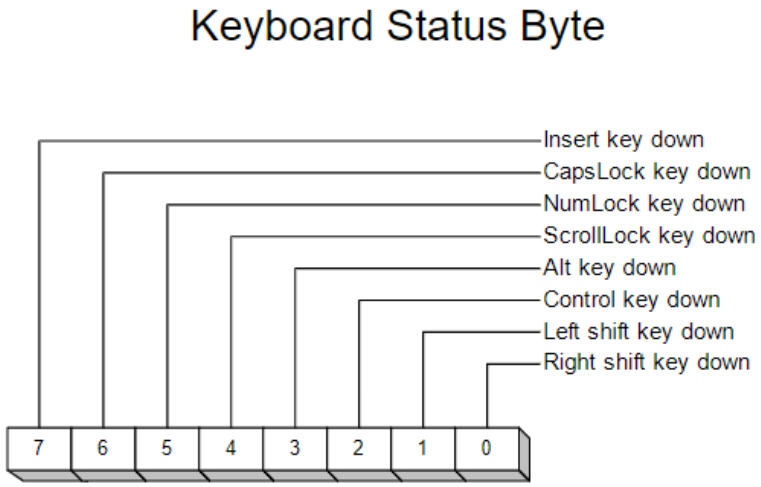
```
5        7        9
Process returned 4 (0x4)   execution time : 1.596 s
Press any key to continue.
```

Manipulate arrays more efficient by moving pointers

Pointers can be very useful when working with arrays. We can use them with existing arrays. Array notation and pointer notation can be used somewhat interchangeably.
adding an integer to a pointer will increment the address it holds by the product of the integer and the data type's size.

```c
#include <stdio.h>
void  main(void)
{
    int data[500]={ 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
                    1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};


    int sum =0;
    printf("You Data: \n");
    for (int i = 0; i < 500; ++i)
    {
        //printf("%d\t",data[i]);
        //sum = sum +data[i];
        printf("%d\t",*(data + i));
        sum = sum +*(data + i);

    }
    printf("\n sum = %d\n",sum);

}
```

Direct Address memory location (1)




```
void main(void)
{

        char far * ptr;

        _

        ptr=( char far *) 0x00400017;
        *ptr=(char) 0x03;
}
```

Direct Address memory location (2)

```
#include <stdio.h>
#include <conio.h>
#define MAXCOL  80
#define MAXROW  25
void main(void)
{
        int row,col;
        int far * farptr;
        char ch;

        farptr=(int far *) 0xB8000000;
        while ((ch=getche())!=27)
        {
        for (row=0;row<MAXROW;row++)
                for (col=0;col<MAXCOL;col++)
                        *(farptr+row*MAXCOL+col)=(int) ch | 0x700;
        }
}
```

Introduction To programming Using
C language
Lesson -29

structures

Structures

We have seen how simple variables can hold one piece of information at a time and how arrays can hold a number of pieces of information of the same data type. These two data storage mechanisms can handle a great variety of situations. But we often want to operate on data items of different types together as a unit. In this case, neither the variable nor the array is adequate.

For example, suppose you want a program to store data concerning an employee in an organization. You might want to store the employee's name (a character array), department number (an integer), salary (a floating point number), and so forth.

To solve this sort of problem, C provides a special data type: the structure. A structure consists of a number of data items which need not be of the same type grouped together. In our example, a structure would consist of the employee's name, department number, salary, and any other pertinent information.

Declaring A Structure Type

Since a structure may, contain any number of elements of different types, the programmer must tell the compiler what a particular structure is going to look like before using variables of that type.

In the ***example*** program, the following statement declares the structure type:

```
struct easy
        {
            int num;
            char ch;
        };
```

This statement declares a new data type called struct easy. Each variable of this type will consist of two members: an integer variable called num, and a character variable called ch.
That this statement doesn't define any variables, and so it isn't setting aside any storage in memory.

Defining Structure Variables

Once we've declared our new data type, we can define one or more variables to be of that type. In our program, we define a variable ez1 to be of type Struct easy:

struct easy ez1;

This statement does set aside space in memory. It establishes enough space to hold all the items in the structure: in this case, five bytes: 4 for the integer, and one for the character.

Accessing Structure Members

Now how do we refer to individual members of the structure?

Structures use a different approach: the "dot operator" (.), which is also called the "membership operator." Here's how we would refer to the num part of the ez1 structure:

ez1.num The variable name preceding the dot is the structure name; the name following it is the specific member of the structure.
Thus the statements

```
ez1.num = 2;
ez1.ch = 'z ';
```

Combining Declarations

You can combine in one statement the declaration of the structure type.
and the definition of structure variables.
As an example.

```
void main(void)
{
 struct easy
    {
      int num;
      char ch;
    } ez1, ez2;

  ez1.num = 2;
  ez1.ch = 'z';
  ez2.num = 3;
  ez2.ch = 'y';
  printf("ez1.num=%d, ez1.ch=%c\n", ez1.num, ez1.ch);
  printf("ez2.num=%d, ez2.ch=%c\n", ez2.num, ez2.ch );
 }
```

Initializing Structures

Like simple variable and arrays, structure variables can be initialized given specific values at the beginning of a program. The format used is quite similar to that used to initialize arrays.

```
struct person
{
    int code;
   char name [30];
   float salary;

};

struct person person1 = { 100,"Youssef Shawky", 500.50 };
struct person person2 = { 105,"Mohab youssef", 600.50 };

void main(void)
{
   printf("\nList of agents:\n");
   printf("--------------------\n");
   printf(" code: %3d\n", person1.code);
   printf(" Name: %s\n", person1.name);
   printf(" salary: %7.2f\n", person1.salary);
   printf(" code: %3d\n", person2.code);
   printf(" Name: %s\n", person2.name);
   printf(" salary: %7.2f\n", person2.salary);

}
```

```
List of agents:
--------------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50
 code: 105
 Name: Mohab youssef
 salary:  600.50

Process returned 17 (0x11)   execution time : 0.045 s
Press any key to continue.
```

Assignment Statements with Structures

Assignment Statements used with Structures In the original version of C defined by Kernighan and Ritchie, it was impossible to assign the values of one structure variable to another variable using a simple assignment statement. In ANSI-compatible version of C.

That is, if person1 and person2 are structure variables of the same type, the following statement can be used:

person2=person1;

```
struct person
{
    int code;
   char name [30];
   float salary;
};
struct person person1 = { 100,"Youssef Shawky", 500.50 };
struct person person2 = { 105,"Mohab youssef", 600.50 };
struct person person3;
void main(void)
{
   printf("\nList of agents:\n");
   printf("--------------------\n");
   printf(" code: %3d\n", person1.code);
   printf(" Name: %s\n", person1.name);
   printf(" salary: %7.2f\n", person1.salary);
   printf(" code: %3d\n", person2.code);
   printf(" Name: %s\n", person2.name);
   printf(" salary: %7.2f\n", person2.salary);

   person3=person1;  /* assignment  structure */

   printf("\n-----------------\n");
   printf(" code: %3d\n", person3.code);
   printf(" Name: %s\n", person3.name);
   printf(" salary: %7.2f\n", person3.salary);
}
```

```
List of agents:
--------------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50
 code: 105
 Name: Mohab youssef
 salary:  600.50

-----------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50
```

Pointer to Structures

Just as pointers can be used to contain the addresses of simple variables and arrays, they can also be used to hold the addresses of structures. Here's an example that demonstrates a pointer pointing to a structure:

person * ptr;

ptr = & person1;

ptr->code =230;

```
typedef struct xx
{
   int code;
  char name [30];
  float salary;
} person
person person1 = { 100,"Youssef Shawky", 500.50 };
person person2 = { 105,"Mohab youssef", 600.50 };
person * ptr;
void main(void)
{
  ptr = &person1;
  printf("\nList of agents:\n");
  printf("--------------------\n");
  printf(" code: %3d\n", ptr->code);
  printf(" Name: %s\n", ptr->name);
  printf(" salary: %7.2f\n", ptr->salary);

  ptr = &person2;
  printf("\nList of agents:\n");
  printf("--------------------\n");
  printf(" code: %3d\n", ptr->code);
  printf(" Name: %s\n", ptr->name);
  printf(" salary: %7.2f\n", ptr->salary);

  ptr->code =201;
  strcpy(ptr->name,"koko wawa");

  printf("\n----------------\n");
  printf(" code: %3d\n", person2.code);
  printf(" Name: %s\n", person2.name);
  printf(" salary: %7.2f\n", person2.salary);
}
```

```
List of agents:
--------------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50

List of agents:
--------------------
 code: 105
 Name: Mohab youssef
 salary:  600.50

----------------
 code: 201
 Name: koko wawa
 salary:  600.50
```

Passing Structures To Functions

Passing Structures To Functions In the same way that it can be passed in an assignment statement, the value of a structure variable can also be passed as a parameter to a function.

A structure can be passed to any function from main function or from any sub function by value .

```
typedef struct xx
{
   int code;
   char name [30];
   float salary;
} person ;
 person person1 = { 100,"Youssef Shawky", 500.50 };
 person person2 = { 105,"Mohab youssef", 600.50 };
void print_s(person p);
void increment_code(person p);
void main(void)
{

   printf("\nList of agents:\n");
   print_s(person1);
    print_s(person2);

    increment_code(person1);
    print_s(person1);
    cor_increment_code(&person1);
    print_s(person1);
}
void print_s(person p)
{
   printf("\n----------------\n");
   printf(" code: %3d\n", p.code);
   printf(" Name: %s\n", p.name);
   printf(" salary: %7.2f\n", p.salary);

}
void increment_code(person p)
{
   p.code++;
}
void cor_increment_code(person *p)
{
   p->code++;
}
```

```
List of agents:

----------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50

----------------
 code: 105
 Name: Mohab youssef
 salary:  600.50

----------------
 code: 100
 Name: Youssef Shawky
 salary:  500.50

----------------
 code: 101
 Name: Youssef Shawky
 salary:  500.50

Process returned 17 (0x11)   execution time : 1.628 s
Press any key to continue.
```

return Structure from Function

return Structure from Function is an efficient way to receive a struct after an operation .

```c
#include<stdio.h>
typedef struct xx
{
    int code;
    char name [30];
    float salary;
} person ;
person get_data(void);
void print_s(person p);

void main(void)
{
    person person1,person2;
    printf("\nList of agents:\n");
    person1= get_data();
    person2= get_data();
    print_s(person1);
    print_s(person2);
}
person get_data(void)
{
    person temp;
    printf("please code:");
    scanf("%d",&temp.code);
    fflush(stdin);
    printf("please name:");
    gets(temp.name);
    printf("please salary:");
    scanf("%f",&temp.salary);
    return temp;
}
void print_s(person p)
{
    printf("\n-----------------\n");
    printf(" code: %3d\n", p.code);
    printf(" Name: %s\n", p.name);
    printf(" salary: %7.2f\n", p.salary);

}
```

```
List of agents:
please code:100
please name:youssef shawky
please salary:500
please code:102
please name:mohab youssef
please salary:456

-----------------
 code: 100
 Name: youssef shawky
 salary:  500.00

-----------------
 code: 102
 Name: mohab youssef
 salary:  456.00

Process returned 17 (0x11)   execution time : 41.875 s
Press any key to continue.
```

Arrays Of Structures

We now know enough to realize our goal of creating an array of structures, where each structure represents the data for one Employee.

```
please number of employee < 100: 3
Employee no[1]
        Code: 201
        Name: youssef
        Salary: 345
Employee no[2]
        Code: 102
        Name: mohab
        Salary: 345
Employee no[3]
        Code: 101
        Name: ali
        Salary: 324

N0              Code            Name            Salary
--------------------------------------------------------------------------------
1               101             ali             324.00
2               102             mohab           345.00
3               201             youssef         345.00

Process returned 3 (0x3)   execution time : 39.669 s
Press any key to continue.
```

```c
#include <stdio.h>
struct employee
{
int code;
char name[30];
float salary;
};
void swap(struct employee *,struct employee *);
void main(void)
{
    struct employee x[100];
    int size, i,j;


    system("cls");
    printf("please number of employee < 100: ");
    scanf("%d", &size);
    fflush(stdin) ;
    for (i=0; i < size; i++)
    {
        printf("Employee no[%d]\n",i+ 1);
        printf("\tCode: ");
        scanf("%d", &x[i].code);
        fflush(stdin) ;
        printf("\tName: ");
        gets(x[i].name);
        fflush(stdin);
        printf("\tSalary: ");
        scanf("%f", &x[i].salary);
    }

    for (i=0 ; i < size-1; i++)
        for (j=i+1 ;j<size;j++)
            if(x[i].code > x[j].code)
                swap (&x[i],&x[j]);
                    printf("\nN0 \t\tCode \t\tName \t\tSalary\n");
                    printf("--------------------------------------------------------------------------------\n");
    for (i=0;i<size;i++)
            printf("%d\t\t%d\t\t%s\t\t%.2f\n", i+ 1 ,x[i]. code,x[i]. name,x[i].salary);
}
void swap(struct employee *a,struct employee *b)
{
struct employee temp;
temp = * a;
*a = * b;
*b = temp;
}
```

Dynamic Memory Allocation

C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:
1.malloc()
2.calloc()
3.free()
4.realloc()

malloc() method

“malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

ptr = (castType*) malloc(size);

free() method

The free() function in C deallocates a block of memory previously allocated using calloc, malloc or realloc functions, making it available for further allocations and return it to heap so that it can be used for other purposes..

free(ptr);

Dynamic Memory Allocation

C Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming. They are:
1.malloc()
2.calloc()
3.free()
4.realloc()

malloc() method

"malloc" or "memory allocation" method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initializes each block with default garbage value.

ptr = (castType*) malloc(size);

calloc() method

The malloc() function allocates memory and leaves the memory uninitialized. Whereas, the calloc() function allocates memory and initializes all bits to zero.
calloc() function is used to allocate multiple continuous blocks of memory.

ptr = (cast_type *) calloc (n, size);

free() method

The free() function in C deallocates a block of memory previously allocated using calloc, malloc or realloc functions, making it available for further allocations and return it to heap so that it can be used for other purposes..

free(ptr);

realloc() method

the realloc function is used to resize a block of memory that was previously allocated. The realloc function allocates a block of memory (which be can make it larger or smaller in size than the original) and copies the contents of the old block to the new block of memory, if necessary.

Ptr=realloc(ptr, size);

Arrays Of pointers to Structure

Before we can understand the program that will create and manipulate our array of pointers, we need to explore some of the building blocks used in its operation. There are two new ideas: how structure elements can be accessed using pointers, and how an area of memory can be assigned to a variable using pointers and the malloc()function.

1.Using the Array of Pointer the time required to access structure reduces.
2.This is used in Dynamic Memory Allocation.
3.Efficient use of memory.

```
please number of employee < 100: 3
Employee no[1]
        Code: 201
        Name: youssef
        Salary: 345
Employee no[2]
        Code: 102
        Name: mohab
        Salary: 345
Employee no[3]
        Code: 101
        Name: ali
        Salary: 324

N0              Code            Name            Salary
--------------------------------------------------------------------------
1               101             ali             324.00
2               102             mohab           345.00
3               201             youssef         345.00

Process returned 3 (0x3)   execution time : 39.669 s
Press any key to continue.
```

```c
#include <stdio.h>
#include<stdlib.h>
struct employee
{
int code;
char name[30];
float salary;
};
void swap(struct employee *,struct employee *);
void main(void)
{
    struct employee * p[100];
    int size, i,j;

    system("cls");
    printf("please number of employee < 100: ");
    scanf("%d", &size);
    fflush(stdin) ;
    for (i=0; i < size; i++)
    {

        p[i]=(struct employee *) malloc(sizeof(struct employee));
        if (!p[i])
            {
            printf("\nout ofmemory\n");
            exit(1);
            }
        printf("Employee no[%d]\n",i+ 1);
        printf("\tCode: ");
        scanf("%d",&p[i]->code);
        fflush(stdin) ;
        printf("\tName: ");
        gets(p[i]->name);
        fflush(stdin);
        printf("\tSalary: ");
        scanf("%f", &p[i]->salary);
    }

    for (i=0 ; i < size-1; i++)
        for (j=i+1 ;j<size;j++)
            if(p[i]->code > p[j]->code)
                swap (p[i],p[j]);
                    printf("\nN0 \t\tCode \t\tName \t\tSalary\n");
                    printf("-------------------------------------------------------------------------\n");
    for (i=0;i<size;i++)
            printf("%d\t\t%d\t\t%s\t\t%.2f\n", i+ 1 ,p[i]->code,p[i]->name,p[i]->salary);
}
void swap(struct employee *a,struct employee *b)
{
struct employee temp;
temp = * a;
*a = * b;
*b = temp;
}
```

Size of Structure-struct padding

The sizeof for a struct is not always equal to the sum of sizeof of each individual member. This is because of the padding added by the compiler to avoid alignment issues.

In order to align the data in memory, one or more empty bytes (addresses) are inserted (or left empty) between memory addresses which are allocated for other structure members while memory allocation. This concept is called structure padding.

Padding is only added when a structure member is followed by a member with a larger size or at the end of the structure.

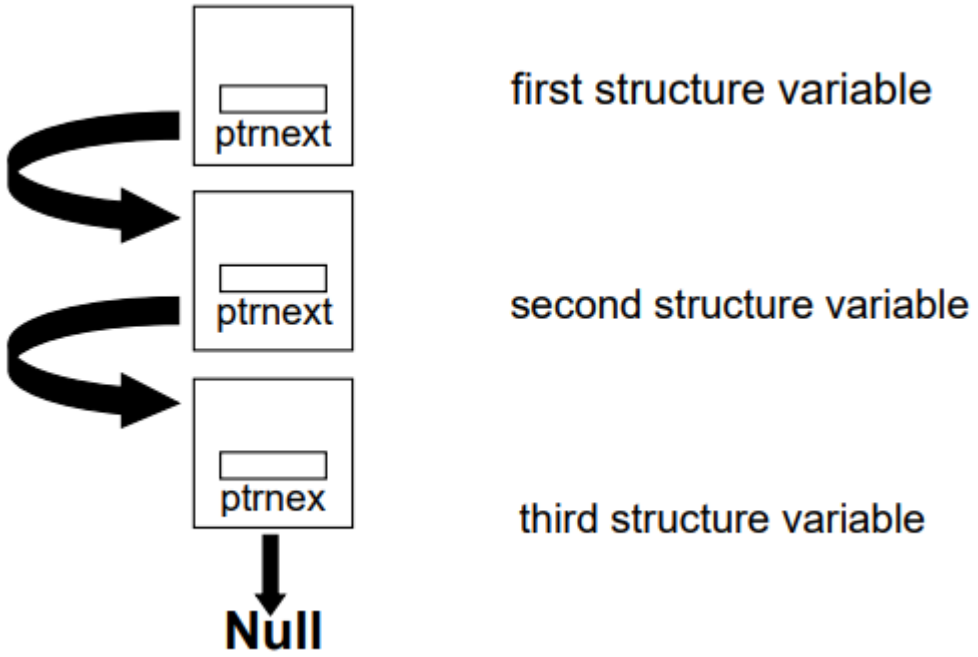
Different compilers might have different alignment constraints as C standards state that alignment of structure totally depends on the implementation.

```c
#include <stdio.h>
int main()
{
    struct A {
        int x;
        double z;
        short int y;
    };

    struct B {
        double z;
        int x;
        short int y;
    };

    struct C {
        double z;
        short int y;
        int x;
    };

    printf("Size of struct A: %d\n", sizeof(struct A));
    printf("Size of struct B: %d\n", sizeof(struct B));
    printf("Size of struct C: %d\n", sizeof(struct C));

}
```

```
Size of struct A: 24
Size of struct B: 16
Size of struct C: 16
```

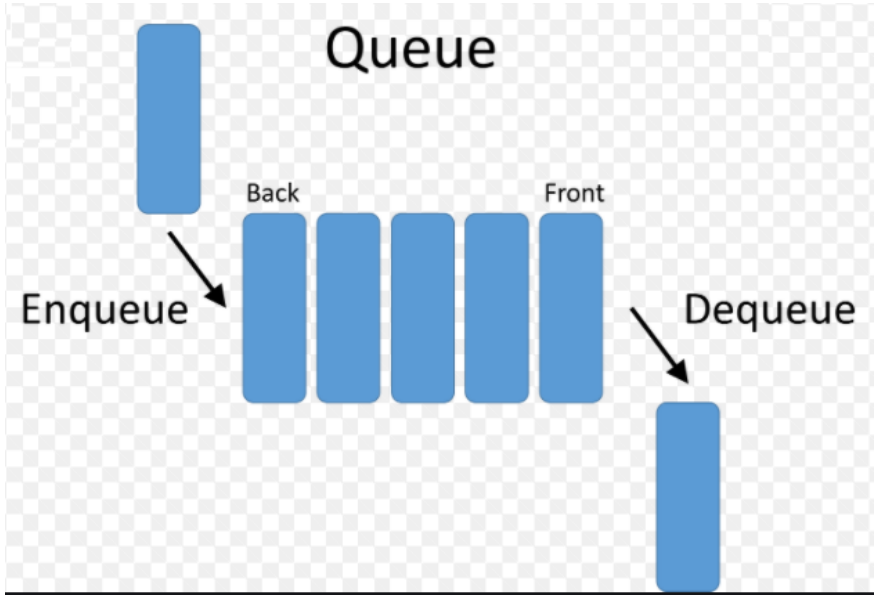
Pointers And Structures
The Linked List

A linked list consists of structures related to one another by pointers, rather than by being members of an array. The basic idea is that each structure on the list contains a pointer that points to the next structure. The pointer in the last structure on the list doesn't point to anything, so we give it the value of 0, or null.



linked list in Queue Style (FIFO) .

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out **(FIFO).** A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.



Insert	➔	Enqueue
Delete	➔	Dequeue

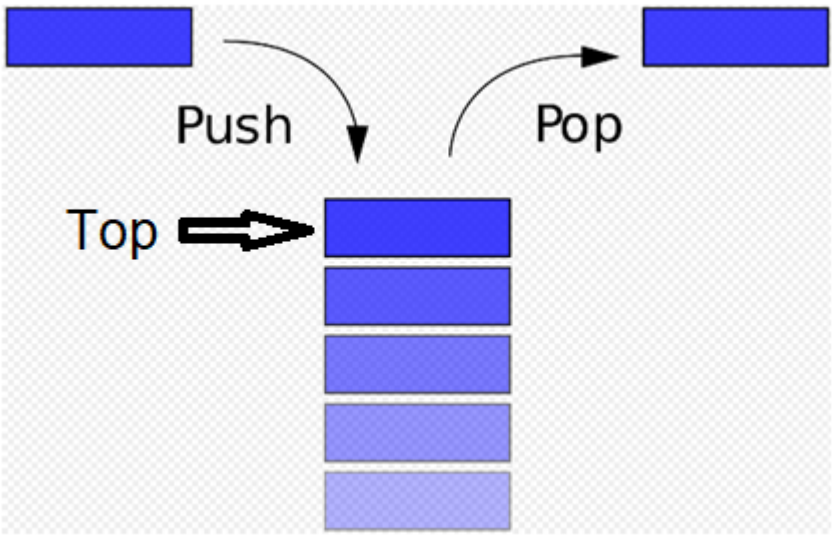
linked list in Queue Style (FIFO) .

```c
#include <stdio.h>
#include <stdlib.h>
/* linked list In Queu (FIFO) */
struct node
{
    int data;
    struct node * link;
};
typedef struct node * pointer;
void main()
{
    struct node * front,*back, * ptr;
    int i,size;

    system("cls");

    printf("\nPlease size of list:");
    scanf("%d", &size);
    front=NULL;
    for(i=1; i<=size; i++)
    {
        if(front==NULL)
        {
            ptr=(pointer) malloc(sizeof(struct node));
            front=ptr;
        }
        else
        {
            ptr->link =(pointer) malloc(sizeof(struct node));
            ptr=ptr-> link;
            back=ptr;
        }
        if(ptr)
        {
            printf("Please value #[%d]:", i);
            scanf("%d", &ptr-> data);
            ptr-> link= NULL;
        }
    }
    /* lost the list in printing */
    /*
     while (front)
     {
         printf("\n%d", front->data);
         front=front-> link;
      }
      /*
      /* temporary pointer for print the head list */
    ptr=front;
    printf("\n front");
    while(ptr)
    {
        printf("->");
        printf("%d",ptr->data);
        ptr=ptr->link;
    }
    printf("<-back\n");

    printf("\nfront point to %d\n",front->data);
    printf("\nback point to %d\n",back->data);

}
```

```
Please size of list:5
Please value #[1]:10
Please value #[2]:20
Please value #[3]:30
Please value #[4]:40
Please value #[5]:50

 front->10->20->30->40->50<-back

front point to 10

back point to 50
```

linked list in Stack Style (LIFO) .

A Stack is a linear structure which follows a particular order in which the operations are performed. The order is Last In First Out **(LIFO).** A good example of a stack is the pile of dinner plates that you encounter when you eat at the local cafeteria: When you remove a plate from the pile, you take the plate on the top of the pile. But this is exactly the plate that was added most recently to the pile by the dishwasher.



Insertion(Push) or deletion(Pop) is done from only one side which called top of stack.

Insert	➔	Push
Delete	➔	Pop

linked list in Stack Style (LIFO) .

```c
#include <stdio.h>
#imclude <stdlib.h>
/* linked list In Stack (LIFO) */
struct node
{
    int data;
    struct node * link;
};
typedef struct node * pointer;
void main(void)
{
        struct node * top, * ptr;
        int i,size;
        system("cls");
        printf("\nPlease size of list:");
        scanf("%d", &size);
        top=ptr= NULL;

        for (i=1; i <=size; i++)
        {
            top=(pointer) malloc(sizeof(struct node));
            if(top)
            {
                printf("Please value #[%2d]: ",i);
                scanf("%d", &top-> data) ;
                top-> link=ptr;
                ptr=top;
            }
        }

        ptr=top;
        while (ptr)
        {
            printf("\n%d",ptr-> data);
            ptr=ptr-> link;
        }

        printf("\n top point to %d\n",top->data);
}
```

```
Please size of list:5
Please value #[ 1]: 10
Please value #[ 2]: 20
Please value #[ 3]: 30
Please value #[ 4]: 40
Please value #[ 5]: 50

50
40
30
20
10
 top point to 50
```

Union in C Language

A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time.

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program.

Like Structures, union is a user defined data type. In union, all members share the same memory location.

```
union color {
        struct color_rgb rgb;
        unsigned int hexvalue;
        char name[10];
};
```

```
#include <stdio.h>
struct color_rgb
{
    unsigned char r;     // Red value
    unsigned char g;     // Green value
    unsigned char b;     // Blue value
};
union color
{
    struct color_rgb  rgb;       // rgb color value
    unsigned int hexvalue;          // integer color value
    char name[10];               // String name of color
};
void main(void)
{
    // Declare color variable
    union color console_color;
    printf("Size of color variable = %d\n\n", sizeof(console_color));
    printf("Enter space separated rgb color value: ");
    scanf("%d %d %d", &console_color.rgb.r,&console_color.rgb.g,&console_color.rgb.b);
    printf("Color in rgb format: %d %d %d\n\n",console_color.rgb.r,console_color.rgb.g,console_color.rgb.b);

    printf("Enter hexdecimal color value: ");
    scanf("%x", &console_color.hexvalue);
    printf("Color in hexdecimal format: %x : %d\n\n", console_color.hexvalue,console_color.hexvalue);

    printf("Enter string color value: ");
    fflush(stdin);// Eat extra new line character
    gets(console_color.name);
    printf("Color in string format: %s\n\n", console_color.name);
}
```

Defining struct Bit Fields in C

In C, we can specify size (in bits) of structure and union members. The idea is to use memory efficiently when we know that the value of a field or group of fields will never exceed a limit or is withing a small range.

```c
#include <stdio.h>
struct wdate {
    unsigned int d ;
    unsigned int m ;
    unsigned int y ;

};
struct date {

    unsigned int d : 5;
    unsigned int m : 4;
    unsigned int y :12;
};

int main()
{
    printf("Size of date is %lu bytes\n", sizeof(struct date));
    struct date dt = { 31, 12, 2014 };
    printf("Date is %d/%d/%d", dt.d, dt.m, dt.y);
    return 0;
}
```

```
Size of date is 4 bytes
Date is 31/12/2014
```

Introduction To programming Using C language

Lesson -30

File I/O

input and output (I/O) to a disk

Files Most programs need to read and write data to disk-based storage systems. Word processors need to store text files, spreadsheets need to store the contents of cells, and databases need to store records.

In this lesson we explore the facilities that C makes available for input and output (I/O) to a disk system.

Disk I/O operations are performed on entities called files.

A file is a collection of bytes that is given a name. In most microcomputer systems, files are used as a unit of storage primarily on floppy-disk and fixed-disk data storage systems

A C-Language program can read and write files in a variety of different ways.

Character, String, Formatted, And Record I/O

The standard I/O package makes available four different ways of reading and writing data.

First, data can be read or written one character at a time. This is analogous to how such functions as putchar() and getche() read data from the keyboard and write it to the screen.

Second, data can be read or written as strings, as such functions as gets() and puts() do with the keyboard and display.

Third, data can be read or written in a format analogous to that generated by printf() and scanf() functions

fourth, data may be read or written in a new format called a record, or block. This is a fixed-length group of data

Text Versus Binary

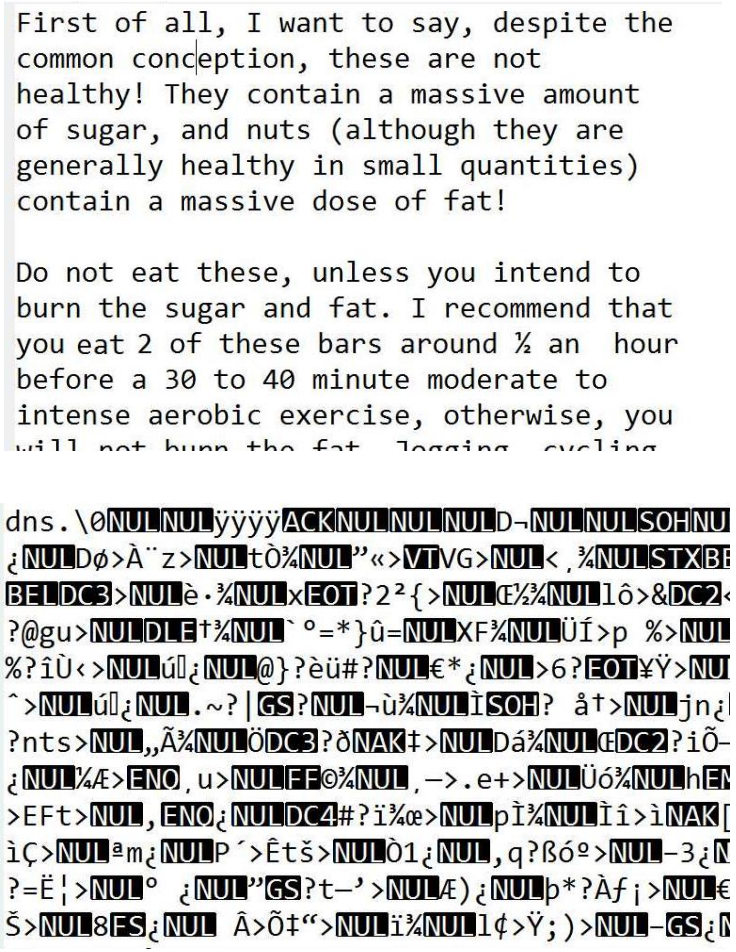
The standard I/O package makes available four different ways of reading and writing data.

Another way to categorize file I/O operations is according to whether files are opened in text mode or binary mode.

Which of these two modes is used to open the file determines how various details of file management are handed; how new lines are stored, for example, and how end-of-file is indicated.

In text format, numbers are stored as strings of characters, while in binary format they are stored as they are in memory: four bytes for an integer, 8 for a double number, and so on.

We'll have more to say about both these text versus-binary distinctions later on.



Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and the program.

```
FILE *fptr;
```

Opening a file:

Opening a file is performed using the fopen() function defined in the stdio.h header file.

The syntax for opening a file in standard I/O is:

```
ptr = fopen("fileopen","mode");
```

Mode	Description
r	Opens an existing text file for reading purpose.
w	Opens a text file for writing. If it does not exist, then a new file is created. Here your program will start writing content from the beginning of the file.
a	Opens a text file for writing in appending mode. If it does not exist, then a new file is created. Here your program will start appending content in the existing file content.
r+	Opens a text file for both reading and writing. If the file exists, loads it into memory and set up a pointer to the first character in it. If file doesn't exist it returns null.
w+	Opens a text file for both reading and writing. If file is present, it first destroys the file to zero length if it exists, otherwise creates a file if it does not exist.
a+	Opens a text file for both reading and writing. It creates the file if it does not exist. The reading will start from the beginning but writing can only be appended.

Closing a File:

The file (both text and binary) should be closed after reading/writing.Closing a file is performed using the fclose() function.

```
fclose(fptr);
```

Character Input/Output

Character Input/Output Our first example program will take characters typed at the keyboard and, one at a time, write them to a disk file.

```c
/* writes one character at a time to a file */
#include <stdio.h>
void main(void)
{
    FILE *fptr;
    char ch;

    fptr = fopen("d:\\data\\textfile.txt", "w");
    while( (ch =getche()) != '\r')
            putc(ch,fptr);
    fclose(fptr) ;
}
```

Character Input/Output Our second example program will take characters from a file, one at a time, write them to a another disk file (copy file).

```c
/* writes one character at a time to a file */
#include <stdio.h>
void main(void)
{
    FILE *fptr_in;
    FILE *fptr_out;
    char ch;



    fptr_in = fopen("d:\\data\\textfile.txt", "r");
    fptr_out = fopen("d:\\data\\textfile_copy.txt", "w");
    while( (ch =getc(fptr_in)) != EOF)
            putc(ch,fptr_out);
    fclose(fptr_in) ;
    fclose(fptr_out) ;
}
```

Trouble Opening The File

it's important for any program that accesses disk files to check whether a file has been opened successfully before trying to read or write to the file. If the file cannot be opened, the fopen() function returns a value of 0 (defined as NULL in stdio.h). Since in C this is not considered a valid address, the program infers that the file could not be opened.

```
if( (fptr_in = fopen("d:\\data\\textfile.txt", "r"))==NULL)
    {
    printf("Can't open file textfile.txt. ");
    exit(1);
    }
```

command line arguments

C allows a program to obtain the command line arguments provided when the executable is called, using two optional parameters of "main()" named "argc (argument count)" and "argv (argument array of strings)".

The "argc" variable gives the count of the number of command-line parameters provided to the program. This count includes the name of the program itself, so it will always have a value of at least one.

The "argv" variable is a pointer to the first element of an array of strings, with each element containing one of the command-line arguments.

```c
#include <stdio.h>
int main(int argc,char * argv[])
{
    int i;

    printf("\n Number of argument %d \n",argc);
    for (i=0; i < argc; i++)
        printf("arg #[%d] = %s\n",i,argv[i]);

    return (argc);
}
```

```
D:\C_Lang_C>echo %ERRORLEVEL%
0

D:\C_Lang_C>test_arg

 Number of argument 1
arg #[0] = test_arg

D:\C_Lang_C>echo %ERRORLEVEL%
1

D:\C_Lang_C>test_arg koko toto

 Number of argument 3
arg #[0] = test_arg
arg #[1] = koko
arg #[2] = toto

D:\C_Lang_C>echo %ERRORLEVEL%
3

D:\C_Lang_C>
```

Generalized copy file

Using command line argument you can generalized file copy program to make input file one of its argument and output file one of its argument .

```c
#include <stdio.h>
void filecopy(FILE *,FILE *);
void main(int argc, char * argv[])
{
    FILE *infp, *outfp;

    if (argc!=3)
     {
        printf("\Invalied Number of Paramters\n");
        exit(0);
     }
    if( (infp=fopen(argv[1], "r"))==NULL)
     {
        printf(" cannot open input file %s\n",argv[1]);
        exit(0);
     }
    if((outfp=fopen(argv[2], "w"))==NULL)
     {
        printf(" cannot open output file %s\n",argv[2]);
        exit(0);
     }

    filecopy(infp, outfp);
    fclose(infp);
    fclose(outfp);
}
void filecopy(FILE * ifp,FILE * ofp)
{
    char ch;

    while((ch=getc(ifp)) != EOF)  putc(ch,ofp);
}
```

```
D:\C_Lang_C>cp
Invalied Number of Paramters

D:\C_Lang_C>cp cp.c
Invalied Number of Paramters

D:\C_Lang_C>cp cp.c cp_copy.c

D:\C_Lang_C>dir cp*.c
 Volume in drive D is DATA
 Volume Serial Number is FC63-F8B0

 Directory of D:\C_Lang_C

08/29/2020  02:51 PM               671 cp.c
08/29/2020  02:52 PM               671 cp_copy.c
               2 File(s)          1,342 bytes
               0 Dir(s)  891,374,612,480 bytes free
```

String (line) Input / Output

Reading and writing strings of characters from and to files is almost as easy as reading and writing individual characters. Here a program that writes strings to a file, using the string I/O function fputs() – fgets().

```c
/* writes sentences typed at keyboard, to file*/
#include <stdio.h>
#include <string.h>
void main(void)
{
    FILE *fptr;
    char string[81];

    fptr = fopen("textfile.txt", "w"); /* open file */
     while (strlen(gets(string)) > 0)
        {
            fputs(string, fptr);
            fputs("\n", fptr);
        }
    fclose(fptr);
}
```

```c
/* read sentences from file */
#include <stdio.h>
#include <string.h>
void main(int argc,char * argv[])
{
    FILE *fptr;
    char string[81];

    fptr = fopen(argv[1], "r"); /* open file */
    /* whats wrong ??? please check */
     while (fgets (string, 80, fptr)!=NULL )
        {
            puts(string);
            puts("\n");
        }
    fclose(fptr);
}
```

Formatted Input / Output

So far we have dealt with reading and writing only characters and text. How about numbers? To see how we can handle numerical data, let's take a leaf from our secret. This program reads the data from the keyboard, and then writes it to our “data.txt" file.

```c
#include <stdio.h>
void main(void)
{
  FILE * fp_out;

  int code;
  char  name[30];
  float salary;

  if ((fp_out=fopen("data.txt","w")) == NULL)
  {
    printf(" can't open file ");
    exit(1);
  }
  do
  {
    printf(" code: ");
    scanf("%d",&code);
    fflush(stdin);
    if (code==0) break;
    printf("name:");
    gets(name);
    printf("salary:");
    scanf("%f",&salary);
    printf("\n");
    fprintf(fp_out,"%-5d%-30s%-7.2f",code,name,salary);

  } while (1);

  fclose(fp_out);
}
```

```
 code: 101
name:youssef shawky
salary:700.70

 code: 202
name:mahir youssef
salary:800.80

 code: 303
name:mehrivan ezz
salary:900.90

 code: 0
```

```c
#include <stdio.h>
void main(void)
{

  FILE * fp_in;

  int code;
  char  name[30];
  float salary;


  if ((fp_in=fopen("data.txt","r")) == NULL)
  {
    printf(" can't open file ");
    exit(1);
  }

  while (fscanf(fp_in,"%d",&code)!=EOF)
  {
       fgets(name,30,fp_in);
       fscanf(fp_in,"%f",&salary);
       printf("%5d\t%30s\t%7.2f\n",code,name,salary);
  }

  fclose(fp_in);
}
```

```
  101       youssef shawky                 700.70
  202       mahir youssef                  800.80
  303       mehrivan ezz                   900.90
```

101 youssef shawky 700.70 202 mahir youssef 800.80 303 mehrivan ezz 900.90

Number Storage in text-binary format

So far we have Number Storage In Text Format It's important to understand how numerical data is stored on the disk by fprintf(). Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, 4 bytes for an integer .

No, They are stored as strings of characters. Thus, the integer 999999 is stored in 4 bytes of memory, but requires 6 bytes in a disk file, one, for each '9' character.

Reading and writing to a binary file:
Functions fread() and fwrite() are used for reading from and writing to a file on the disk respectively in case of binary files.

Writing to a binary file

To write into a binary file, you need to use the `fwrite()` function. The functions take four arguments:

1. address of data to be written in the disk
2. size of data to be written in the disk
3. number of such type of data
4. pointer to the file where you want to write.

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

Reading from a binary file

Function `fread()` also take 4 arguments similar to the `fwrite()` function as above.

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

Reading and writing to a binary file

```c
/* writes binary data to file */
#include <stdio.h>
#include <string.h>
void main(void)
{
    FILE *fptr;
    int code;
    char name[41];
    float salary;

    fptr = fopen("data_formated_bin", "wb");
    do
    {
        printf("code   :");
        scanf("%d",&code);
        if (code==0) break;
        fflush(stdin);
        printf("name   :");
        gets(name);
        fflush(stdin);
        printf("salary :");
        scanf("%f",&salary);
        fwrite(&code, sizeof(code), 1, fptr);
        fwrite(name, sizeof(name), 1, fptr);
        fwrite(&salary, sizeof(salary), 1, fptr);
        printf("\n");
    } while(1);
    fclose(fptr);
}
```

```
code   :100
name   :youssef shawky
salary :600.50

code   :200
name   :namir youssef
salary :800.30

code   :300
name   :mahir youssef
salary :900.34

code   :0
```

data_formated_bin - Notepad

File Edit Format View Help

d youssef shawky ` ío©vÀ▯@ €ÿ` ▯@ À▯@ * ▯DÈ namir youssef ` ío©vÀ▯@ €ÿ` ▯@ À▯@ * 3▯HD,▯ mahir youssef ` ío©vÀ▯@ €ÿ` ▯@ À▯@ * Ã▯aD

```c
/* read binary data from file */
#include <stdio.h>
#include <string.h>
void main(void)
{
    FILE *fptr;
    int code;
    char name[41];
    float salary;
    unsigned char byte;

    fptr = fopen("data_formated_bin", "rb");
    while (!feof(fptr)) /*  whats wrong ? */
    {
        fread(&code, sizeof(code), 1, fptr);
        fread(name, sizeof(name), 1, fptr);
        fread(&salary, sizeof(salary), 1, fptr);

        printf("%5d:%40s:%7.2f\n", code, name, salary);
    } ;
    fclose(fptr);
}
```

```
  100:                                    youssef shawky: 600.50
  200:                                     namir youssef: 800.30
  300:                                     mahir youssef: 900.34
  300:                                     mahir youssef: 900.34
```

```c
if(fread(&code, sizeof(code), 1, fptr)==0) break;
```

Record Input / Output

standard I/O, one answer to these problems is record I/O; sometimes called block I/O. Record I/O writes numbers to disk files in binary (or "untranslated") format, so that integers are stored in four bytes, floating point numbers in four bytes, and so on for the other numerical types

the same format used to store numbers in memory. Record I/O also permits writing any amount of data at once; the process is not limited to a single character or string or to the few values that can be placed in a fprintf() or fscanf() function. Arrays, structures, structures of arrays, and other data constructions can be written with a single statement.

Writing Structures With fwrite()

```c
/* writes agent's records to file */
#include <stdio.h>
#include <stdlib.h>
void  main(void)
{
    struct
    {
        int agnumb;
        char name[40];
        double height;
     } agent;



    FILE *fptr;

    if((fptr=fopen("agents.rec", "wb"))==NULL)
     {
         printf("Can't open file agents.rec");
         exit(1);
     }
    do
    {

        printf("\nagent number:");
        scanf("%d",&agent.agnumb);
        fflush(stdin);
        printf("agent name: ");
        gets(agent.name) ;
        printf("agent height:");
        scanf("%lf",&agent.height);
        fwrite(&agent, sizeof(agent), 1, fptr);
        printf("Add another agent (y/n):? ");
    } while(getche()== 'y');

 fclose(fptr);
}
```

```
agent number:100
agent name: youssef shawky
agent height:189
Add another agent (y/n):? y
agent number:200
agent name: mohab youssef
agent height:182
Add another agent (y/n):? y
agent number:300
agent name: mahir youssef
agent height:185
Add another agent (y/n):? n
Process returned 0 (0x0)   execution time : 52.482 s
Press any key to continue.
```

```
dNULNULNULyoussef shawkyNULu'HövÞÿÿÿèÞ`NULíoHu€EM@NUL€ÿ`NULÞEM@NULNULNULNULNULNUL g@ÈNULNUL
NULmohab youssefNULNULu'HövÞÿÿÿèÞ`NULíoHu€EM@NUL€ÿ`NULÞEM@NULNULNULNULNULNULÀf@,SOHNULNULmahir youssefNUL
NULu'HövÞÿÿÿèÞ`NULíoHu€EM@NUL€ÿ`NULÞEM@NULNULNULNULNULNUL g@
```

reading Structures With fread()

```c
/* read agent's records to file */
#include <stdio.h>
#include <stdlib.h>
void  main(void)
{
    struct
    {
        int agnumb;
        char name[40];
        double height;
     } agent;



    FILE *fptr;

    if((fptr=fopen("agents.rec", "rb"))==NULL)
     {
         printf("Can't open file agents.rec");
         exit(1);
     }


      printf("number\tname\t\t\t\t\theight\n");
      while(fread(&agent, sizeof(agent), 1, fptr))
        printf ("%5d\t%-40s\t%6.2lf\n", agent.agnumb,agent.name,agent.height);

      fclose(fptr);
}
```

```
number  name                                            height
  100   youssef shawky                                  189.00
  200   mohab youssef                                   182.00
  300   mahir youssef                                   185.00

Process returned 0 (0x0)   execution time : 1.291 s
Press any key to continue.
```

Random Access fseek()

So far all our file reading and writing has been sequential. That is, when writing a file we've taken a group of items whether characters, strings, or more complex structures and placed them on the disk one at a time. Likewise, when files reading, we've started at the beginning of the file and gone on until we came to the end. It's also possible to access files "randomly." This means directly accessing a particular data item, even though it may be in the middle of the file.

```
int fseek(FILE *pointer, long int offset, int position)
pointer: pointer to a FILE object that identifies the stream.
offset: number of bytes to offset from position
position: position from where offset is added.

returns:
zero if successful, or else it returns a non-zero value
```

The first argument of tile fseek() function is the pointer to the FILE structure for this file.

The second argument in fseek() is called the offset. This is the number of bytes from a particular place to start reading.

The last argument of the fseek() function is called the mode. There are three possible mode numbers, and they determine where the offset will be measured from.

	Mode	Offset is measured from
0	SEEK_SET	beginning of file
1	SEEK_CUR	current position of file pointer
2	SEEK_END	end of file

Random Access fseek()- Example

```c
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    struct
    {
        int agnumb;
        char name[40];
        double height;
     } agent;



    FILE *fptr;
    int recno;
    long int offset;

    if((fptr=fopen("agents.rec", "r"))==NULL)
     {
     printf("Can 't open file agents.rec ");
     exit(1);
     }


    printf("Enter record number: ");
    scanf("%d", &recno);
    offset = (recno-1) * sizeof(agent);
    if(fseek(fptr, offset, 0) != NULL)
        {
         printf("Can 't move pointer there, ");
         exit(1);
         }

    fread(&agent, sizeof(agent) , 1,fptr );
    printf("Number: %3d\n", agent.agnumb);
    printf("\Name: %s\n", agent. name);
    printf("Height: %.2lf\n", agent. height);

    fclose(fptr);

}
```

```
Enter record number: 2
Number: 200
Name: mohab youssef
Height: 182.00

Process returned 0 (0x0)   execution time : 6.032 s
Press any key to continue.
```