

Lecture 1

Lecture 1

Objectives:

- ANSI C standard library
- Write a C program
 - Print and display a simple string
 - Compile and run a C program
- Comments, declarations, variables, and arithmetic expressions.

■ ANSI C standard library

- The ANSI (American National Standard Institute) C standard library consists of 24 C header files. Each header file contains one or more function declarations, data type definitions, and macros.
- The C library provides a basic set of mathematical functions, string manipulation, type conversions, and file and console-based I/O.
- C is a procedural, imperative, and a general purpose computer programming language (Kernighan and Ritchie, 1972).
- The C Standard Library is a set of C built-in functions, constants and header files:
<stdio.h>, <stdlib.h>, <math.h>, etc.
- The C Standard Library is a reference for C programmers to help them in their projects related to system programming. All the C functions are user-friendly.

- Write a C program
 - Print and display a simple string
 - Compile and run a C program

■ Print and display a simple string

- Dive-in and use a program approach to learn on-by-doing to learning-by-trial and error.

Example 1: Print and display a simple string, and exit the program. The string is “Write a C program and run it”.

Print and display a simple string, and exit the program.
The string is “Write a C program and run it”.

1. **#include<stdio.h>**
2. **main()**
3. **{**
4. **printf(“Write a C program and run it!\n”);**
5. **return 0;**
6. **}**

Standard I/O library

- The first line is a standard I/O library.

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

- The I/O library is used to call the library function **printf** correctly.

Function **main**

- The second line is a function called **main**
main()
- The function **main** will be ‘**called**’ first when the program starts running.
- The empty pair of parentheses indicates the **main** has **no arguments** – no information is to be passed in when **main** is called.

Braces { }

- The third line is an open left brace after the function **main**

{

- The sixth line is a close right brace after the return

}

- The **braces { and }** surround a list of statements that makes up the function **main** in C.

printf statement

- The forth line **printf** is a library function to print the formatted output.

```
printf("Write a C program and run it!\n");
```

- **printf**'s first argument is the string enclosed in double parenthesis "**Write a C program and run it!\n**" followed by **\n** to be printed out.
- The parentheses surround **printf**'s argument list is to define the information that is given to it which it is to act on.
- In strings, any two-character sequence beginning with the backslash **** represents a single special character.
- The sequence **\n** represents the 'new line' character, which prints a carriage return or line feed to end one line of output and move down to the next line.
- The semicolon at the end of the line terminates the statement.

return statement

- The fifth line is a **return**, the program is at its end, the program stops functioning.

return 0;

- The return value from main tells the operating system (OS) whether it succeeded or not.
- By convention, a return value of 0 indicates success.

■ Compile and run a C program

How *one* compiles and runs a C program depends on the compiler and the operating system that are used.

- **The first step** is to type the C program in, a text editor may be used to create a file containing the program text.
- **The second step** is to give the file a name, all C compilers require that files containing C source end with the **extension .c**. The program text can also be placed on a file called **hello.c**
- **The third step** is to run (**execute**) the C program that consists of two steps, compilation proper followed by linking, the compiler initializes the linking step automatically.

On many Unix systems, the command to compile a C program from a source file **hello.c** is **cc -o hello hello.c**. One can type this command at the Unix shell prompt, and it requests the **cc** (C compiler) program be run, creating its output in the file **hello**, and taking its input (the source code is to be compiled) from the file **hello.c**.

- Comments, declarations, variables, and arithmetic expressions.

Example 2.

Print a few numbers to show a simple loop.

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

```
/*print a few numbers to show a loop*/
```

comments

```
// print a few numbers to show a loop //
```

main()

{

or

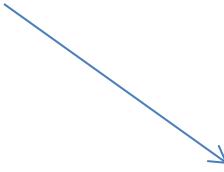
main() {

int i; → **statement**

```
for(i = 0; i < 10; i = i + 1)
```

or

```
for(i=0;i<10;i=i+1)
```



“for loop”

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

```
return 0;  
}
```

or

```
return 0; }
```

```
#include <stdio.h>

/*print a few numbers to show a loop*/

main()
{
    int i;
    for(i = 0; i < 10; i = i + 1)
        printf("i is %d\n", i);
    return 0;
}
```

```
#include <stdio.h>
/*print a few numbers to show a loop*/

main(){
    int i;
    for(i = 1; i < 20; i = i + 1)
        printf("i is %d\n", i);
    return 0;
}
```

The C Programming Language

The George Washington University

Lecture 2

Lecture 2

Program structure, Chapter 1

Objectives:

- Source code for a C program
- **scanf** statement
- Variables in C:
 - Definition of a C variable
 - Global variables
 - Local variables

■ Source code for a C program:

- Free form of writing - Compiler does not take into account the way the C code is arranged:
 - The way it is broken into lines
 - The way the lines are indented
 - The way the whitespace is used between variable names and other punctuation.

■ **scanf** statement

- **scanf** accepts input from a keyboard.
- **scanf** statement `scanf("%d", &num);` reads a value from the keyboard into the integer variable num.
- **scanf** statement `scanf("%f", &sale_price);` reads a float variable from the keyboard into the variable sale_price.
- **scanf** statement `scanf("%c", &grad);` reads a single character from the keyboard into the variable grad.

Example 1.

Write a C program to input a value from the keyboard.

```
#include <stdio.h>
//user uses keyboard to enter data
main()
{
int value;
printf("Enter value from the keyboard:\n");
scanf("%d", &value);
printf("The value of %d\n", value);
}
```

■ Variables in C

- Definition of a C variable name:
 - Variable name (variables) in C represent some unknown, or variable, value.
 - Variables in C may be containing multiple characters:
 - Variables start with lower case letters:
 - Use meaningful identifiers
 - Use separate words within the identifiers with underscores or mixed upper and lower case.
 - Examples:
`area_circle; areaCircle;`
`area_Circle.`
 - Use declaration statement to include the data type of the variable
 - Examples:
`int num; float area ; char arr.`

- Global variables in C:

- Global variables are defined outside a function, usually on top of the program.
- Global variables hold their values throughout the entire life of the program after its declaration.
- Global variables can be accessed inside any of the functions defined for the program.
- Global variables can be accessed by any function.

- Local variables in C:

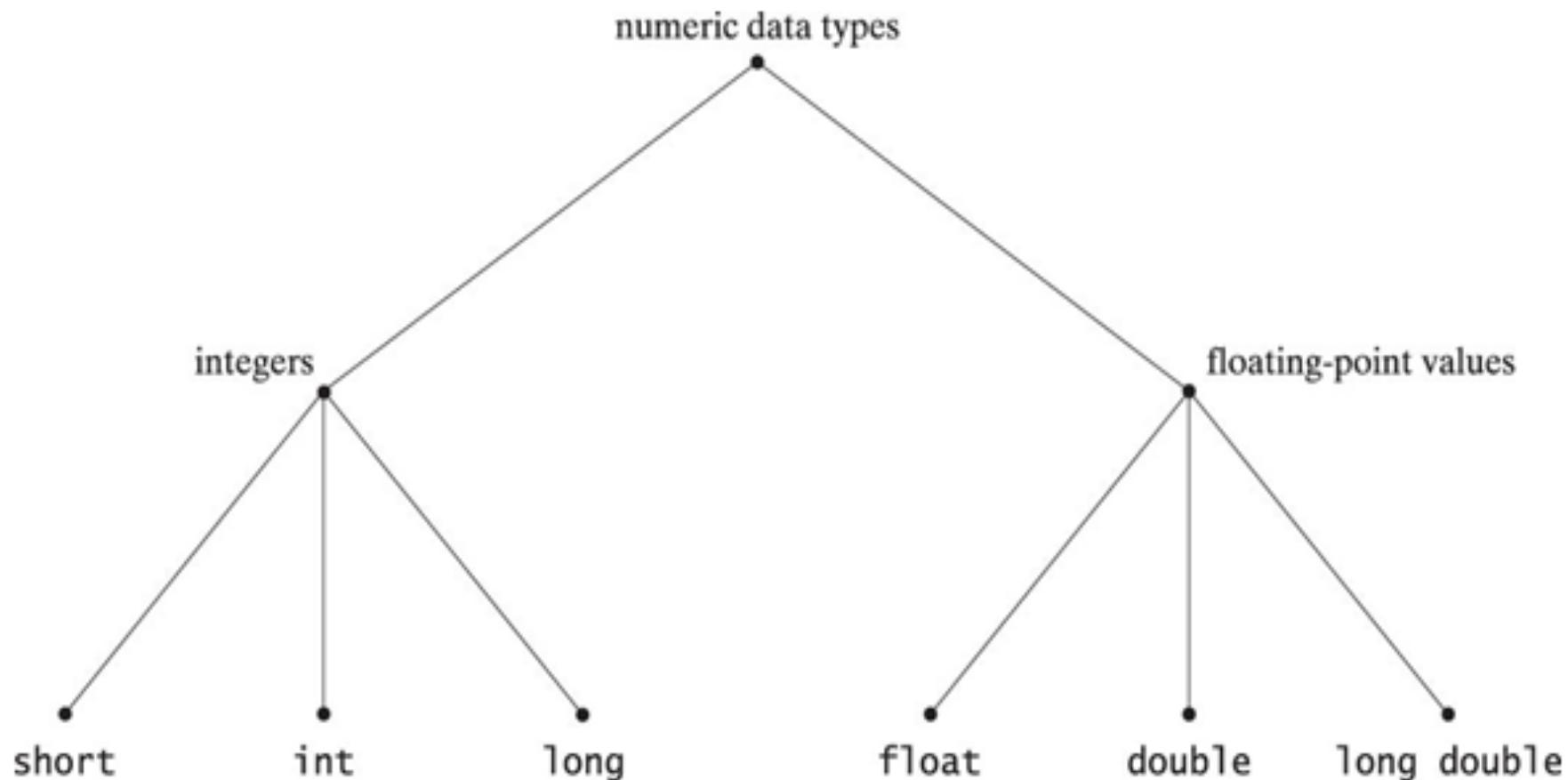
- Local variables are declared inside a function or code block.
- Local variables are used only by statements that are inside that function or block of code.
- Local variables are not known to functions outside their own.
- Local variables p and q are local to main() function.

Example 2:

Write a C program that has a global variable and print the value of the variable.

```
#include <stdio.h>
//global variable declaration
int z;
int main () {
    //local variable declaration
    int p, q;
    //initialize variables
    p = 35;
    q = 19;
    z = p + q;
    printf("value of p=%d, q=%d and z=%d\n", p, q, z);
return 0;}
```

Numeric data types:



Data-type limits for Microsoft Compiler

Integers	
short	Maximum = 32,767
int	Maximum = 2,147,483,647
long	Maximum = 2,147,483,647
Floating Point	
float	6 digits of precision Maximum exponent 38 Maximum value 3.402823e+38
double	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308
long double	15 digits of precision Maximum exponent 308 Maximum value 1.797693e+308

ASCII: American Standard Code for Information Interchange

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr		Dec	Hx	Oct	Html	Chr		Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	 	Space		64	40	100	@	Ø		96	60	140	`	'
1	1	001	SOH	(start of heading)	33	21	041	!	!		65	41	101	A	A		97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"		66	42	102	B	B		98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#		67	43	103	C	C		99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	\$		68	44	104	D	D		100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%		69	45	105	E	E		101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&		70	46	106	F	F		102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'		71	47	107	G	G		103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	((72	48	110	H	H		104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051))		73	49	111	I	I		105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*		74	4A	112	J	J		106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+		75	4B	113	K	K		107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,		76	4C	114	L	L		108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-		77	4D	115	M	M		109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.		78	4E	116	N	N		110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/		79	4F	117	O	O		111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	Ø		80	50	120	P	P		112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	!		81	51	121	Q	Q		113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2		82	52	122	R	R		114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3		83	53	123	S	S		115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4		84	54	124	T	T		116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5		85	55	125	U	U		117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6		86	56	126	V	V		118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7		87	57	127	W	W		119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8		88	58	130	X	X		120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9		89	59	131	Y	Y		121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:		90	5A	132	Z	Z		122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	:		91	5B	133	[[123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<		92	5C	134	\	\		124	7C	174	|	
29	1D	035	GS	(group separator)	61	3D	075	=	=		93	5D	135]]		125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>		94	5E	136	^	^		126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?		95	5F	137	_	_		127	7F	177		DEL

ASCII **Character code 0-31** are unprintable characters,
The first 32 characters are unprintable control
codes and are used to control peripherals such as
printers.

Character code 32-127 are printable characters,
represent letters, digits, punctuation marks, and a
few symbols.

Example: Character 127 represents the command
DEL.

Character code 128-255 are extended characters.

Example 3:

Write C code to print variables as characters and integers.

```
#include <stdio.h>
//Print two values as characters and
//integers
int main() {
//Declare and initialize variables
    char ch='a';
    int i=56;

//Print both variables as characters
    printf("Value of ch is %c; Value of i is
%c \n", ch, i);
//Print both variables as integers
    printf("Value of ch is %i; Value of i is
%i \n", ch, i);
//Exit program
return 0; }
```

Precedence of arithmetic operators

Precedence	Operator	Associativity
1	Parentheses: ()	Innermost first
2	Unary operators: + - (type)	Right to left
3	Binary operators: * / %	Left to right
4	Binary operators: + -	Left to right

Precedence of arithmetic and assignment operators

Precedence	Operator	Associativity
1	Parentheses: ()	Innermost first
2	Unary operators: + - ++ -- (type)	Right to left
3	Binary operators: * / %	Left to right
4	Binary operators: + -	Left to right
5	Assignment operators: = += -= *= /= %=	Right to left

Conversion specifiers for output statements

Variable Type	Output Type	Specifier
Integer Values		
short, int	int	%i, %d
int	short	%hi, %hd
long	long	%li, %ld
int	unsigned int	%u
int	unsigned short	%hu
long	unsigned long	%lu
Floating-Point Values		
float, double	double	%f, %e, %E, %g, %G
long double	long double	%LF, %Le, %LE, %Lg, %LG
Character Values		
char	char	%c

Conversion specifiers for input statements

Variable Type	Specifier
Integer Values	
int	%i, %d
short	%hi, %hd
long int	%li, %ld
unsigned int	%u
unsigned short	%hu
unsigned long	%lu
Floating-Point Values	
float	%f, %e, %E, %g, %G
double	%lf, %le, %lE, %lg, %lG
long double	%Lf, %Le, %LE, %Lg, %LG
Character Values	
char	%c

Example 4:

Write a C code to add and to subtract two numbers entered by user.

```
#include <stdio.h>
//Print addition and subtraction of 2 numbers entered by
user
int main(void) {
//Declare variables
    int num1;
    int num2;
    int result;
//Print numbers entered by user
    printf("Enter first number: ");
    scanf("%d", &num1);
    printf("Enter second number: ");
    scanf("%d", &num2);
    result = num1 + num2;
    printf("Add %d + %d = %d\n", num1, num2, result);
    result = num1 - num2;
    printf("Subtraction %d - %d = %d\n", num1, num2,
result);
    return 0; }      //Exit program
```

Example 5:

Write a source code for a function **max()** to take two parameters n1 and n2 to return the maximum value between the two numbers.

```
int max(int n1, int n2); //function declaration
int main () {
    //definition of local variables
    int a = 7;
    int b = 9;
    int num;
    num = max(a, b); //calling a function to find max number
    printf( "Max number is %d.\n", num );
    return 0;}
//function returning the max between two numbers
int max(int n1, int n2) {
    int result;    //local variable declaration
    if (n1 > n2)
        result = n1;
    else
        result = n2;
    return result;
}
```

■ Arrays in C:

Example 6:

Write a C program to find the sum of all n student grades using arrays.

■ Arrays in C:

```
#include <stdio.h>
int main() {
    int grades[10], p, n, sum=0;
    printf("Enter number of students: ");
    scanf("%d", &n);
    for(p=0; p<n; ++p){
        printf("Enter student grades %d: ", p+1);
        scanf("%d", &grades[p]);
        sum += grades[p];
    }
    // (step size += add AND assignment operator. It adds the right operand to the left operand
    // and assign the result to the left operand).
    printf("Sum = %d", sum);
    return 0;
}
```

■ Pointers in C:

Example 7:

Explain a pointer variable, its declaration, meaning, and initialization:

■ Pointers in C, cont.,

- A pointer is a variable that can hold a memory address.

To declare a pointer: **data_type *pointer_name;**

The name of the pointer variable is preceded by
the *** character**

For example: **int *ptr;**

- **ptr** is declared as a pointer variable to type **int**
- **ptr** can hold the memory address of an **int** variable.
- If the type of the pointer is **T**, it reads the type of ***T** is **T**.

For example: the type of the expression ***ptr** is **int**.

- Pointer variables can be declared together with other variables of the same type.

For example: **int *ptr, i, j, k;**

■ Pointers in C, cont.,

- After declaring a pointer variable, it is to store the memory address of another variable.
- To find the address of a variable, it is to use the **& address operator** before its name.
- A pointer variable can be initialized when declared, provided that the variable that it points to has already been declared.

For example:

```
int a, b, *ptr = &a;
```

■ Pointers in C, cont.,

For example:

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;
    ptr = &a; /* ptr "points to" the memory address of
                a. ptr becomes equal or else "points to" the memory address
                of a.
    printf("Address = %p\n", ptr); /* Display the
                                    memory address of a. The %p specifier is used to display the
                                    memory address in hex. */
    return 0;
}
```

■ Pointers and Arrays in C

- The elements of an array are stored in successive memory locations, with the first one stored at the lowest memory address. The type of the array defines the distance of its elements in memory.

For example:

In a **char** array the distance is one byte (8 bits).

In an **int** array the distance is four bytes (32 bits).

■ Pointers and Arrays in C, cont.,

- The close relationship between pointers and arrays is based on the fact that the name of an array can be used as a pointer to its first element.
 - `arr+1` can be used as a pointer to the second element,
 - `arr+2` as a pointer to the third one, and so on.
 - The following expressions are equivalent:

`arr == &arr[0]`

`arr + 1 == &arr[1]`

`arr + 2 == &arr[2]`

...

`arr + n == &arr[n]`

■ Pointers and Arrays in C, cont.,

What this C program does?

What will be the output of this C program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, arr[5];
    ptr = arr;
    printf("%p %p %p %p\n", ptr, &arr[0], arr,
           &arr);
    return 0;
}
```

Thank you

The C Programming Language

Lecture 3

Data types, operators and expressions,
Chapter 2

Objectives:

- Variables in C
- Constants, #define
- Data types and sizes
- Constants
- Character escapes
- Declarations
- Arithmetic operations
- Logical and relational operators
- Problem solving.

▪ Variables in C

The RAM (random access memory) in the computer consists of millions of successive memory cells. The size of each cell is one *byte*.

For example, a PC with 8 GB (gigabytes) of RAM would have $8 \times 1024 \text{ MB} = 8192 \times 1024 \text{ KB} = 8.388.608 \times 1.024 = 8.589.934.592$ memory cells.

A *variable* in C is a memory location with a given name.

The value of a variable is the content of its memory location. A program may use the name of a variable to access its value.

- The **#define** directive, cont.

For example:

Write a program that reads the radius of a circle and displays its area of the circle.

```
#include <stdio.h>
#define PI 3.142
int main(void)
{
    double radius;
    printf("Enter radius: ");
    scanf("%lf", &radius);
    printf("The area of circle is %.2f\n",
PI*radius*radius);
    return 0;
}
```

- The **#define** directive, cont.

The **#define** directive is used to define a macro which is to represent values that appear many times within the program. A macro is a name that represents a numerical value.

#define macro_name value

For example, the line: **#define PI 3.142**

defines a macro named **PI**. When a program is compiled, the preprocessor replaces each occurrence of **PI** with 3.142.

- The **#define** directive, cont.

Both **const** and **#define** can be used to create names for constants, there are significant differences between them.

#define is used to create a name for a numerical, character, or string constant.

On the other hand, **const** can be used with any type, such as pointers, arrays, structures, and unions.

■ Data types and sizes

- Basic data types in C with minimum ranges define the number of bits (8bits = 1 byte) for the variables.
- **char** (character) - a *character set* that maps the set of characters to set of small numeric codes, using ASCII character set (small codes):
 - The letter **A** is represented by the code 65;
 - The digit **1** is represented by the code 49;
 - The **space** character is represented by the code 32.
 - These code values are translated into shapes on the screen when characters are printed out or generated when typing characters on the keyboard.
 - Arrays use the type of **char**.

- Data types and sizes, cont.

int - an integer, in the range -32,767 to 32,767 of 16 bits;

long int - a larger integer (up to + 2,147,483,647) of 32 or 64 bits;

float- a floating-point number;

double - a floating-point number.

Most variables are of types **int**, **long int**, or **double**. **long int** is used to hold integer values greater than 32,767.

▪Constants

A *constant* is an absolute value used in expressions.

- Simple constants are decimal integers: 0, 1, , 83, 145.
- Constants are presented in base 8 or base 16 (octal or hexadecimal) by prefixing an extra 0 (zero) for octal, or 0x for hexadecimal.
- The constants 100, 0144, and 0x64 represent the same number.
- Compilers convert constants into binary form internally.
- A constant can be of the type **long int** by suffixing it with the letter L .

- Constants, cont.

Floating-point constant is a constant with decimal point or the letter **e** (or both):

3.14

5.

.01

123e4

333.555e7 .

The **e** shows multiplication by a power of 10:

123.456e7 is 123.456 times 10 to the 7th,
or 1,234,560,000.

Floating-point constants are of type double by default.

- Constants, cont.

- A **character constant** is a single character:

'A'

'. '

'%'

- The numeric value of a **character constant** is that character's value corresponding to the machine's character set.

Example: in ASCII 'A' has the value 65.

- Character escapes:

The most common character escapes are:

- \n a **newline character**
- \b a **backspace**
- \r a **carriage return** (without a line feed)
- \' a **single quote** (in a character constant)
- \\" a **double quote** (in a string constant)
- \\ a **single backslash**
- \a an **audible alert**

- **Declarations:**

- A *variable (object)* is a place one can store a value.
- A declaration for the compiler (before its use):

```
float x;  
char c;  
int i;
```

- A declaration with several variables of the same type:

```
int z1, z2;
```

- Declarations with initializers, qualifiers and storage, classes, arrays, functions, pointers, and data structures.

```
int z1=12, z2=34;
```

- Declarations can be placed at the beginning of a function, or a brace-enclosed block of statements , or outside of any function.

- Arithmetic operators:

- **Division operator /** discards any remainder,

Example: $1 / 3$ is 0 and $8 / 5$ is 1.

- If either operand is a floating-point or double, the **division operator /** yields a floating-point result, with a nonzero fractional part.

Example: $2. / 4.0$ is 0.5, and $14.0 / 8.0$ is 1.75.

- **Modulus operator %** gives a remainder when two integers are divided:

Example: $7 \% 4$ is 3.

- **Modulus operator %** only applies to integers.

- Logical operators:

Logical operators perform logical AND (**&&**) and
Logical OR (**||**) operations:

- Logical operators evaluate each operand in terms of its equivalence to 0.
- The result of a logical operation is either 0 or 1. The data type of the result is **int**.

- Relational operators:

Operators	Example	Description
>	$x > y$	x is greater than y
<	$x < y$	x is less than y
\geq	$x \geq y$	x is greater than or equal to y
\leq	$x \leq y$	x is less than or equal to y
\equiv	$x \equiv y$	x is equal to y
\neq	$x \neq y$	x is not equal to y

■ Example of relational operator:

```
#include<stdio.h>

main(){
    int m=75,n=90;
    if (m == n){
        printf("m and n are equal");}
    else{
        printf("m and n are not equal");}}
```

Example 1

What is the output of the C program below?

```
#include <stdio.h>
int main(void)
{
    int i = 10;
    i = i+i;
    printf(" A:%d B:%d\n", i+i, i);
    return 0;
}
```

Example 2:

What is the output of the C program below?

```
#include <stdio.h>
int main(void)
{
    int i = 40;
    float j = 5.65;
    printf(" %f %d\n", i, j);
    return 0;
}
```

Example 3:

What is the output of the program below?

```
#include<stdio.h>
int low(int d);
int main(void)
{
    int d;
    while((d=getchar())!=EOF)
    {
        putchar(low(d));
    }
}
int low(int d)
{
    return d>='A' && d<='Z'? d+'a'-'A':d;
}
```

Example 4:

Write a program to declare two integers, assign to them the values 70 and 20, and display their sum, product, and the remainder (10). To find the remainder use the % operator.

Solution:

```
#include <stdio.h>
int main(void)
{
    int i = 70;
    int j = 20;
    printf(" Sum = %d\n", i+j);
    printf(" Product = %d\n", i*j);
    printf(" Reminder = %d\n", i%j);
    return 0;
}
```

Example 5:

The C program below displays the average of two float numbers.
Do you think there is a bug in this code? Please explain.

```
#include <stdio.h>
int main(void)
{
    double i = 12, j = 5, avg;
    avg = i+j/2;
    printf("Avg = %.2f\n", avg);
    return 0;
}
```

Solution:

Yes, there is a bug. Because the division operator / has priority over the addition operator +, the division $5/2 = 2.5$ is performed first and, next is the addition $12+2.5$ performed, which causes the program to display the incorrect value 14.50 instead of 8.50.

This bug is eliminated by enclosing the expression **i+j** in parentheses.

Because parentheses have the highest priority among all C operators, the expression `in()` is executed first. Therefore, the right assignment is `avg=(i+j)/2.`

Example 6:

Write a program that assigns a two-digit positive value to an integer variable and displays the sum of its digits. For example, if the assigned value is 35, the program should display 8.

Solution:

```
#include <stdio.h>
int main(void)
{
    int i, j, k;
    i = 35;
    j = i/10;
    k = i - (10*j);
    printf("Sum = %d\n", j+k);
    return 0;
}
```

Solution:

The term **i/10** calculates the tens of **i**.

Notice that we could omit the declarations of **j** and **k** and write

```
printf("Sum = %d\n", i/10+(i-(10*(i/10))));
```

Example 7:

Write a C program to use escape sequences.

Solution:

```
#include <stdio.h>
int main(void)
{
    printf("\a"); //Audible alert//
    printf("This\b\b\b is a text\n");
    printf("\75\n"); //The octal number 75 corresponds to the '=' character.
    printf("This\tis\ta\x9text\n"); //The hexadecimal number 9 corresponds to the horizontal tab.
    printf("This is a \"text'\n");
    printf("This is a \\text\\\"n");
    printf("Sample\rtext\n"); //the \r moves the cursor back to the beginning of the line. Samp is overwritten and text is print. As a result, the output is textle.
    printf("This printf uses three lines, but the \
text will appear \
on one line ");
    return 0; }
```

Problems to solve:

Problem 1: Write C program to convert meters to miles.

Problem 2: Write C program to convert Newton to pound.

Problem 3: Write C program to print ASCII values.

Thank you

Lecture 4

Fundamental data types, Chapter 2

Objectives

- The comma operator
- The **sizeof** operator
- The header <limits.h>
- The **typedef** statement
- Table of type conversation code
- Increment and decrement of unary operators
- Problem solving.

Category	Type	Description	Bytes	Minimum	Maximum
Integers	int (or signed int)	Signed integer (of at least 16 bits)	4 (2)	-2147483648	2147483647
	unsigned int	Unsigned integer (of at least 16 bits)	4 (2)	0	4294967295
	char	Character (can be either signed or unsigned depends on implementation)	1		
	signed char	Character or signed tiny integer (guarantee to be signed)	1	-128	127
	unsigned char	Character or unsigned tiny integer (guarantee to be unsigned)	1	0	255
	short (or short int) (or signed short) (or signed short int)	Short signed integer (of at least 16 bits)	2	-32768	32767
	unsigned short (or unsigned shot int)	Unsigned short integer (of at least 16 bits)	2	0	65535
	long (or long int) (or signed long) (or signed long int)	Long signed integer (of at least 32 bits)	4 (8)	-2147483648	2147483647
	unsigned long (or unsigned long int)	Unsigned long integer (of at least 32 bits)	4 (8)	0	same as above
	long long (or long long int) (or signed long long) (or signed long long int)	Very long signed integer (of at least 64 bits)	8	-2^{63}	$2^{63}-1$
	unsigned long long (or unsigned long long int)	Unsigned very long integer (of at least 64 bits)	8	0	$2^{64}-1$
Real numbers	float	Floating-point number, ≈7 digits (IEEE 754 single-precision floating point format)	4	3.4e38	3.4e-38
	double	Double precision floating-point number, ≈15 digits (IEEE 754 double-precision floating point format)	8	1.7e308	1.7e-308
	long double	Long double precision floating-point number, ≈19 digits (IEEE 754 quadruple-precision floating point format)	12 (8)		
Wide Characters	wchar_t	Wide (double-byte) character	2 (4)	Wide Characters	wchar_t

■ The **comma** operator (,)

The comma (,) operator is:

- Left associative: the expressions are evaluated from left to right.
- It is used to merge several expressions to form a single expression.
- The type and the value of the entire expression are those of the last expression evaluated.

Example:

```
#include <stdio.h>
int main()
{
    int a;
    a=10, a=a+50, printf(" The sum is %d.\n", a);
    return 0;
}
```

- The **comma** operator (,)

- The **comma** operator (,)
 - Left associative: the expressions are evaluated from left to right.

■ The **comma** operator (,)

- Left associative: the expressions are evaluated from left to right.
- It is used to merge several expressions to form a single expression.

■ The **comma** operator (,)

- Left associative: the expressions are evaluated from left to right.
- It is used to merge several expressions to form a single expression.
- The type and the value of the entire expression are those of the last expression evaluated.

■ The **comma** operator (,)

- Left associative: the expressions are evaluated from left to right.
- It is used to merge several expressions to form a single expression.
- The type and the value of the entire expression are those of the last expression evaluated.

Example:

■ The **comma** operator (,)

- Left associative: the expressions are evaluated from left to right.
- It is used to merge several expressions to form a single expression.
- The type and the value of the entire expression are those of the last expression evaluated.

Example:

```
#include <stdio.h>
int main()
{
    int a;
    a=10, a=a+50, printf(" The sum is %d.\n", a);
    return 0;
}
```

The **sizeof** operator gets the size of the operand bytes.

Example: Write C code to find out the size of the operands.

```
#include <stdio.h>
int main() {
    printf("sizeof(char) is %d bytes.\n", sizeof(char));
    printf("sizeof(short) is %d bytes\n", sizeof(short));
    printf("sizeof(int) is %d bytes\n", sizeof(int));
    printf("sizeof(long) is %d bytes\n", sizeof(long));
    printf("sizeof(long long) is %d bytes\n", sizeof(long long));
    printf("sizeof(float) is %d bytes.\n", sizeof(float));
    printf("sizeof(double) is %d bytes.\n", sizeof(double));
    printf("sizeof(long double) is %d bytes.\n",
           sizeof(long double));
    return 0; }
```

The header <limits.h>

The **library macros values** are implementation-specific and defined with the **#define directive**:

MacroValueDescription:

CHAR_BIT8

SCHAR_MIN-128

The **C99 standard** also specifies the <[stdint.h](#)> header file, providing names and limits for explicitly-sized platform-independent integer data types such as **int32_t** for a 32-bit signed integer.

The header **<limits.h>** tests the length of the integers.

Example: Write C code to test the limits of the integers.

```
#include <stdio.h>
#include <limits.h>
int main() {
    printf("int max = %d\n", INT_MAX);
    printf("int min = %d\n", INT_MIN);
    printf("unsigned int max = %u\n", UINT_MAX);

    printf("long max = %ld\n", LONG_MAX);
    printf("long min = %ld\n", LONG_MIN);
    printf("unsigned long max = %lu\n", ULONG_MAX);

    printf("long long max = %lld\n", LLONG_MAX);
    printf("long long min = %lld\n", LLONG_MIN);
    printf("unsigned long long max = %llu\n", ULLONG_MAX);

    printf("Bits in char = %d\n", CHAR_BIT);
    printf("char max = %d\n", CHAR_MAX);
    printf("char min = %d\n", CHAR_MIN);
    printf("signed char max = %d\n", SCHAR_MAX);
    printf("signed char min = %d\n", SCHAR_MIN);
    printf("unsigned char max = %u\n", UCHAR_MAX);
return 0; }
```

The header **<limits.h>** tests the length of the integers.

Example: Write C code to test the limits of the integers.

```
#include <stdio.h>
#include <limits.h>
int main() {
    printf("int max = %d\n", INT_MAX);
    printf("int min = %d\n", INT_MIN);
    printf("unsigned int max = %u\n", UINT_MAX);

    printf("long max = %ld\n", LONG_MAX);
    printf("long min = %ld\n", LONG_MIN);
    printf("unsigned long max = %lu\n", ULONG_MAX);

    printf("long long max = %lld\n", LLONG_MAX);
    printf("long long min = %lld\n", LLONG_MIN);
    printf("unsigned long long max = %llu\n", ULLONG_MAX);

    printf("Bits in char = %d\n", CHAR_BIT);
    printf("char max = %d\n", CHAR_MAX);
    printf("char min = %d\n", CHAR_MIN);
    printf("signed char max = %d\n", SCHAR_MAX);
    printf("signed char min = %d\n", SCHAR_MIN);
    printf("unsigned char max = %u\n", UCHAR_MAX);
return 0; }
```

Example:

Write C code to print the system limitations.

```
#include <stdio.h>
#include <limits.h>
#include <float.h>
int main(void)    {
    //Print integer type maximums.
    printf("short maximum: %i \n",SHRT_MAX);
    printf("int maximum: %i \n",INT_MAX);
    printf("long maximum: %li \n\n",LONG_MAX);
    //Print float precision, range, maximum.
    printf("float precision digits: %i \n",FLT_DIG);
    printf("float maximum exponent: %i \n", FLT_MAX_10_EXP);
    printf("float maximum: %e \n\n",FLT_MAX);
    //Print double precision, range, maximum.
    printf("double precision digits: %i \n",DBL_DIG);
    printf("double maximum exponent: %i \n", DBL_MAX_10_EXP);
    printf("double maximum: %e \n\n",DBL_MAX);
    //Print long double precision, range, maximum.
    printf("long double precision digits: %i \n",LDBL_DIG);
    printf("long double maximum exponent: %i \n", LDBL_MAX_10_EXP);
    printf("long double maximum: %Le \n\n",LDBL_MAX);
    return 0;
}
```

The **typedef** statement

- To assign an "**unsigned int**"
- The **typedef** statement is used to create a new name for an existing type.

Example:

- Create a new type for "***unsigned int***"
- Immediately after **#include**, place **typedef**:
typedef unsigned int uint;

Many C compilers define **size_t** as a **typedef** of **unsigned int** or **typedef unsigned int type_t;**

Table of type conversation code

Type	Type Conversion Code	Type & Format
Integers	%d (or %i)	(signed) int
	%u	unsigned int
	%o	int in octal
	%x, %X	int in hexadecimal (%X uses uppercase A-F)
	%hd, %hu	short, unsigned short
	%ld, %lu	long, unsigned long
	%lld, %llu	long long, unsigned long long
Floating-point	%f	float in fixed notation
	%e, %E	float in scientific notation
	%g, %G	float in fixed/scientific notation depending on its value
	%f, %lf (printf), %lf (scanf)	double: Use %f or %lf in printf(), but %lf in scanf().
	%Lf, %Le, %LE, %Lg, %LG	long double
	%c	char
String	%s	string

Field Width

```
int number = 456789;  
printf("number=%d.\n", number);  
    //number=456789.  
  
printf("number=%8d.\n", number);  
    //number= 456789.  
  
printf("number=%3d.\n", number);  
    //Very short, so it is ignored.  
    //number=456789.
```

Precision (decimal places) for floating-point numbers

```
double value = 123.14159265;
printf("value=%lf; \n", value);
    //value=123.141593;
printf("value=%6.2lf; \n", value);
    //value=123.14;
printf("value=%9.4lf; \n", value);
    //value= 123.1416;
printf("value=%3.2lf; \n", value);
    // Since the field-width is too short, it is ignored.
    //value=123.14;
```

Increment /decrement of unary operators

```
#include <stdio.h>
int main() {
    int grade = 20;           // declare and assign
    printf("%d\n", grade);   // 20
    grade++;                 // increase by 1 (post-increment)
    printf("%d\n", grade);   // 21
    ++grade;                 // increase by 1 (pre-increment)
    printf("%d\n", grade);   // 22
    grade = grade + 1;       // also increase by 1 (grade+=1)
    printf("%d\n", grade);   // 23
    grade--;                 // decrease by 1 (post-decrement)
    printf("%d\n", grade);   // 22
    --grade;                 // decrease by 1 (pre-decrement)
    printf("%d\n", grade);   // 21
    grade = grade - 1;       // also decrease by 1 (grade-=1)
    printf("%d\n", grade);   // 20
    return 0;}
```

Increment/decrement of unary operators, cont.

- The increment/decrement unary operator is placed before the operand (prefix operator), or after the operand (postfix operator).
- **++var** is a **pre-increment** var
Use the new value of var: `y=++x;` same as `x=x+1; y=x;`
- **var++** is a **post-increment**
Use the old value of var, then increment var
`y = x++; same as oldX=x; x=x+1; y=oldX;`
- **--var** is a **pre-decrement**
`y = --x; same as x=x-1; y=x;`
- **var--** is a **post-decrement**
`y = x--; same as oldX=x; x=x-1; y=oldX;`

Increment/decrement of unary operators, cont.

```
#include <stdio.h>
int main() {
    int x = 5;
    printf(" %d\n\n", x++);
// x=5; increment x=6; prints old x=5.

    x = 5;
    printf(" %d\n", ++x);
// increment x=6; prints x is 6.

    return 0;
}
```

```
#define token [value]
```

Defining a constant : #define token [value]

- When defining a constant, a value for that constant may not be provided, the token will be replaced with blank text and "defined" for the purposes of #ifdef and ifndef.
- If a value is provided, the token will be replaced with the remainder of the text on the line (See C preprocessor for the list of gotchas).

Defining a parameterized macro:

```
#define token(<arg> [, <arg>s ... ]) statement
```

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

To define a multiline macro, each line before the last is to end with a \, which will result in a line continuation.

Example:

Write a C program to use 0 and 1 to find for the leap year.

```
#include <stdio.h>
int main(){
    char feb;
    int days;
    printf("Enter 0 if the year is not a leap year
otherwise enter 1: ");
    scanf("%c",&feb);
    days=(feb=='0')?28:29;
//If (feb=='1') is true, days are equal to 29.
//If (feb=='0') is false, days are equal to 28.
    printf("Number of days in February= %d",days);
    return 0; }
```

To do for practice:

Write a C program to create the following output:

*

**

Example:

Write a C program to print all 256 ASCII values.

```
#include<stdio.h>
main()
{
    int i;
    char ch;
    for(i=0; i<256; i++)
    {
        printf("%c ", ch);
        ch = ch + 1;
    }
}
```

To do for practice:

Write a C program to create the following output:

```
*****  
**** * ***  
***   ***  
**     **  
*       *
```

Example:

Write a C program to print the first 7 natural numbers using a “for” loop

```
int main()
{
    int i = 1;
    for (i = 1; i <= 7; i++)
    {
        printf(" %d ", i);
    }
    return (0);
}
```

Example:

Write a C program to print the first 10 natural numbers using a “while” loop.

```
int main()
{
    int i = 1;
    while (i <= 10)
    {
        printf(" %d ", i);
        i++;
    }
    return (0);
}
```

Example:

Write a C program to print the first 12 natural numbers using a “do-while” loop.

```
#include<stdio.h>
int main()
{
    int i = 1;
    do
    {
        printf(" %d ", i);
        i++;
    }
    while (i <= 12);
    return (0);
}
```

Example:

Write a C program to convert an integer from decimal number system(base-10) to binary number system(base-2). Size of integer is assumed to be 32 bits.

```
#include <stdio.h>
int main() {
    int n, c, k;
    printf("Enter an integer in decimal number system\n");
    scanf("%d", &n);
    printf("The integer %d in binary number system is:\n", n);
    for (c=31;c>=0;c--) {
        k = n >> c; //Number is shifted by 1 bit
        if (k & 1) //It is either 1 or 0, depending on the least significant bit of k:
            if the last bit is 1, the result of k & 1 is 1; otherwise, it is 0. This is a bitwise AND
            operation.
            printf("1");
        else
            printf("0");
    }
    printf("\n");
    return 0;
}
```

Example:

Write a C program to generate a table of conversions from Fahrenheit to Kelvin for values from 0 degrees F to 200 degrees Fahrenheit. It allows the user (user defined program) to enter the increment between lines.

```
#include <stdio.h>
int main(void)
{
    //Define and initialize the variables .
    double farenheit=0, increment=0, kelvin;
    //Prompt the user for increment.
    while (increment <= 0)
    {
        printf("Enter increment for table:");
        scanf("%lf",&increment);
    }
    //Print the title and the table.
    printf("Farenheit to Kelvin \n");
```

cont.

Example:

Write a C program to generate a table of conversions from Fahrenheit to Kelvin for values from 0 degrees F to 200 degrees Fahrenheit. It allows the user (user defined program) to enter the increment between lines.

```
do
{
    kelvin = (5.0/9.0)*(farenheit + 459.67);
    printf("%4.2f F      %4.2f K \n",farenheit,kelvin);
    farenheit += increment;
}
while (farenheit <= 200.0);
//Exit program.
return 0;
}
```

To do for practice:

The C program below uses linear interpolation to compute the freezing temperature of seawater.

```
#include <stdio.h>
#include <math.h>
int main(void) {
    //Declare variables.
    double a, f_a, b, f_b, c, f_c;
    //Get user input from the keyboard.
    printf ("Use ppt for salinity values. \n");
    printf ("Use degrees F for temperatures. \n");
    printf ("Enter first salinity and freezing temperature: \n");
    scanf ("%lf %lf",&a,&f_a);
    printf ("Enter second salinity and freezing temperature: \n");
    scanf ("%lf %lf",&c,&f_c);
    printf ("Enter new salinity: \n");
    scanf ("%lf",&b);
    //Use linear interpolation to compute new freezing temperature.
    f_b = f_a + (b-a)/(c-a)*(f_c - f_a);
    //Print new freezing temperature.
    printf("New freezing temperature in degrees F: %4.1f \n",f_b);
    return 0; }
```

cont.

To do for practice, cont.

1. Modify the program for the linear interpolation so that it determines the freezing temperatures to go with the following salinity measurements in ppt:
3 8.5 19 23.5 26.8 30.5

1. Modify the program for the linear interpolation so that it converts and prints the new temperature in degrees Centigrade. (Recall the relation between the temperature in degrees Fahrenheit and the temperature in degrees Centigrade.)

The C Programming Language

Computer Science Department
The George Washington University

Lecture 5

Solving problems, Chapter 2

Objectives:

- Write C code
- Compile C program
- Run C program

Symbolic Constants

#define

#define SEC_PER_HOUR 3600

#define Name “gmail.com”

#define TIME (30/2)

Example 1:

Write a C program to state the age of gmail.com

This is a C program that states the age of gmail.com using symbolic constant from #define:

```
#include <stdio.h>
#define NAME "gmail.com"
#define AGE 13
int main() {
printf("%s is about %d years old.\n", NAME, AGE);
return 0; }
```

Example 2:

Write a C code to compute the volume of a sphere with radius r.

```
#include <stdio.h>
#define PI 3.14

int main(void) {
    //Declare variables
    double r, volume;
    //Enter the radius of the sphere
    printf("Enter the radius r of the sphere: \n");
    scanf("%lf",&r);
    //Compute the volume of the sphere
    volume = (4.0/3.0)*PI*r*r*r;
    //Print the volume
    printf("The volume of a sphere with radius %4.2f is %4.2f \n", r,volume);
    //Exit program
    return 0; }
```

Example 3:

Write a C code to print degree-to-radian table using a “*do-while*” loop.

Write C code to print degree-to-radian table using a *do-while* loop

```
#include <stdio.h>
#define PI 3.141
int main(void){
    //Declare and initialize variables
    int degrees=0;
    double radians;
    //Create a loop for degrees to radians
    printf("Degrees to radians \n\n");
    do {
        radians = degrees*PI/180;
        printf("%7i %10.3f \n",degrees,radians);
        degrees += 30;    }
    while (degrees <= 360);
    return 0;    }
```

To do for practicing:

Write a C code to print degree-to-radian table using a “*for*” loop.

Example 4:

Write C program to compute the area of a sector of a circle with angle beta [radians].

```
#include <stdio.h>
int main(void) {
    //Declare variables
    double beta, r, area;
    //Enter the the radii and the angle beta between them
    printf("Enter the radii and angle beta [radians]: \n");
    scanf("%lf %lf",&r,&beta);
    //Compute the area of the sector
    area = (r*r*beta)/2.0;
    //Print the value of the area
    printf("The area of the sector is %4.3f \n",area);
    //Exit program
    return 0; }
```

To do for practicing:

Write C code to relate degree-to-radians in a table using a *for* loop.

Example 6:

Write C code to print a conversion table from inches to centimeters. The inches column starts at 0 and increments by 2.0 in. The last line is to contain the value 40.0 inches.

```
#include <stdio.h>
#define CM_PER_INCH 2.54
int main(void){
    //Declare and initialize variables
    double cm=0, inches=0, increment=2;
    //Print table title, then the values
    printf("Inches to Centimeters\n");
    while (inches <= 40.0) {
        cm = inches*CM_PER_INCH;
        printf("%6.2f %10.2f\n",inches,cm);
        inches += increment;
    }
    return 0; }
```

Example 7:

**Write C code to generate a conversion table from ft/s to mph.
Start the ft/s column at 0 with increments of 10 ft/s.
The last line is to contain the value 200 ft/s.**

```
#include <stdio.h>
#define FT_PER_MI 5280
#define SEC_PER_HOUR 3600
int main(void){
    //Define and initialize variables
    int fps=0, increment=10;
    double mph=0;
    //Print titles and table
    printf("Feet/second to Miles/hour \n");
    while (fps <= 200) {
        mph = (double)fps*SEC_PER_HOUR/FT_PER_MI;
        printf("%5i %.3f\n",fps,mph);
        fps += increment;
    }
    return 0;
}
```

Example 9:

Write C code to generate a conversion table for `EUROS_PER_DOLLAR`
`1.2166; DOLLARS_PER_YEN 0.009289; POUNDS_PER_DOLLAR`
`0.5781.` Use assumptions for the length of the table,
for example, the table may contain 9 or bigger
number of lines.

```
#include <stdio.h>
#define EUROS_PER_DOLLAR 1.2166
#define DOLLARS_PER_YEN 0.009289
#define POUNDS_PER_DOLLAR 0.5781
int main(void){
    int dollars=1, number_lines=9;
    double euros, pounds, yen;

    printf("USD to           Euros,           yen,
pounds \n");
    for (dollars=1; dollars <=number_lines; dollars++){
        euros = (double)dollars*EUROS_PER_DOLLAR;
        yen = (double)dollars/DOLLARS_PER_YEN;
        pounds = (double)dollars*POUNDS_PER_DOLLAR;
        printf("$ %i  %10.2f Euros  %9.2f Y  %7.2f pounds \n",
              dollars,euros,yen,pounds); }
    return 0; }
```

Example 10:

Assume that there are 10,000 acres total with 3000 acres uncut with a reforestation rate of 0.02. Write C code to print a table to show the number of acres forested at the end of each year, for a total of 10 years.

```
#include <stdio.h>
#define REFOREST_RATE 0.02
#define UNCUT_ACRES 3000
#define MAX_YEARS 10
int main(void){
    int year=1;
    double forested=UNCUT_ACRES;
    printf("YEAR      FORESTED ACRES AT END OF YEAR\n");
    printf("-----\n");
    printf("-----\n");
    //Print amount forested for MAX_YEARS
    while (year <= MAX_YEARS){
        forested += REFOREST_RATE*forested;
        printf("%3i      %f\n",year,forested);
        year++;
    }
    return 0;
}
```

Thank you

Lecture 6

Flow of control, Chapter 3 cont.

Objectives

- Program control flow
- **if**
- **if else**
- **if else if**
- **switch-case-break**
- **break**
- **continue**
- **goto** and labels
- Problem solving.

■ Program control flow

- Program begins execution at the `main()` function.
- Statements within the `main()` function are then executed from top-down style, line-by-line.
- The order is rarely encountered in real C program.
- The order of the execution within the `main()` body may be branched.
- Changing the order in which statements are executed is called program control.
- Accomplished by using program control statements.
- One can control the program flows.

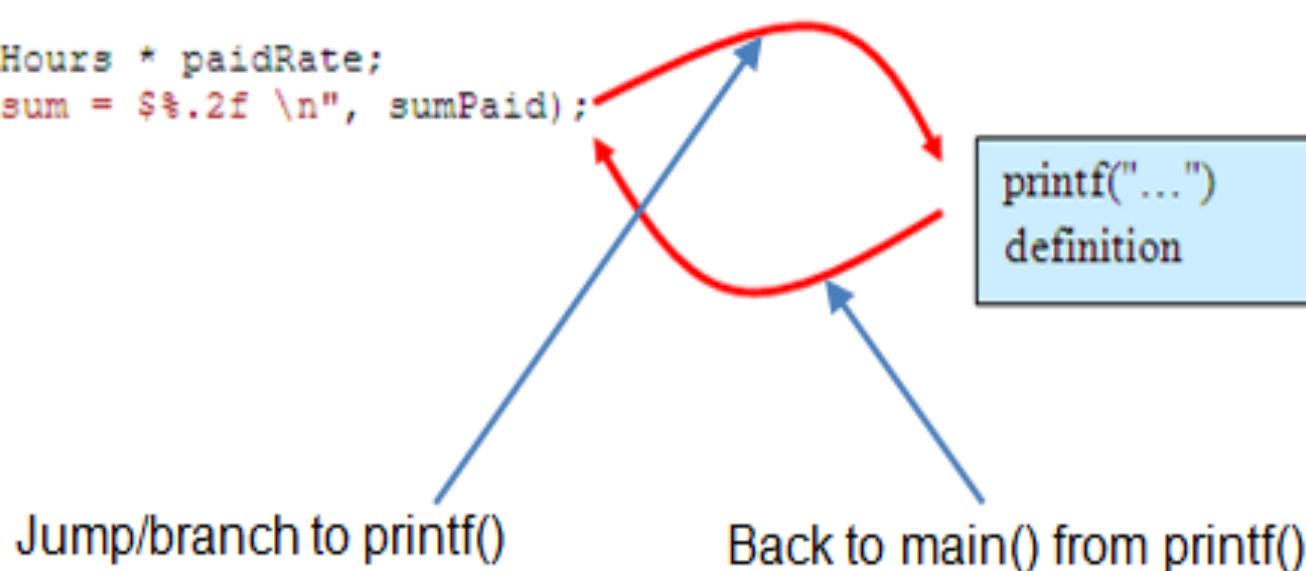
■ Program control flow

- There are three types of program controls:
 - Sequence control structure.
 - Selection structures such as **if**, **if-else**,
nested if, **if-if-else**, **if-else-if** and
switch-case-break.
 - Repetition (loop) such as **for**, **while** and
do-while.
- Certain C functions and keywords also can be used to control the program flows.

■ Program control flow

```
#include <stdio.h> // put stdio.h file here

int main(void)
{
    float paidRate = 5.0, sumPaid, paidHours = 25;
    sumPaid = paidHours * paidRate;
    printf("Paid sum = $%.2f \n", sumPaid);
    return 0;
}
```



■ Program control flow

float paidRate = 5.0, sumPaid, paidHours = 25;	S1
sumPaid = paidHours * paidRate;	S2
printf("Paid sum = \$%.2f \n", sumPaid);	S3
return 0;	S4



- One entry point and one exit point.
- Conceptually, a control structure like this means a sequence execution.

■ Program control flow, cont.,

- The program needs to select from the options given for execution.
- At least 2 options are selected or more than two.
- The option selected is based on the *condition* evaluation result: true or false.

■ Program control flow: if, if-else, if-else-if

Basic if syntax

if (condition)	if (condition)
statement;	{ statements; }
next_statement;	next_statement;

- (condition) is evaluated.
- If TRUE (non-zero) the statement is executed.
- If FALSE (zero) the next_statement following the if statement block is executed.
- During execution, based on some condition, some codes may be skipped.

■ Program control flow

Example:

```
if(myCode == '1')  
rate = 7.20;  
else  
rate = 12.50;
```

If myCode is *equal* to '1', the rate is 7.20 else, if myCode is *not equal* to '1' the rate is 12.50.

character comparison

■ if else statement

```
if(condition_1)
    if(condition_2)
        if(condition_3)
            statement_4;
        else
            statement_3;
    else
        statement_2;
else
    statement_1;
next_statement;
```

■ **if else statement**

- In this nested form, condition _1 is evaluated. If it is zero (FALSE), statement _1 is executed and the entire nested if statement is terminated.
- If non-zero (TRUE), control goes to the second if (within the first if) and condition _2 is evaluated.
- If it is zero (FALSE), statement _2 is executed; if not, control goes to the third if (within the second if) and condition _3 is evaluated.
- If it is zero (FALSE), statement _3 is executed; if not, statement _4 is executed. The statement _4 (inner most) will only be executed if all the if statement are TRUE.
- Again, only one of the statements is executed other will be skipped.
- If the else is used together with if, always match an else with the nearest if before the else.
- *statements_x* can be a block of codes and must be put in curly braces.

■ Three level if-else-if statement

```
if(condition_1)
    statement_1;
else if (condition_2)
    statement_2;
else if(condition_3)
    statement_3;
else
    statement_4;
next_statement;
```

■ Program control flow

- condition_1 is first evaluated. If it is non zero (TRUE), statement_1 is executed and the whole statement terminated and the execution is continue on the next_statement.
- If condition_1 is zero (FALSE), control passes to the next else-if and condition_2 is evaluated.
- If it is non zero (TRUE), statement_2 is executed and the whole system is terminated. If it is zero (FALSE), the next else-if is tested.
- If condition_3 is non zero (TRUE), statement_3 is executed; if not, statement_4 is executed.
- Note that only one of the statements will be executed, others will be skipped.
- *statement_x* can be a block of statement and must be put in curly braces.

■ **Example of if else if**

- If the grade is less than 40 then grade 'F' will be displayed; if it is greater than or equal to 40 but less than 50, then grade 'E' is displayed.
- The test continues for grades 'D', 'C', and 'B'.
- Finally, if the grade is greater than or equal to 90, then grade 'A' is displayed.

■ The **switch-case-break**

- The most flexible selection program control.
- Enables the program to execute different statements based on an condition or expression that can have more than two values.
- Also called multiple choice statements.
- The if statement were limited to evaluating an expression that could have only two logical values:
 TRUE or FALSE.
- If more than two values, have to use nested if.
- The switch statement makes such nesting unnecessary.
- Used together with case and break.

The **switch** constructs has the following form:

```
switch(condition)
{
    case template_1 : statement(s);
                      break;
    case template_2 : statement(s);
                      break;
    case template_3 : statement(s);
                      break;

    ...
    case template_n : statement(s);
                      break;

    default : statement(s);
}

next_statement;
```

■ The **switch-case-break**

- Evaluates the `(condition)` and compares its value with the templates following each `case` label.
- If a match is found between `(condition)` and one of the templates, execution is transferred to the statement (s) that follows the `case` label.
- If no match is found, execution is transferred to the statement (s) following the optional `default` label.
- If no match is found and there is no `default` label, execution passes to the first statement following the `switch` statement closing brace which is the `next_` statement.
- To ensure that only the statements associated with the matching template are executed, include a `break` keyword where needed, which terminates the entire `switch` statement.
- The statement (s) can be a block of code in curly braces.

■ The **switch-case-break**

```
switch(expression) {  
    case constant-expression :  
        statement(s);  
        break; //optional  
    case constant-expression :  
        statement(s);  
        break; //optional  
    //One can have any number of case statements  
    default : //Optional  
        statement(s);  
}
```

■ The syntax of switch-case-break

The expression in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

One can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true.

No break is needed in the default case.

<conio.h>

C inbuilt functions in **conio.h** header file are:

Functions	Description
clrscr()	This function is used to clear the output screen.
getch()	It reads character from keyboard
getche()	It reads character from keyboard and echoes to o/p screen
textcolor()	This function is used to change the text color
textbackground()	This function is used to change text background

■ **Example of block statement:**

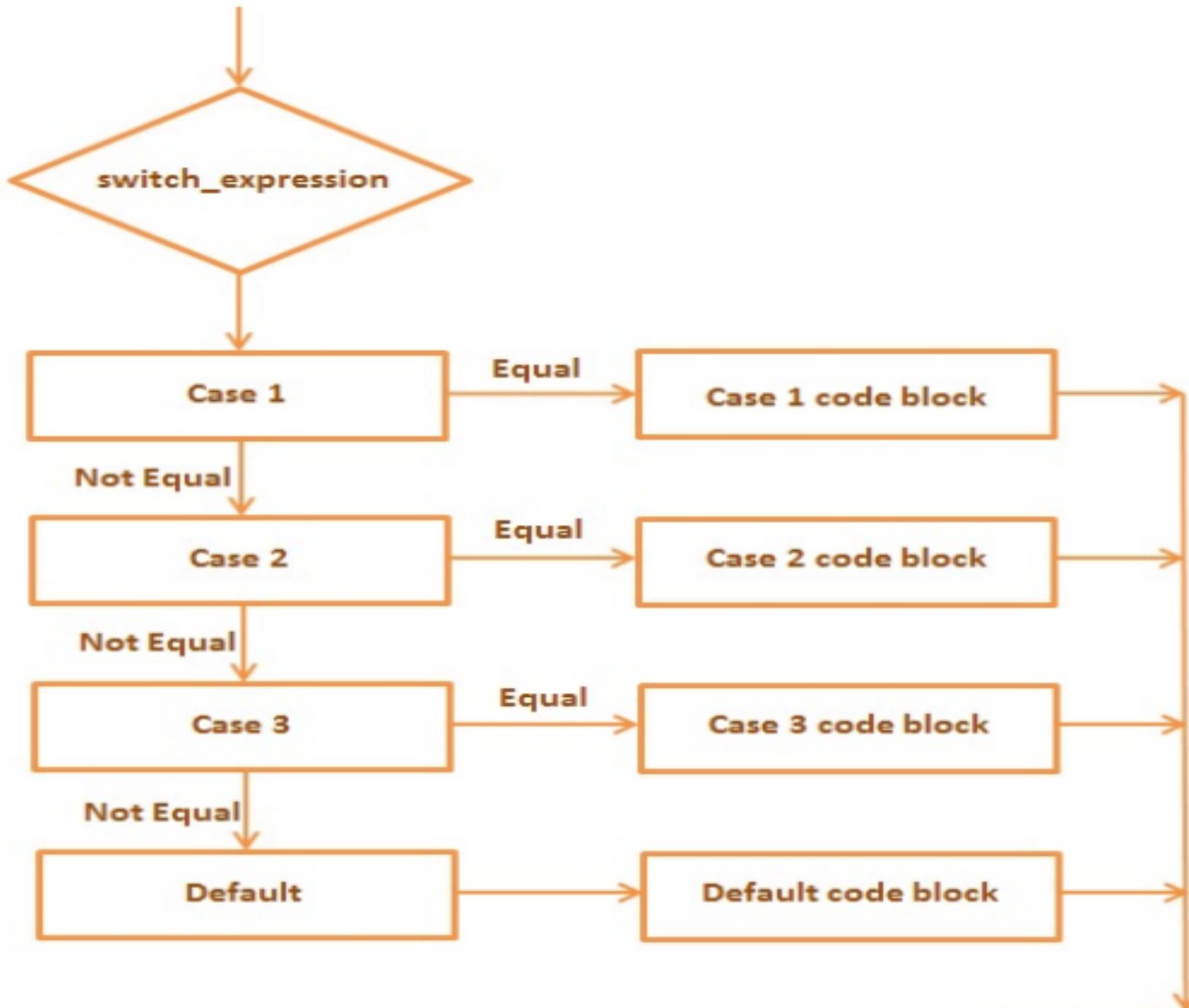
Write a C code to compare two variables m and n and print them out. Use an if statement.

```
#include <stdio.h>
int main()
{ //Main block
    int m = 10;
    int n = 45;
    if (m<n) {
        printf("The variable m is less than n");
    }
    return 0;
//end of main block
}
```

Use nested else if statement to write a C program to print a number to check if it is divisible by both 6 and 9.

```
#include <stdio.h>
#include <conio.h>
void main( ) {
    int num;
    printf("Type a number: ");
    scanf("%d",&num);
    if( num%6==0 && num%9==0) {
        printf("Number is divisible by both 6 and 9");
    }
    else if( num%9==0 ) {
        printf("Number is divisible by 9"); }
    else if(num%6==0) {
        printf("Number is divisible by 6"); }
    else {
        printf("Number is divisible by none"); }
getch();}
```

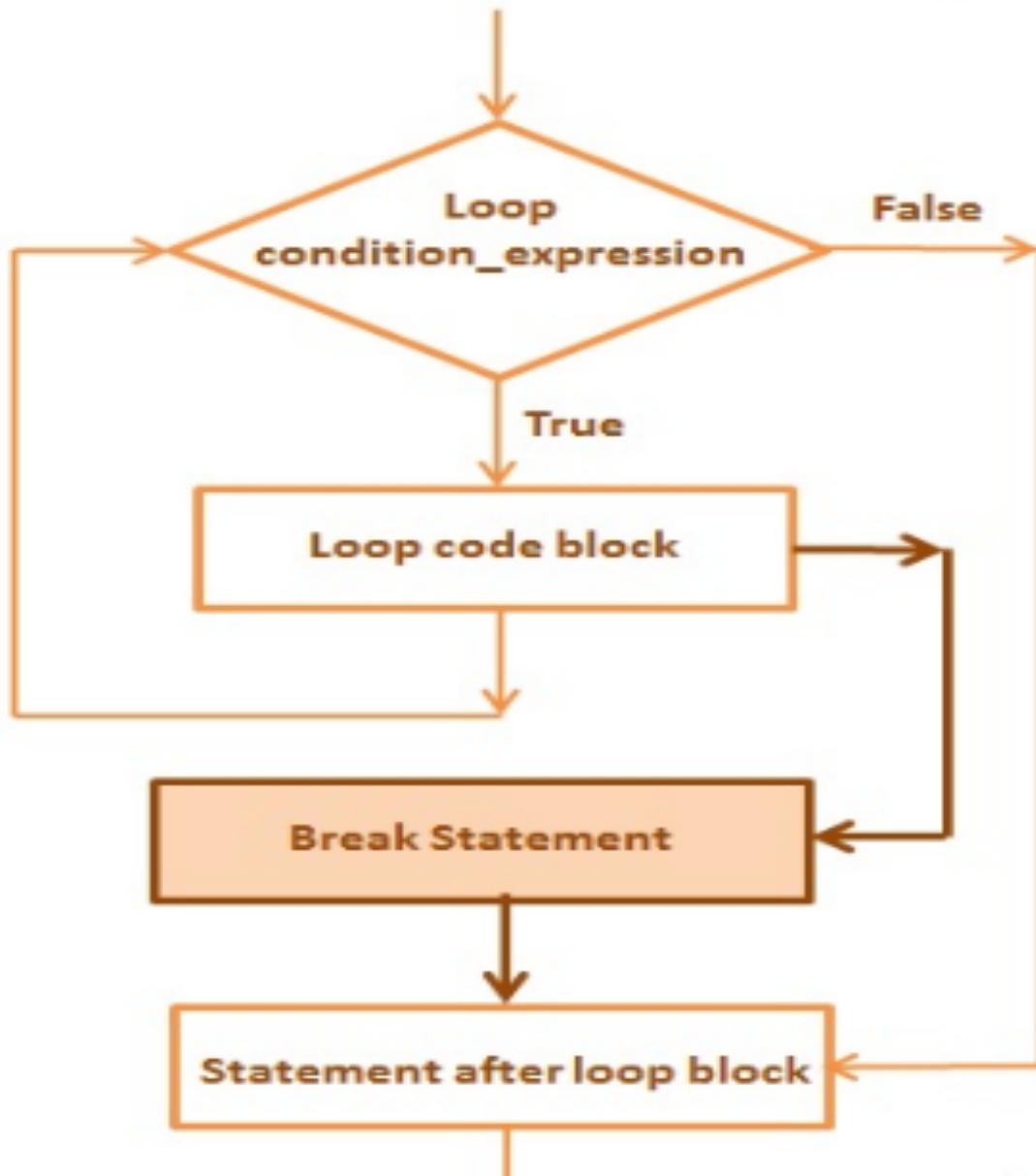
switch statement flow diagram:



**Write C code to create a four operation calculator using
switch...case statement**

```
#include<stdio.h>
#include<conio.h>
int main() {
    char oper;
    float num1,num2;
    printf("Input two numbers : \n");
    scanf("%f%f", &num1, &num2);
    printf("Input an operator (+ - * /)\n");
    scanf("%*c%c",&oper);
    switch(oper) {
        case '+':
            printf("%.1f + %.1f = %.1f",num1, num2, num1+num2);
            break;
        case '-':
            printf("%.1f - %.1f = %.1f",num1, num2, num1-num2);
            break;
        case '*':
            printf("%.1f * %.1f = %.1f",num1, num2, num1*num2);
            break;
        case '/':
            printf("%.1f / %.1f = %.1f",num1, num2, num1/num2);
            break;
        default:
            printf("PROBLEM: CANNOT SOLVE"); }
getch();
return 0; }
```

Control flow of break statement



Write a C program to output prime numbers using a break statement.

```
#include<stdio.h>
#include<conio.h>
void main() {
int z, a;
printf(" Enter a number to find out if it is a
prime number:\n");
scanf(" %d", &z);
for ( a = 2 ; a <= z - 1 ; a++ )
{
if ( z % a == 0 )
{
printf(" %d is not a prime number.\n", z);
break;
}
}
if ( a == z )
printf(" %d is a prime number.\n", z);
getch(); }
```

Write C code to output all prime numbers between 2 and 100 using a break statement:

Write C code to output all prime numbers between 2 and 100 using a break statement:

```
#include <stdio.h>
#include <math.h>
main() {
    int i, j;
    printf("%d\n", 2);
    for(i = 3; i <= 100; i = i + 1) {
        for(j = 2; j < i; j = j + 1) {
            if(i % j == 0)
                break;
            if(j > sqrt(i))
                {
                    printf("%d\n", i);
                    break;
                }
        }
    }
    return 0; }
```

Write C code to find the addition of the integers from **1** to **num** that is an user input number. Use **for** loop. When the addition is greater than 67, use **break** statement to terminate the loop.

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

int main(){
    int count, num, add=0;
    printf("Input a positive number\n");
    scanf("%d", &num);
    for(count=1; count <= num; count++){
        add+= count;
    //If add is greater than 67 break statement terminates the loop
        if(add > 67){
            printf("When add is > 67, the loop terminates.\n");
            break;
        }
    }
    if(count > num)
        printf("Add integers from 1 to %d = %d", num, add);
getch(); }
```

Continue Statement

- The **continue statement** is used to skip some statements inside the loop and continues the next iteration of the loop.
- The **continue statement** only skips the statements after **continue** is inside a loop code block.
- The **continue statement** is used with conditional **if** statement.
- The use of **continue statement** changes the normal sequence of execution of statements inside loop.
- The **continue statement** can be used inside **for**, **while** and **do-while**. It skips the remaining loop's statements and starts next iteration.
- The **continue statement** inside the nested loop will only skip statements of inner loop from where continue will be executed.

goto statement

- Transfers control to a label which resides in the same function and appears before only one statement in the same function.

statement:

labeled-statement

jump-statement

jump-statement:

goto identifier ;

labeled-statement:

identifier : statement

Write a C program to calculate the sum and average of maximum of 4 numbers. User inputs negative number, the sum and average of already entered positive number is displayed.

```
#include <stdio.h>
int main() {
    const int max = 5;
    int n;
    double num, ave, sum=0.0;
    for(n=1; n<=max; ++n) {
        printf("%d Input a number: ", n);
        scanf("%lf",&num);
//User inputs negative number, flow of program moves
to label jump
        if(num < 0.0)
            goto jump;
        sum += num; // sum=sum+num;
    }
jump:
    ave=sum/(n-1);
    printf("Sum = %.1f\n", sum);
    printf("Average = %.1f", ave);
    return 0;}
```

Write a C program to print a right-angled triangular array of natural numbers. It is defined by filling the rows of the triangle with consecutive numbers, starting with the number one in the top left corner.

```
#include <stdio.h>
int main() {
    int rows, p, q, num = 1;
    printf("Enter integer number of rows to print
triangle:");
    scanf("%d",&rows);
    for ( p = 1 ; p <= rows ; p++ ) {
        for ( q = 1 ; q <= p ; q++ ) {
            printf(" %d ", num);
            num++;
        }
        printf("\n");
    }
    return 0;
}
```

Write C code to add all even numbers between 1 and an user input number.

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

int main(){
    //Addition of even numbers
    int count, num, add=0;
    printf("Input a positive number\n");
    scanf("%d", &num);
    for(count=1; count <= num; count++){
        //Use continue statement to skip odd numbers
        if(count%2 == 1) {
            continue;
        }
        add+= count;
    }
    printf("Add all even numbers between 1 to %d
= %d", num, add);
    return(0); }
```

Thank you

Lecture 7

Flow of control, Chapter 3 cont.

Objectives

- Program control flow
- **if**
- **if else**
- **if else if**
- **switch-case-break**
- **break**
- **continue**
- **goto** and labels
- Problem solving.

■ Program control flow

- Program begins execution at the `main()` function.
- Statements within the `main()` function are then executed from top-down style, line-by-line.
- However, this order is rarely encountered in real C program.
- The order of the execution within the `main()` body may be branched.
- Changing the order in which statements are executed is called program control.
- Accomplished by using program control statements.
- So we can control the program flows.

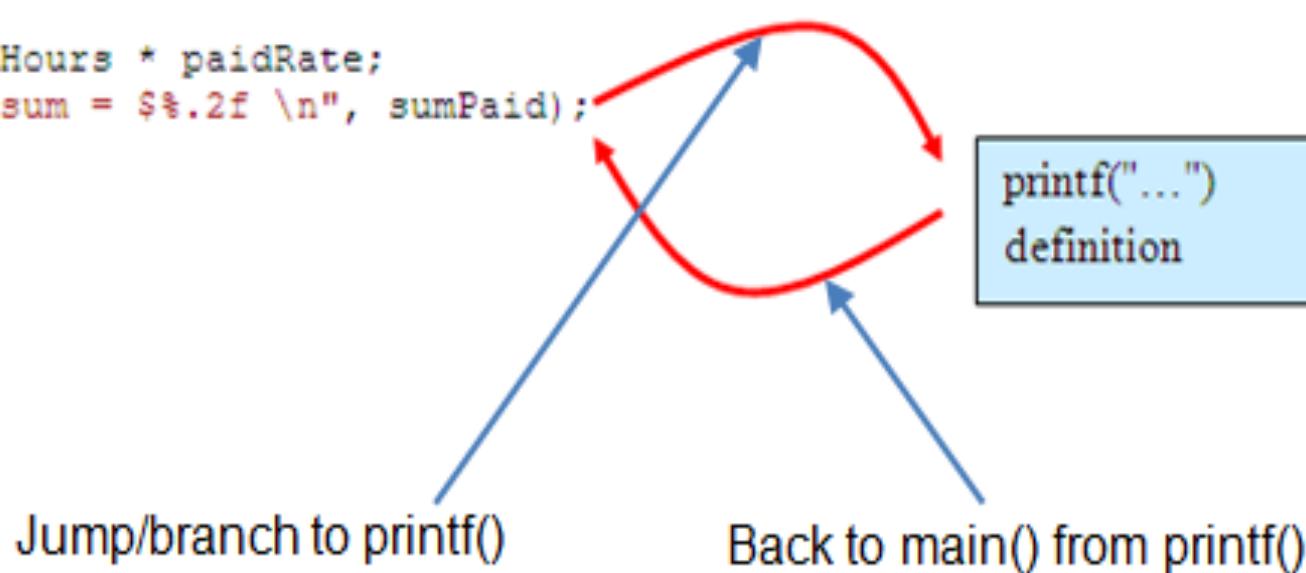
■ Program control flow

- There are three types of program controls:
 - Sequence control structure.
 - Selection structures such as **if**, **if-else**,
nested if, **if-if-else**, **if-else-if** and
switch-case-break.
 - Repetition (loop) such as **for**, **while** and
do-while.
- Certain C functions and keywords also can be used to control the program flows.

■ Program control flow

```
#include <stdio.h> // put stdio.h file here

int main(void)
{
    float paidRate = 5.0, sumPaid, paidHours = 25;
    sumPaid = paidHours * paidRate;
    printf("Paid sum = $%.2f \n", sumPaid);
    return 0;
}
```



■ Program control flow

float paidRate = 5.0, sumPaid, paidHours = 25;	S1
sumPaid = paidHours * paidRate;	S2
printf("Paid sum = \$%.2f \n", sumPaid);	S3
return 0;	S4



- One entry point and one exit point.
- Conceptually, a control structure like this means a sequence execution.

■ Program control flow

- The program needs to select from the options given for execution.
- At least 2 options are selected or more than two.
- The option selected is based on the *condition* evaluation result: TRUE or FALSE.

■ Program control flow: if, if-else, if-else-if

Basic if syntax

if (condition)	if (condition)
statement;	{ statements; }
next_statement;	next_statement;

- (condition) is evaluated.
- If TRUE (non-zero) the statement is executed.
- If FALSE (zero) the next_statement following the if statement block is executed.
- During execution, based on some condition, some codes may be skipped.

■ Program control flow

Example:

```
if(myCode == '1')  
rate = 7.20;  
else  
rate = 12.50;
```

If myCode is *equal* to '1', the rate is 7.20 else, if myCode is *not equal* to '1' the rate is 12.50.

character comparison

■ if else statement

```
if(condition_1)
    if(condition_2)
        if(condition_3)
            statement_4;
        else
            statement_3;
    else
        statement_2;
else
    statement_1;
next_statement;
```

■ **if else statement**

- In this nested form, condition _1 is evaluated. If it is zero (FALSE), statement _1 is executed and the entire nested if statement is terminated.
- If non-zero (TRUE), control goes to the second if (within the first if) and condition _2 is evaluated.
- If it is zero (FALSE), statement _2 is executed; if not, control goes to the third if (within the second if) and condition _3 is evaluated.
- If it is zero (FALSE), statement _3 is executed; if not, statement _4 is executed. The statement _4 (inner most) will only be executed if all the if statement are TRUE.
- Again, only one of the statements is executed other will be skipped.
- If the else is used together with if, always match an else with the nearest if before the else.
- *statements_x* can be a block of codes and must be put in curly braces.

■ Three level if-else-if statement

```
if(condition_1)
    statement_1;
else if (condition_2)
    statement_2;
else if(condition_3)
    statement_3;
else
    statement_4;
next_statement;
```

■ Program control flow

- condition_1 is first evaluated. If it is non zero (TRUE), statement_1 is executed and the whole statement terminated and the execution is continue on the next_statement.
- If condition_1 is zero (FALSE), control passes to the next else-if and condition_2 is evaluated.
- If it is non zero (TRUE), statement_2 is executed and the whole system is terminated. If it is zero (FALSE), the next else-if is tested.
- If condition_3 is non zero (TRUE), statement_3 is executed; if not, statement_4 is executed.
- Note that only one of the statements will be executed, others will be skipped.
- *statement_x* can be a block of statement and must be put in curly braces.

■ **Example of if else if**

- If the grade is less than 40 then grade 'F' will be displayed; if it is greater than or equal to 40 but less than 50, then grade 'E' is displayed.
- The test continues for grades 'D', 'C', and 'B'.
- Finally, if the grade is greater than or equal to 90, then grade 'A' is displayed.

■ The **switch-case-break**

- The most flexible selection program control.
- Enables the program to execute different statements based on an condition or expression that can have more than two values.
- Also called multiple choice statements.
- The if statement were limited to evaluating an expression that could have only two logical values:
 TRUE or FALSE.
- If more than two values, have to use nested if.
- The switch statement makes such nesting unnecessary.
- Used together with case and break.

The **switch** constructs has the following form:

```
switch(condition)
{
    case template_1 : statement(s);
                      break;
    case template_2 : statement(s);
                      break;
    case template_3 : statement(s);
                      break;

    ...
    case template_n : statement(s);
                      break;

    default : statement(s);
}

next_statement;
```

■ The **switch-case-break**

- Evaluates the `(condition)` and compares its value with the templates following each `case` label.
- If a match is found between `(condition)` and one of the templates, execution is transferred to the statement (s) that follows the `case` label.
- If no match is found, execution is transferred to the statement (s) following the optional `default` label.
- If no match is found and there is no `default` label, execution passes to the first statement following the `switch` statement closing brace which is the `next_` statement.
- To ensure that only the statements associated with the matching template are executed, include a `break` keyword where needed, which terminates the entire `switch` statement.
- The statement (s) can be a block of code in curly braces.

■ The **switch-case-break**

```
switch(expression) {  
    case constant-expression :  
        statement(s);  
        break; //optional  
    case constant-expression :  
        statement(s);  
        break; //optional  
    //One can have any number of case statements  
    default : //Optional  
        statement(s);  
}
```

■ The syntax of switch-case-break

The expression in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.

One can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.

When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.

When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.

A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true.

No break is needed in the default case.

- Example of block statement: Write a C code to compare two variables m and n and print them out. Use an if statement.

```
#include <stdio.h>
int main()
{ //Main block
    int m = 10;
    int n = 45;
    if (m<n) {
        printf("The variable m is less than n");
    }
    return 0;
//end of main block
}
```

<conio.h>

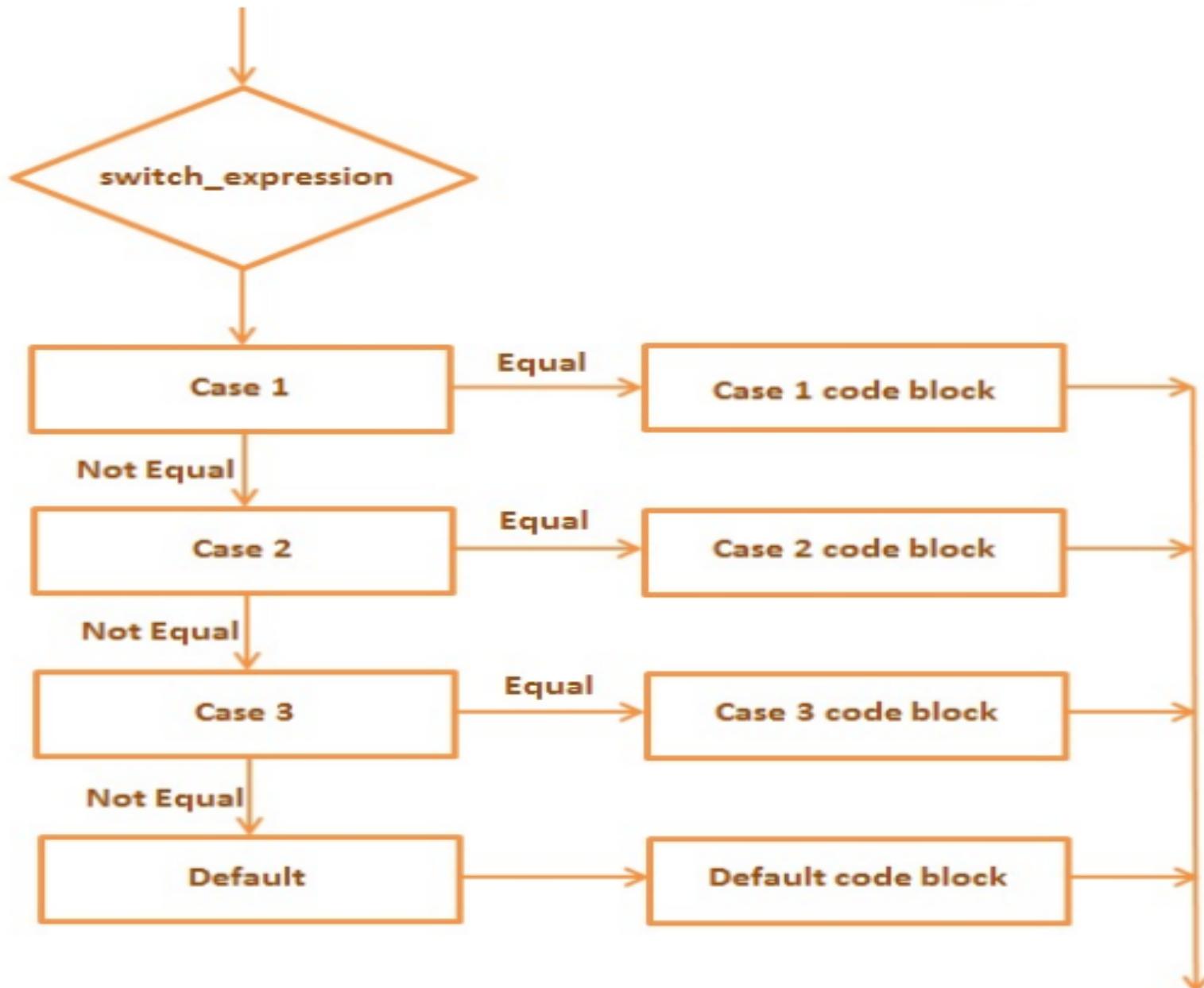
C inbuilt functions in **conio.h** header file are:

Functions	Description
clrscr()	This function is used to clear the output screen.
getch()	It reads character from keyboard
getche()	It reads character from keyboard and echoes to o/p screen
textcolor()	This function is used to change the text color
textbackground()	This function is used to change text background

Use nested else if statement to write a C program to print a number to check if it is divisible by both 6 and 9.

```
#include <stdio.h>
#include <conio.h>
void main( ) {
    int num;
    printf("Type a number ");
    scanf("%d",&num);
    if( num%6==0 && num%9==0) {
        printf("Number is divisible by both 6 and 9");
    }
    else if( num%9==0 ) {
        printf("Number is divisible by 9"); }
    else if(num%6==0) {
        printf("Number is divisible by 6"); }
    else {
        printf("Number is divisible by none"); }
getch();}
```

switch statement flow diagram:

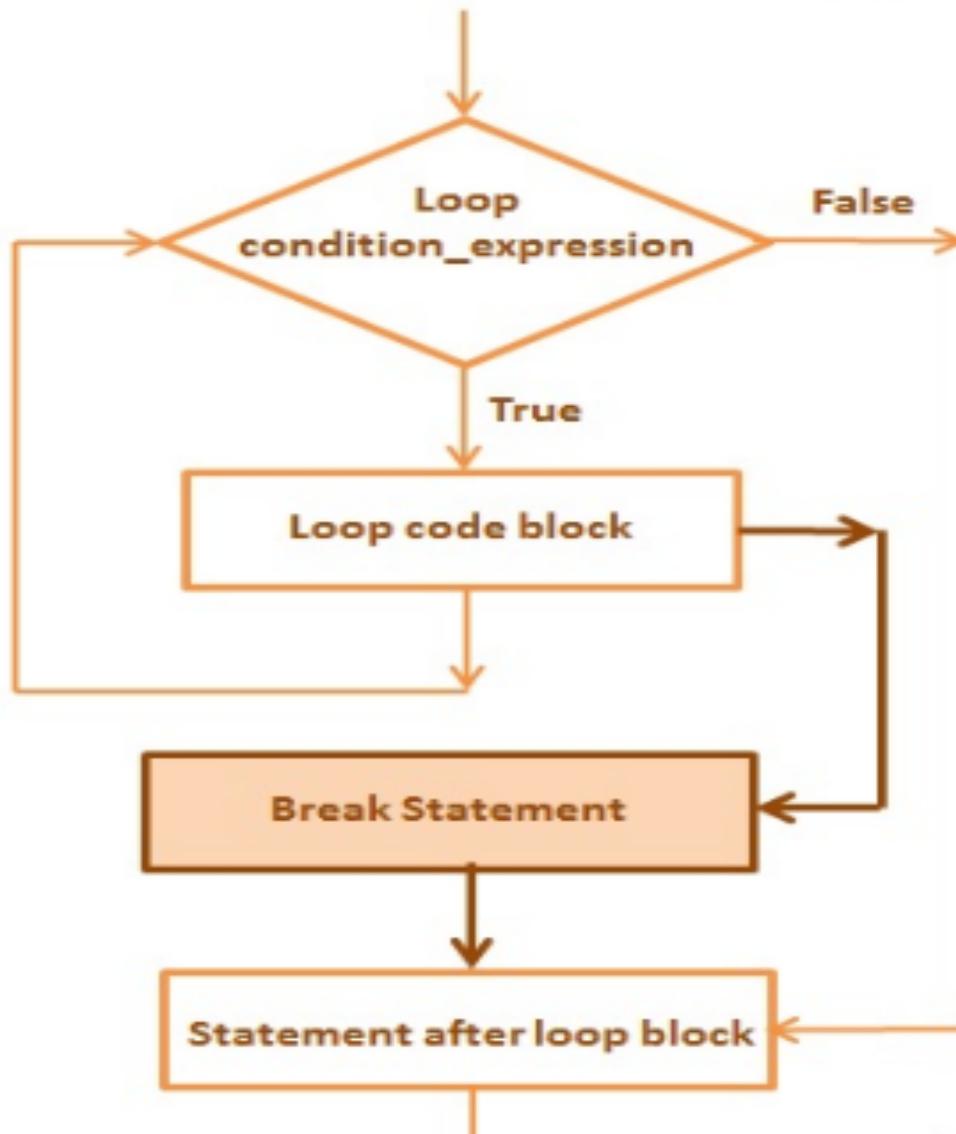


Write C code to create a four operation calculator using
`switch...case` statement

```
#include<stdio.h>
#include<conio.h>
int main() {
    char oper;
    float num1,num2;
    printf("Input two numbers : \n");
    scanf("%f%f", &num1, &num2);
    printf("Input an operator (+ - * /)\n");
    scanf("%*c%c",&oper);
    switch(oper) {
        case '+':
            printf("%.1f + %.1f = %.1f",num1, num2, num1+num2);
            break;
        case '-':
            printf("%.1f - %.1f = %.1f",num1, num2, num1-num2);
            break;
        case '*':
            printf("%.1f * %.1f = %.1f",num1, num2, num1*num2);
            break;
        case '/':
            printf("%.1f / %.1f = %.1f",num1, num2, num1/num2);
            break;
        default:
            printf("PROBLEM: CANNOT SOLVE"); }
getch();
return 0; }
```

Break statement

Control flow of break statement



Write a C program to output all prime numbers from 2 to 100 using a **break** statement.

```
#include <stdio.h>
#include <math.h>
main() {
int i, j;
printf("%d\n", 2);
for(i = 3; i <= 100; i = i + 1)
{
    for(j = 2; j < i; j = j + 1)
        {
            if(i % j == 0)
                break;
            if(j > sqrt(i))
                {
                    printf("%d\n", i);
                    break;
                }
        }
}
return 0;
}
```

Write C code to find the addition of integers from **1** to **num** which is an user input number. Use **for** loop. When the addition is greater than 67, use **break statement** to terminate the loop.

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

int main(){
    int count, num, add=0;
    printf("Input a positive number\n");
    scanf("%d", &num);
    for(count=1; count <= num; count++){
        add+= count;
    //If add is greater than 64 break statement terminates the loop
        if(add > 67){
            printf("When add is > 67, the loop terminates.\n");
            break;
        }
    }
    if(count > num)
        printf("Add integers from 1 to %d = %d", num, add);
getch();
return(0); }
```

Continue Statement

- The **continue statement** is used to skip some statements inside the loop and continues the next iteration of the loop.
- The **continue statement** only skips the statements after **continue** inside a loop code block.
- The **continue statement** is used with conditional **if** statement.
- The use of **continue statement** changes the normal sequence of execution of statements inside loop.
- The **continue statement** can be used inside **for**, **while** and **do-while**. It skips the remaining loop's statements and starts next iteration.
- The **continue statement** inside the nested loop will only skip statements of inner loop from where **continue** will be executed.

goto statement

- Transfers control to a label which resides in the same function and appears before only one statement in the same function.

statement:

labeled-statement

jump-statement

jump-statement:

goto identifier ;

labeled-statement:

identifier : statement

Write a C program to calculate the sum and average of maximum of 4 numbers. User inputs negative number, the sum and average of already entered positive number is displayed.

```
#include <stdio.h>
int main() {
    const int max = 5;
    int n;
    double num, ave, sum=0.0;
    for(n=1; n<=max; ++n)
    {
        printf("%d Input a number: ", n);
        scanf("%lf",&num);
//User inputs negative number, flow of program moves to label jump
        if(num < 0.0)
            goto jump;
        sum += num; // sum=sum+num;
    }
jump:
    ave=sum/(n-1);
    printf("Sum = %.1f\n", sum);
    printf("Average = %.1f", ave);
    return 0;}
```

Write a C program to print Floyd's triangle.

```
#include <stdio.h>
int main() {
    int rows, p, q, num = 1;
    printf("Enter integer number of rows to print
triangle:");
    scanf("%d",&rows);
    for ( p = 1 ; p <= rows ; p++ ) {
        for ( q = 1 ; q <= p ; q++ ) {
            printf(" %d ", num);
            num++;
        }
        printf("\n");
    }
    return 0;
}
```

Write C code to add all even numbers between 1 and an user input number.

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

int main(){
    //Addition of even numbers
    int count, num, add=0;
    printf("Input a positive number\n");
    scanf("%d", &num);
    for(count=1; count <= num; count++){
        //Use continue statement to skip odd numbers
        if(count%2 == 1) {
            continue;
        }
        add+= count;
    }
    printf("Add all even numbers between 1 to %d
= %d", num, add);
    return(0); }
```

Thank you

Lecture 8

Program control flow,
problem solving using loops and statements,
Chapter 3

CSCI 1121
Attendance for February 13

Answer the question and write your first and last name

What operator is used in the C code below and what will be the output of the program?

```
#include<stdio.h>
int main(void)
{
    int a;
    a = (21, 32, 33);
    printf(" %d", a);
    return 0;
}
```

Objectives

- Control flow
- Comma operator
- “for” loops
- Problem solving.

Control flow:

What does the following program output?

```
for(a = 0; a > 1; a++)
{
    printf("%d\n", a);
    printf("End\n");
}
```

Since a is 0, the value of a > 1 is false, the loop body is not executed and the code displays nothing.

Comma operator (,):

//C Program 1

```
#include<stdio.h>
int main(void)
{
    int c;
    c = (1, 2, 33);
    printf(" %d", c);
    return 0;
}
```

// comma operator is executed first and the output is 33.

Comma operator, cont.

//C Program 2

```
#include<stdio.h>
int main(void)
{
    int c = 7;
    c = 5, 7, 33;
    printf(" %d", c);
    return 0;
}
```

// Precedence of (,) is ***least in operator precedence***, the assignment operator takes precedence over (,) and the expression

“c = 5, 7, 33” becomes equivalent to “(c = 5), 7, 33”, it prints 5.

Comma operator, cont.

//C Program 3

```
#include<stdio.h>
int main(void)
{
    int c = 5, 7, 33;
    printf(" %d", c);
    return 0;
}
```

// The **comma operator** (,) is a separator, creates a compilation error.

“**for**” operator

```
for(exp1; exp2; exp3)
```

```
{
```

```
//a block of statements (loop body), that is repeatedly executed as long  
as the value of exp2 is true. //
```

```
}
```

exp1 is executed only once. **exp1** initializes a variable used in the other two expressions.

The value of **exp2** is evaluated (it is a relational condition). If it is false, the loop terminates and the execution of the program continues with the statement after the closing brace. If it is true, the loop is executed.

exp3 is executed, **exp3** changes the value of a variable used in **exp2**.

Steps 2 and 3 are repeated until **exp2** becomes false.

“for” operator, cont.

```
#include <stdio.h>
int main(void) {
    int a;
    for(a=0; a<5; a++) //Braces can be omitted if a single for loop
    {
        printf("%d ", a);
    }
    return 0; }
```

“for” operator, cont.

For example:

```
int a, b;
```

```
for(a = 1, b = 2; b < 10; a++, b++)
```

A **petaflop** is a measure of a computer's processing speed and can be expressed as: A quadrillion (thousand trillion) floating point operations per second (FLOPS) A thousand teraflops. 10^{15} FLOPS.

“for” loop

```
#include <stdio.h>
int main(void)
{
    int i;
    for(i = 2; i < 7; i+=3) ;
        printf("%d\n", i);
    return 0;
}
```

Intentional empty loop:

```
for(i = 2; i < 7; i+=3)  
;
```

Write C code to generate random numbers:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void)
{
    int a, num;
    srand(time(NULL));
    for(a = 0; a < 7; a++)
    {
        num = rand();
        printf(" %d\n", num);
    }
    return 0;
}
```

Omitted expressions in “*for*” loop:

```
int a = 0;
```

```
for(; a < 5; a++)
```

Omitted expressions in “*for*” loop, cont.

```
for(a = 0; a < 5;)
{
    printf("%d ", a);
    a++;
}
```

Omitted expressions in “*for*” loop, cont.

```
for(a = 0; ; a++)
```

//If the controlling expression is missing, the compiler treats it as always **true**, and the loop becomes infinite; it never ends.

“*for*” loop creates an infinite loop

Omitted expressions in “**for**” loop, cont.

for(; ;)

When all three expressions are omitted in order to create an infinite loop.

for(; a<7;) is equivalent to: **while(a<7)**

The **1st** and **3rd** expressions are both missing,
the “**for**” statement is equivalent to a **while** statement:

Example 1:

What is the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int i, j;
    for(i = j; i < 10; j++)
        printf("One\n");
    return 0;
}
```

Example 2:

What is the output of the following program?

```
#include <stdio.h>
int main(void)  {
    int i;
    unsigned int j;
    for(i = 12; i > 2; i-=5)
        printf("%d ", i);
        printf("End = %d\n", i);
    for(j = 5; j >= 0; j--)
        printf("Test\n");
    return 0; }
```

Example 3:

Write a program to read an integer, and if it is within [10, 20], the program is to display the word *ONE* a number of times that is equal to the input integer. Otherwise, the program is to read 10 integers and to display the negatives.

```
#include <stdio.h>
int main(void)  {
    int i, num;
    printf("Enter number: ");
    scanf("%d", &num);
    if(num >= 10 && num <= 20)
    {
        for(i = 0; i < num; i++)
            printf("One\n");
    }
    else
    {
        for(i = 0; i < 10; i++)
        {
            printf("Enter number: ");
            scanf("%d", &num);
            if(num < 0)
                printf("%d\n", num);
        }
    }
    return 0; }
```

Example 4:

Write C program to print the numbers from 20 to 35. Skip number 22.

Example 4:

Write C program to print the numbers from 20 to 35. Skip number 22.

```
#include <stdio.h>
int main () {
    int m = 20; //Local variable
    //Create a loop for execution
    //Label is used, it is a plain text, above the goto
    loop:do {
        if( m == 22) {
            //skip the iteration
            m = m + 1;
            goto loop;}
        printf("The number m is: %d\n", m);
        m++; }
    while( m < 36 );
return 0; }
```

Example 5:

Write a C program to key-in two numbers, write the addition of the two numbers, ask if “you would like to do more additions”, enter “y” for “yes” or “n” for “no” and when it is a valid selection, exit the program. Use goto statement with both forward jump and backward jump.

```
#include<stdio.h>
main() {
    int m,n;
    char ch;
    //clrscr();
    y:printf("Key-in two numbers: ");
    scanf("%d%d",&m,&n);
    printf("The first number is %d\n",m);
    printf("The second number is %d\n",n);
    printf("The addition of the two numbers is: %d+%d=%d\n",m,n,m+n);
    printf("\nWould you like to do more additions? Type Yes or NO ");
    scanf(" %c",&ch);
    if(ch=='y'){
        goto y;
    }
    else if(ch=='n'){
        goto n;
    }
    else {
        printf("Invalid selection.Please type y for 'yes' and n for 'no");
    }
    n:printf("Thank you for exiting!");
}
```

- What is the output of the C program below?

```
int main(){
    int k=2;
    switch (k){
        case 1:
            printf(" Case1 ");
        case 2:
            printf(" Case2 ");
        case 3:
            printf(" Case3 ");
        case 4:
            printf(" Case4 ");
        default:
            printf(" Default ");
    }
    return 0;
}
```

Example 6:

Write C program to test if a number is positive by using nested if... else.

Solution:

```
#include <stdio.h>
int main()
{
    double number;
    printf("Key-in a number: ");
    scanf("%lf", &number);
    //T if number < 0
    if (number < 0.0)
        printf("You keyed-in a negative number.");
    //T if number > 0
    else if ( number > 0.0)
        printf("You keyed-in a positive number.");
    //If both test expression is evaluated to false
    else
        printf("You keyed-in 0.");
    return 0;
}
```

- **while** statement with prefix increment of **(++i)** and postfix increment of **(i++)**

```
#include<stdio.h>
int main(void) {
    int i;
    i = 0;
    while(i++ < 9) {
        printf(" %d\n", i);
        printf("\n");
        i = 0;
    while(++i < 79) {
        printf(" %d\n", i);
    }
    return 0;
}
```

i++ increments the value of **i**, the pre-incremented value tests against **< 9**, output **9** numbers.

++i increments the value of **i**, the incremented value tests against **< 9**; output **8** numbers.

To do for practicing:

Select an interval between two numbers. Write an user defined C program to find the prime numbers in the said interval.

The algorithm is given below to use to create the user defined C program.

Solution :

The while loop is iterated (**high - low - 1**) times.

In each iteration, whether **low** is a prime number or not is checked and the value of **low** is incremented by **1** until **low** is equal to **high**.

If the **for** loop terminates when the test expression of loop **i <= n/2** is false, the entered number is a prime number. The value of **p** is equal to **0** in this case.

If the loop terminates because of **break** statement inside the **if** statement, the entered number is a nonprime number. The value of **p** is **1** in this case.

If the user enters larger number first, this program will not work as intended. Search for and find the solution using swapping the numbers in case the user enters larger numbers.

**1. What will be the output of the C program below?
Write in words your explanation.**

```
#include<stdio.h>
int main()
{
    int a=2;
    int b = a + (1, 2, 3, 4, 8);
    printf(" %d\n", b);
    return 0;
}
```

To do for practicing:

What will be the output of the C program below?

Explain the output of the executed C code.

```
#include<stdio.h>
int main(){
    static int k;
    for(++k;++k;++k) {
        printf(" %d ",k);
        if(k==4)
            break;
    }
    return 0;
}
```

**2. What will be the output of the C program below?
Explain the output of the executed C code.**

```
#include <stdio.h>
int main()
{
    int num=2;
    switch(num+2)
    {
        case 1:
            printf("Case1: The value is: %d", num);
        case 2:
            printf("Case1: The value is: %d", num);
        default:
            printf("Default: The value is: %d", num);
    }
    return 0;
}
```

**3. What will be the output of the following C program?
Explain the output of the executed C code.**

```
#include <stdio.h>
int main()
{
    int i=2;
    switch (i)
    {
        case 1:
            printf("Case1 ");
        case 2:
            printf("Case2 ");
        case 3:
            printf("Case3 ");
        case 4:
            printf("Case4 ");
        default:
            printf("Default ");
    }
    return 0; }
```

4. What will be the output of the following C program?

```
#include <stdio.h>
int main(){
    int i=2;
    switch (i) {
        case 1:
            printf(" Case1 ");
            break;
        case 2:
            printf(" Case2 ");
            break;
        case 3:
            printf(" Case3 ");
            break;
        case 4:
            printf(" Case4 ");
            break;
        default:
            printf(" Default ");
    }
    return 0;
}
```

Thank you

Lecture 9

Problem solving

Objectives

- Problem solving using control flow.

C code to read and write an integer as a sum of two prime numbers

```
#include <stdio.h>
int checkPrime(int n); //checkPrime() returns 1 if
the number passed to the function is a prime number
int main() {
    int n, i, flag = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    for(i = 2; i <= n/2; ++i) {
        //i is a prime number
        if (checkPrime(i) == 1) {
            //n-i is a prime number
            if (checkPrime(n-i) == 1) {
                //n = primeNumber1 + primeNumber2
                printf("%d = %d + %d\n", n, i, n - i);
                flag = 1; } } }
```

cont.

```
if (flag == 0)
printf("%d cannot be expressed as the sum of two
prime numbers.", n);
return 0; }

//Function to check prime number
int checkPrime(int n)
{
    int i, isPrime = 1;
    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
        {
            isPrime = 0;
            break;
        }
    }
    return isPrime; }
```

22, 33, 34, 46, 56, 68, 68, 78 35, 47, 57

C program to find the roots of a quadratic equation

- If determinant is greater than 0, the roots are real and distinct.
- If determinant is equal to 0, the roots are real and repeated.
- If determinant is less than 0, the roots are complex and different (complex conjugate roots).

$$ax^2 + bx + c = 0,$$

where **a**, **b** and **c** are real numbers
and **a** ≠ 0

Using determinant to find the roots

If determinant > 0,

$$\text{root1} = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$\text{root2} = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

If determinant = 0,

$$\text{root1} = \text{root2} = \frac{-b}{2a}$$

If determinant < 0,

$$\text{root1} = \frac{-b}{2a} + i \frac{\sqrt{-(b^2 - 4ac)}}{2a}$$

$$\text{root2} = \frac{-b}{2a} - i \frac{\sqrt{-(b^2 - 4ac)}}{2a}$$

```
#include <stdio.h>
#include <math.h>
int main() {
    double a, b, c, determinant, root1,root2,
realPart, imaginaryPart;
    printf("Enter the coefficients a, b and c: ");
    scanf("%lf %lf %lf",&a, &b, &c);
    determinant = b*b-4*a*c;
//Real and distinct roots
    if (determinant > 0) {
//sqrt() function returns square root
        root1 = (-b+sqrt(determinant))/(2*a);
        root2 = (-b-sqrt(determinant))/(2*a);
        printf("root1=%.2lf and root2=%.2lf",root1,root2);}
```

//Real and repeated roots

```
else if (determinant == 0) {  
    root1 = root2 = -b/(2*a);  
    printf("root1 = root2 = %.2lf;", root1); }
```

//Roots are not real numbers (complex conjugate roots)

```
else {  
    realPart = -b/(2*a);  
    imaginaryPart = sqrt(-determinant)/(2*a);  
    printf("root1 = %.2lf+%.2li and root2 =  
    %.2f-%.2fi",realPart,imaginaryPart,realPart,  
    imaginaryPart); }  
return 0; }
```

C code to compute the power of a number using while loop

```
#include <stdio.h>
int main() {
    int base, power;
    long long answer = 1;
    printf("Enter the base of a number: ");
    scanf("%d", &base);
    printf("Enter the power: ");
    scanf("%d", &power);
    while (power != 0) {
        answer *= base;
        --power; }
    printf("The power of the number is %lld", answer);
    return 0; }
```

C code to compute the power of a number using pow function

```
#include <stdio.h>
#include <math.h>
int main() {
    double base, power, answer;
    printf("Enter a number: ");
    scanf("%lf", &base);
    printf("Enter a power: ");
    scanf("%lf", &power);
    //Compute the power of a number
    answer = pow(base, power);
    printf("The number taken to the power is
%.1lf^.1lf = %.2lf", base, power, answer);
    return 0; }
```

C code to add numbers until user enters zero

```
#include <stdio.h>
int main() {
    double num, add = 0;
    // loop body is executed at least once
    do
    {
        printf("Enter a number: ");
        scanf("%lf", &num);
        add += num;
    }
    while(num != 0.0);
    printf("Sum = %.2lf", add);
    return 0;
}
```

C program to calculate the addition of maximum of 5 numbers with the negative numbers being excluded from the addition:

```
#include <stdio.h>
int main() {
    int i;
    double num, add = 0.0;
    for(i=1; i <= 5; ++i) {
        printf("Enter number%d: ",i);
        scanf("%lf",&num);
        //When user enters negative number, the loop terminates
        if(num < 0.0)
        {
            continue;
        }
        add += num; }
    printf("Sum = %.2lf",add);
    return 0; }
```

C program to output characters without using a format specifier:

```
#include <stdio.h>
int main()
{
    printf(" \x41"); printf("\n");
    printf(" \x42"); printf("\n");
    printf(" \x43"); printf("\n");
    printf(" \x44"); printf("\n");
    return 0;
}
```

Write C program to validate a date:

The logic bused to implement this program:

- **Enter** date.
- **Check** year validation, if year is not valid print error.
- **If year is valid**, check month validation (i.e. month is between 1 to 12), if month is not valid print error.
- **If month is valid**, then finally check day validation with leap year condition, here we will day range from 1 to 30, 1 to 31, 1 to 28 and 1 to 29.
- **If day is valid** print date is correct otherwise print error.

C program that checks for the leap year

```
#include <stdio.h>
int main(){
    int year;
    printf("Enter a year: ");
    scanf("%d",&year);
    if(year%4 == 0)  {
        // A leap year is exactly divisible by 4 except for century years (years ending with 00). The century year is a leap year only if it is perfectly divisible by 400.
        if( year%100 == 0)  {
            if ( year%400 == 0)
                printf("%d is a leap year.", year);
            else
                printf("%d is not a leap year.", year);    }
        else
            printf("%d is a leap year.", year );}
        else
            printf("%d is not a leap year.", year);
    return 0; }
```

```
#include <stdio.h>
int main(){
int dd,mm,yy;
printf("Enter date (DD/MM/YYYY format): ");
scanf("%d/%d/%d",&dd,&mm,&yy);
    //check the year
if(yy>=1900 && yy<=9999) {
    //check the month
    if(mm>=1 && mm<=12){
        //check the days
if((dd>=1 && dd<=31) && (mm==1 || mm==3 || mm==5 ||
mm==7 || mm==8 || mm==10 || mm==12))
            printf("Date is valid.\n");
    }
}
```

cont.

```
else if((dd>=1 && dd<=30) && (mm==4 || mm==6 ||  
mm==9 || mm==11))  
printf("Date is valid.\n");  
    else if((dd>=1 && dd<=28) && (mm==2))  
        printf("Date is valid.\n");  
    else if(dd==29 && mm==2 && (yy%400==0 ||(yy%4==0  
&& yy%100!=0)))  
        printf("Date is valid.\n");  
    else  
        printf("Day is invalid.\n");}  
    else {  
        printf("Month is not valid.\n"); } }  
else {  
    printf("Year is not valid.\n");}  
return 0; }
```

C program to calculate the compound interest

This program reads principal, rate and time in years and then print compound interest on entered principal for given time period.

Compound interest is the amount in which interest is added into the principle so that interest also can be earned interest with the principle.

Use the formula to find the compound interest:

compound_interest=principal*((1+rate/100)^{time}-1)

C program to calculate the compound interest

```
#include <stdio.h>
#include <math.h>
int main() {
    float principal, rate, year, compint;
    printf("Enter principal: ");
    scanf("%f", &principal);
    printf("Enter rate: ");
    scanf("%f", &rate);
    printf("Enter time in years: ");
    scanf("%f", &year);
    //Compute the compound interest
    compint = principal*((pow((1+rate/100),year)-1));
    printf("Compound interest is: %.3f\n", compint);
    return 0;}
```

To do for practicing:

Write C program using a UDF to find the sum and the average of two integers.

In order to create a **User Define Functions**:

Declare the function with appropriate prototype before calling and defining the function.

Function name is to be **very descriptive**.

Keep same argument data type and return data type in declaration and definition of the function.

Pass the same data type arguments which are provided in declaration and definition.

The arguments/parameters which are used while calling the functions are known as **actual arguments/parameters** and arguments/parameters which are used in the definition of the function are known as **local (format) arguments/parameters**.

Thank you

Lecture 10

Functions in C

Objectives

- Function declaration.
- The return statement.
- Function definition.
- Functions calls:
 - Function calls without parameters
 - Function calls by reference
 - Function calls by value
- Problem solving.

Function declaration:

Consists of a function prototype and specifies the name of the function

Example:

```
return_type function_name(parameter_list);
```

- Functions improve the readability of the program,
- Functions are written in the same file with **main()**

Return value from a function

- A function can return one value at most
- The **return_type** specifies the type of the returned value
- A function can return any type: **char**, **int**, **float**, ... or a pointer to some type
- A function is not allowed to return an array or another function
- A function can return a pointer to an array or a function
- If the return type is missing, the function is to return a value of type **int**.
- The return type **void** specifies the function does not return any value.

Function parameters

- A function may take a list of parameters, separated by commas. Each parameter is preceded by its type.
- A function parameter is a variable of the function that is assigned with a value when the function is called.
- If the function has no parameters, **void** sis be used inside parentheses to indicate the empty list.
- However, many programmers omit the word **void** and use an empty pair of parentheses (coming from an older version of C or due to their involvement with C++, where it is not needed to use the word **void**).
- When it is used `main()`, the pair of empty parentheses indicates the function takes an unknown number of parameters, not any.

Function parameters, cont.

void show(**char** ch); //Declare a function that takes a **char** parameter and returns nothing.

double show(**int** a, **float** b); //Declare a function that takes an integer and a float parameter and returns a value of type double.

int *show(**int** *p, **double** a); //Declare a function that takes as parameters a pointer to integer and a double parameter and returns a pointer to integer.

The type of each parameter must be specified, even if several parameters have the same type:

void test(**int** a, b, c);

void test(**int** a, **int** b, **int** c);

Function definition

The general form of a function definition is:

```
return_type function_name(parameter_list)
{
    //Function body
}
```

- No semicolon is added at the end.
- The names of the parameters do not have to match the names as in the function declaration.
- The body of the function contains declarations and statements enclosed in braces.
- The body of a function can be empty.
- Functions can be defined in several files; however, the definition of a function cannot be split between files.
- If a library function is used, the function has already been defined and compiled.

Function definition, cont.,

Example:

```
#include <stdio.h>
void test(void);          //Function declaration
int main(void)
{
    test();               //Function call
    return 0;
}
void test(void)           //Function definition
{
    //Function body
    printf("In\n");
}
```

Complex declarations

- Always decipher the declaration from inside out.

```
int *p[5];
```

Apply precedence rules.

//The variable p is not enclosed in parentheses, because [] take precedence over * go to right, so that p is “an array of five ...,” then, go left to * and have “an array of five pointers ...” add the type and end up with “an array of five pointers to integers.”

Complex declarations, cont.,

- Always decipher the declaration from inside out.
- Find the name of the variable and, if it is enclosed in parentheses (), start deciphering from there.

```
int (*p)[5];
```

//p is enclosed in parentheses.

Deciphering starts inside there - go to left to *

so that p is “a pointer to ...,”

then go to right - “a pointer to an array of five ...,”

next, go left to add the type to end up with

“a pointer to an array of 5 integers.”

Complex declarations, cont.,

- Always decipher the declaration from inside out.
- Find the name of the variable and, if it is enclosed in parentheses (), start deciphering from there.
- Then, continue with the next operator where the precedence order, from high to low, is:
 - The postfix operators. The parentheses () indicate a function, and the brackets indicate [] an array.
 - The prefix operator * indicates “**pointer to**.”

Initialize a pointer >> first array **arr** is declared as:

int arr[5];

Initialize pointer **p** with the address of **arr**:

p = &arr; or

together with the declaration we write:

int (*p)[5] = &arr; //”pointer p to an arr of five”

Complex declaration, cont.,

```
int *(*p)( int );
```

// We start p is “a pointer to ...,”

the () take precedence over *, so we go right

and have “a pointer to a function with an integer argument ...,”

then, go left to add the return type and end up with

“a pointer to a function with an integer argument that
returns a pointer to an integer.”

Complex declaration, cont.,

Example:

```
int *(*p[5])( int *);
```

// We start *deciphering* p inside the parentheses. p is “...”
because [] take precedence over *, we go right and have “an array of
five ...,”
then, go left to *, “an array of five pointers ...,”
because () take precedence over * go right
and have “an array of five pointers to functions with
argument pointer to an integer ...,”
next, go left to add the return type and end up with “an array of 5
pointers to functions with argument pointer to an
integer that returns a pointer to an integer.”

The return statement

```
#include <stdio.h>
int main(void) {
    int num;
    while(1)
    {
        printf("Enter number: ");
        scanf("%d", &num);
        if(num == 2)
            return 0; //Terminate the program
        else
            printf("Num = %d\n", num);
    }
    return 0; }

//The return statement terminates immediately the execution of a function.
//The return statement terminates the main()function if the user enters the
value 2, otherwise it outputs the input value.
```

The return statement, cont.,

Example:

C code for the **avg()** function that compares the values of two parameters, and if they are different, it displays their average. If they are the same, the function terminates.

The return statement, cont.,

```
void avg(int a, int b)
{
    //Function body
    if(a == b)
        return;      //The return statement is unnecessary
    printf("%f\n", (a+b)/2.0);
}
```

//The last **return** is never executed, b/c the first **return** terminates the program. The value returned by **main()** terminates the status of the program. The value **0** indicates normal termination, whereas a nonzero value indicates *abnormal* termination. If the function returns nothing, we just write **return**. The **return** statement at the end of a **void** function is *unnecessary*, the function will return automatically.

The return statement, cont..

Example:

The function **avg()** is modified to return an integer value:

```
int avg(int a, int b)
{
    if(a == b)
        return 0;
    printf("%f\n", (a+b)/2.0);
    return 1;
}
```

The return statement, cont.

Example:

The function `avg()` is modified to return an integer value:

```
int avg(int a, int b)
{
    if(a == b)
        return 0;
    printf("%f\n", (a+b)/2.0);
    return 1;
}
```

- The function is declared **to return a value**, each `return` will be followed by a value.
- The value of the executed `return` is returned to the point at which the function is called.
- The calling function is free to use or discard the returned value.

The second `return` statement at the end of the function can be omitted (compiler issues a message such as “‘`avg`’ : **not all control paths return a value**” or “**control reaches end of non-void function.**”)

The function returns an undefined value, which is definitely undesirable. It is recommended that a **non-void** function always return a value, even if we are allowed not to do it.

The return statement, cont.

Example:

```
int test()
{
    return 4.9;
}
```

The type of the returned value is to match the function's return type. If it does not match, the compiler converts the returned value, if possible, to the return type.

Since `test()` is declared to return an int value, the returned value is implicitly converted to int. Therefore, the function returns 4.

Function calls

A function can be called as many times as needed.

- When a function is called, the execution of the program continues with the execution of the function's code.
- The compiler allocates memory to store the function's parameters and the variables that are declared within its body.
- This memory is reserved from a specific part of the memory, called **stack**;

Example:

- Parameters are passed in system registers for faster access, assuming that the memory is allocated in the stack.
- The compiler allocates memory to store the address where the program returns once the function terminates.
- The allocated memory is automatically released when the function terminates.

Function call without parameters

```
#include <stdio.h>
void test(void);
int main(void) {
    printf("Call 1 ");
    test();          //Function call. Test function does not take any parameters
    printf("Call 2 ");
    test();          //Second call
    return 0; }
void test(void) {
    int i;          //Function body
    for(i = 0; i < 2; i++)
        printf ("In "); }
```

//A call to a function that does not take any parameters when the function name is followed by a pair of empty parentheses. The calling function does not pass any data to the called function. The calling function, `main()`, calls twice the `test()` function:

At the first call of `test()`, the program continues with the execution of the function body. When `test()` terminates, the execution of the program returns to the calling point and continues with the execution of the next statement.

The main program displays Call 2 and calls `test()` again. As a result, the program displays Call 1 In In Call 2 In In.

Function call without parameters, cont.,

Example: C code for the function **test()** w/o parameters to return an integer value.

```
#include <stdio.h>
int test(void);
int main(void) {
    int sum;
    sum = test();          //Function call. The returned value is stored in sum.
    printf(" %d\n", sum);
    return 0; }
int test(void) {
    int i = 10, j = 20;
    return i+j; } The parentheses are omitted when the function is called.
```

Write **test** for the program to compile, the function won't get **called-the compiler treats the name of a function as a pointer**; therefore, it permits the statement. which statement has no effect; it evaluates the address of the function but does not call it. The compiler may issue a warning message such as "*function call missing argument list*" or "*statement with no effect*" to inform the programmer.

test() declares two integers with values 10 and 20 and returns their **sum** which value is stored into **sum** and the program displays 30. It is not needed to declare **sum**; It can be done by **printf("%d\n", test());**

test() is called first and the returned value is used as an argument in **printf()**.

Function call without parameters, cont.,

Functions can be called in any order in a compound expression

```
d = a() + b()* c();
```

//Do not assume that the **b()** and **c()** functions will be called first because of the multiplication precedence.

And if any of them changes a variable on which some other may depend on, the value assigned to **d** will depend on the evaluation order.

Function call with parameters

- A call to a function that takes parameters is made by writing the function name with a list of arguments, enclosed in parentheses.
- Using arguments is one way to pass information to a function.
- The difference between *parameter* and *argument* is that the term ***parameter*** refers to the variables in the ***definition of the function***, while the term ***argument*** refers to the expressions that is in the **function call**.

Function call with parameters, cont.,

For example:

```
#include <stdio.h>
int test(int x, int y);
int main(void) {
    int sum, a = 10, b = 20;
    sum = test(a, b); //The variables a and b become the
function arguments.
    printf("%d\n", sum);
    return 0; }
int test(int x, int y) //The variables x and y are the function
parameters.
{   return x+y; }
```

//The argument can be any *constant*, *variable*, *math*, or *logical* expression, even another function with a return value.

We can omit **sum** and write **printf("%d\n", test(a, b));**
test() is called first, and then **printf()** outputs the return value.

Function call with parameters, cont.,

For example:

```
#include <stdio.h>
int test(int x, int y);
int main(void) {
    int sum, a = 10, b = 20;
    sum = test(a, b); //The variables a and b become the function
arguments.
    printf("%d\n", sum);
    return 0; }
int test(int x, int y) //The variables x and y are the function
parameters. {
    return x+y; }
```

//When it is executed, the compiler allocates eight bytes to store the values of **a** and **b**. When **test()** is called, the compiler allocates another eight bytes to store the values of **x** and **y**. The arguments are evaluated and their values are assigned **one to one** to the corresponding memory locations of the **x** and **y**. Each parameter is a variable that is initialized with the value of the corresponding argument. Therefore, **x** becomes **10** and **y** becomes **20**. When **test()** terminates, the memory allocated for **x** and **y** is automatically released.

Function call with parameters, cont.,

- The number of the arguments and their types need to match the function declaration.
- If the arguments are less, the compiler will produce an error message.
- If the types of the arguments do not match the types of the parameters, the compiler will try to convert implicitly the types of the mismatched arguments to the types of the corresponding parameters. If it is successful, the compiler may display a warning message to inform the programmer for the type conversion. If not, it will produce an error message.

Function call with parameters, cont.,

Example:

Consider the call **test(10.9, b);**

Since the type of the first parameter is **int**, the compiler passes the value 10 to **test()** and the program compiles. In the absence of a function prototype, **char** and **short** arguments are promoted to **int**, and **float** arguments to **double**.

An example with no prototype causes wrong behavior.

Function call with parameters, cont.,

Example:

C code for a program without a declared prototype function

```
#include <stdio.h>
int main(void) {
    test(10);
    return 0; }
void test(double a) {
    printf(" %f\n", a); }
```

When **test()** is called, the compiler does not see a prototype, so it does not know that it accepts an argument of type **double**. Thus, it does not convert the **int** value to **double**, and since the **int** type is not promoted by default, the function outputs an invalid value. **Always declare functions before calling them.**

Function call with parameters, cont.,

The function is able to modify the value of the original variable, this is done when passing its address, not its value. The function must declare the corresponding parameter to be a pointer and use that pointer to access the variable.

Example:

```
#include <stdio.h>

void test(int *p);
int main(void)
{
    int i = 10;
    test(&i);
    printf("%d\n", i);
    return 0;
}
void test(int *p)
{
    *p = 20;
```

Since the `&` operator produces the address of `i`, `&i` is a pointer to `i` and the call matches the declaration.

When `test()` is called, `p` is `&i`.

Since `p` points to `i`, `test()` may change the value of `i`.

Therefore, `*p` is an alias for `i`, and the statement `*p=20` changes `i` from `10` to `20`.
The program outputs `20`.

Function call by reference

The function cannot return more than one value, then one can pass the addresses of the arguments and change their values. By passing the address of the argument it is often referred to as another type of call, named ***call by reference***. It is no different from the call by value, since the function still cannot change the value of the passing argument.

```
#include <stdio.h>
void test(int *p);
int main(void) {
    int *ptr, i;
    ptr = &i;
    printf("%p\n", ptr);
    test(ptr);
    printf("%p\n", ptr);
    return 0; }
void test(int *p) {
    int j;
    p = &j; }
```

Will the value of **ptr** change after calling **test()**?

NO. Although the value of **p** changed and points to **j**,
ptr remains as is because it is the value that is passed; **p** is a copy of **ptr**.
The pointer acts like a reference, it can be used to change the value of an argument.

Function call with parameters, cont.,

When a function is called, it is allowed to **mix pointers and plain values**.

Example :

C program with mix pointers and plain values:

```
#include <stdio.h>
void test(int *p, int a);
int main(void) {
    int i = 100, j = 200;
    test(&i, j);
    printf("%d %d\n", i, j);
    return 0; }
void test(int *p, int a)
{
    *p = 300;
    a = 400;
}
```

//When **test()** is called, we have **p = &i**, and **test()** changes **i** from **100** to **300**, whereas the value of **j** does not change. Thus, the program outputs **300 200**.

Function call by a value

How can a function change the value of a pointer?

Pass an argument to the function of type *pointer to a pointer*.

```
#include <stdio.h>
void test(int **arg);
int var = 100;
int main(void) {
    int *ptr, i = 30;
    ptr = &i;
    test(&ptr); //The value of &ptr is the memory address of ptr, which points to
the address of i. So, the type of the argument is a pointer to a pointer to an integer and matches
the function declaration.
    printf("%d\n", *ptr);
    return 0;
}
void test(int** arg) {
    *arg = &var;
}
```

The memory address of **ptr** is passed to **test()**,
test() may change its value. When **test()** is called, **arg = &ptr**, so ***arg = ptr**. Thus, the
statement ***arg = &var;** is equiv. to **ptr = &var;**
which means that the value of **ptr** changes and points to the address of **var**. Thus, the program
displays **100**.

Problems to solve:

Example:

Write a function that takes as parameters an integer and a character and displays the character as many times as the value of the integer. Write a C program that reads an integer and a character and uses the function to display the character.

```
#include <stdio.h>
void show_char(int num, char ch);
int main(void){
    char ch;
    int i;
    printf("Enter character: ");
    scanf("%c", &ch);
    printf("Enter number: ");
    scanf("%d", &i);
    show_char(i, ch);
    return 0; }

void show_char(int num, char ch) {
    int i;
    for(i = 0; i < num; i++)
        printf("%c", ch);
}
```

Comments: The return type of `show_char()` is declared `void`, because it does not return any value.

Example:

What is the output of the following program?

```
#include <stdio.h>
int f(int a);
int main(void)  {
    int i = 10;
    printf("%d\n", f(f(f(i))));
    return 0;  }
int f(int a)
{
    return a+1;
}
```

Each time **f()** is called, it returns the value of its argument incremented by one.

The calls are executed inside out.

Therefore, the **first** call returns 11, which becomes the argument in the second call.

The second call returns 12, which becomes the argument in the third call.

Therefore, the program displays **13**.

Example:

What is the output of the following program?

```
#include <stdio.h>
void test(int *ptr1, int *ptr2);
int main(void) {
    int i = 10, j = 20;
    test(&i, &j);
    printf("%d %d\n", i, j);
    return 0; }

void test(int *ptr1, int *ptr2) {
    int m, *tmp;
    tmp = ptr1;
    ptr1 = &m;
    *ptr1 = 100;
    *ptr2 += m;
    ptr2 = tmp;
    *ptr2 = 100; }
```

When **test()** is called, we have **ptr1 = &i** and **ptr2 = &j**.
The statements **ptr1 = &m;** and ***ptr1 = 100;** make **m=120**.
Since **ptr2** points to the address of **j**, ***ptr2 is 20.**
Thus, the statement ***ptr2 += m;** makes **j=20+100=120**.
Since **tmp** points to the address of **i**, the **ptr2=tmp;**
is equivalent to **ptr2 = &i.**
Thus, the statement ***ptr2 = 100;** changes the value
of **i** to **100**. As a result, the program outputs: **100 120**

Thank you

Lecture 11

Problem solving using functions in C

Objectives

- Problem solving using functions in C.
- Defining output of C programs.

Example 1:

Write two functions to take an integer parameter and to return the square and the cube of this number, respectively. Write a C program to read an integer and to use the functions to display the sum of the square and cube of the number.

Example 1:

```
#include <stdio.h>
int square(int a);
int cube(int a);
int main(void)
{
    int i, j, k;
    printf("Enter number: ");
    scanf("%d", &i);
    j = square(i);
    k = cube(i);
    printf("sum = %d\n", j+k); //Without declaring the j and k variables, we
write printf("sum = %d\n", square(i)+cube(i));
    return 0;
}
int square(int a)
{
    return a*a;
}
int cube(int a)
{
    return a*a*a;
}
```

Example 2:

Write a function that takes as parameters three values and returns the minimum of them. Write a C program to read the grades of three students and use the function to display the lowest grade.

Example 2:

```
#include <stdio.h>
float min(float a, float b, float c);
int main(void){
    float i, j, k;
    printf("Enter grades: ");
    scanf("%f%f%f", &i, &j, &k);
    printf("Lowest grade = %.2f\n", min(i, j,
k));
    return 0;}
float min(float a, float b, float c){
    if(a <= b && a <= c)
        return a;
    else if(b < a && b < c)
        return b;
    else
        return c;}
```

Example 3:

What is the output of the following C program?

```
#include <stdio.h>
void test(int *ptr1, int *ptr2);
int main(void){
    int i = 10, j = 20;
    test(&i, &j);
    printf("%d %d\n", i, j);
    return 0; }
void test(int *ptr1, int *ptr2) {
    int m, *tmp;
    tmp = ptr1;
    ptr1 = &m;
    *ptr1 = 100;
    *ptr2 += m;
    ptr2 = tmp;
    *ptr2 = 100; }
```

Example 3:

Solution:

When **test()** is called,

We obtain **ptr1 = &i** and **ptr2 = &j**.

The statements

ptr1 = &m; and ***ptr1 = 100;**

make m equal to 100. Since **ptr2** points to the address of j,

***ptr2** is 20.

Thus, the statement ***ptr2 += m;**

makes j equal to **20+100 = 120**.

Since **tmp** points to the address of i,

the statement **ptr2 = tmp;** is

equivalent to **ptr2 = &i**.

Thus,

the statement ***ptr2 = 100;**

changes the value of i to **100**.

As a result, the program outputs: **100 120**

Example 4:

C program that uses a pointer arithmetic

```
unsigned int test(const char *str){  
    const char *ptr = str;  
    while(*str++) //Equivalent to while(*str++ != '\0')  
        ;  
    return str - ptr - 1;}
```

In each iteration, the loop compares the value of `*str` with 0, equally to '`\0`', when `str` is increased, to point to the next array element.

In the first iteration, `*str` is equal to `str[0]`, then it equal to `str[1]`, and so on. Once `*str` becomes equal to the **null** character, the loop terminates.

Recall the **pointer arithmetic** that the result of the subtraction of two pointers that point to the same array is the number of elements, so, in this example, the number of characters between them.

`ptr` points to the first element, while `str` after its last increase points to the next place following the **null** character.

That's why we put **-1**, to subtract this place.

Example 5:

Use the function `power(double a, int b);` to return the result of a^b . Write a C program that reads a float number (a) and an integer (b) and that uses the function to display the result of a^b .

```
#include <stdio.h>
double power(double a, int b);
int main(void){
    int exp;
    double base;
    printf("Enter base and exponent: ");
    scanf("%lf%d", &base, &exp);
    printf("%.2f power %d = %.2f\n", base, exp, power(base,
exp));
    return 0; }
double power(double a, int b) {
    int i, exp;
    double value;
    value = 1;          //Necessary initialization
    exp = b;
    if(exp < 0)         //If the exponent is negative, make it positive
        exp = -exp;
    for(i = 0; i < exp; i++)
        value *= a;
    if(b < 0)
        value = 1/value;
    return value; }
```

Example 5

Example 6:

Write a void function that generates a random number from 0 to 1 with two decimal digits and uses a proper parameter to return it. Write a C program that calls the function and displays the return value.

Example 6

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void make_rand(double *p);
int main(void){
    double d;
    make_rand(&d);
    printf("%.2f\n", d);
    return 0;
}
void make_rand(double *p) {
    int i;
    srand(time(NULL)); //It is reminded that srand() together with
time() are used to generate random positive integers, each time the program
runs.
    i = rand()%101; //rand() returns a random integer which is
constrained in [0, 100]
    *p = i/100.0; } //This division constrains the value within [0, 1]
with two decimal digits.
```

Example 7:

Write a **void** function that takes as parameters the coefficients of a quadratic trinomial and returns its real roots, if any. Write a program that reads the coefficients of a trinomial (e.g., a, b, and c) and uses the function to solve the equation. The program is to make the user to enter a nonzero value for a. The roots of the trinomial are $ax^2 + bx + c$ with $a \neq 0$ and the discriminant $D = b^2 - 4ac$ is to be tested.

If $D > 0$, it has two real roots $r_{1,2} = (-b \pm \sqrt{D}) / 2a$.

If $D = 0$, it has one real double root $r = -b / 2a$.

If $D < 0$, it has no real root.

To calculate the square root, use the **sqrt()** function, as shown in **math library**.

```
#include <stdio.h>                                         Example 7
#include <math.h>
void find_roots(double a, double b, double c, double*r1,
                double*r2, int*code);
int main(void) {
    int code;
    double a, b, c, r1, r2;
    do
    {
        printf("Enter coefficients (a<>0): ");
        scanf("%lf%lf%lf", &a, &b, &c);
    }
    while(a == 0);
    find_roots(a, b, c, &r1, &r2, &code);
    if(code == 2)
        printf("Two roots: %.2f %.2f\n", r1, r2);
    else if(code == 1)
        printf("One root: %.2f\n", r1);
    else
        printf("Not real roots\n");
    return 0; }
```

```
void find_roots(double a, double b, double c, double*r1,
double*r2, int*code)
{
    double d;
    d = b*b-4*a*c;
    if(d > 0)
    {
        *code = 2;
        *r1 = (-b+sqrt(d))/(2*a);
        *r2 = (-b-sqrt(d))/(2*a);
    }
    else if(d == 0)
    {
        *code = 1;
        *r1 = *r2 = -b/(2*a);
    }
    else
        *code = 0;
}
```

```
1 -2 -4
9 12  4
3   4   2
```

Example 8:

What will be the output of the following C program?

```
#include <stdio.h>
#include <string.h>
int main(void){
    char str1[10], str2[] = "engine";
    printf(" %c\n", ++str1[strcmp(strcpy(str1, "ine"),
str2+3)+2]);
    return 0; }
```

One dimensional array as argument

An array is a data structure that contains a number of values, or else elements, of the same type.

Each element can be accessed by its position within the array.

data_type array_name[number_of_elements];

The **number_of_elements**, or else the length of the array, is specified by an integer constant expression **> 0** enclosed in **[]**

If the length of the array is used several times in the program, a good practice is to use a macro instead. If there is need to change the array, one just changes the macro.

For example:

```
#define SIZE 150float arr[SIZE]; //The compiler replaces SIZE with 150 and creates an array of 150 floats.
```

Unlike macros, it is an error to use a **const** constant to specify the length.

```
const int size = 150;
```

```
float arr[size]; //Illegal
```

One dimensional array as argument:

```
void test(int arr[]);
```

When passing an array to a function, one writes only its name, without brackets.

For example:

```
test(arr);
```

When an array name is passed to a function, it is always treated as a pointer. The passing argument is the memory address of its first element, not a copy of the array itself.

Since no copy of the array is made, the time required to pass an array to a function does not depend on the size of the array.

One dimensional array as argument, cont.,

When an ordinary variable is passed to a function, the function works with a copy.

If an array is passed, no copy of the array is made; the function works with the original.

Why is that decision taken?

To improve performance, since it could be expensive in memory and time to copy an entire array. It is just a pointer to its first element that is passed and the function may access any of its elements.

Example 9

Write a function that takes as parameters an array containing the students' grades in a test and two grades (e.g., a and b) and returns the average of the grades within [a, b]. Write a program that reads the grades of 50 students and the two grades a and b and uses the function to display the average. The program should force the user to enter a value for a less than or equal to b.

Example 9

```
#include <stdio.h>
#define SIZE 2
float avg_arr(float arr[], float min, float max);
int main(void){
    int i;
    float a, b, k, arr[SIZE];
    for(i = 0; i < SIZE; i++) {
        printf("Enter grade: ");
        scanf("%f", &arr[i]); }
    do {
        printf("Enter min and max grades: ");
        scanf("%f%f", &a, &b); }
        while(a > b);
    k = avg_arr(arr, a, b);
    if(k == -1)
        printf("None grade in [%f, %f]\n", a, b);
    else
        printf("Avg = %.2f\n", k);
return 0; }
```

Example 9, cont.,

```
float avg_arr(float arr[], float min, float max) {  
    int i, cnt = 0;  
    float sum = 0;  
    for(i = 0; i < SIZE; i++) {  
        if(arr[i] >= min && arr[i] <= max) {  
            cnt++;  
            sum += arr[i]; } }  
    if(cnt == 0)  
        return -1;  
    else  
        return sum/cnt; }
```

Example 10:

Does the following program contain any error? If not, what does it output?

```
#include <stdio.h>
#include <string.h>
char *test(void);
int main(void) {
    char ptr[100] = "sample";
    strcpy(ptr, test());
    printf(" %s\n", ptr);
    return 0; }
char *test(void){
    char str[] = "example";
    return str; }
```

Example 10, cont.,

Solution:

When **test()** is called, **the compiler** allocates memory for the **str** array and **stores the string into it**. This **memory location** is returned. The memory of a **local variable** is released when the function terminates.

Most probably, the program will not display example.

But even if it is displayed, the code is erroneous.

Remember, do not return the address of a local variable unless it is declared as static.

Thank you

Lecture 12

Problem solving using arrays,
pointers, and functions in C

Objectives

- Strings
- Arrays
- Pointers
- Defining output of C programs
- Problem solving.

Strings:

- A string can be stored in a variable - an array of characters.
- A string ends with null character, to store a string of N characters, the size of the array is N+1.
- To declare an array that stores a string of up to 5 characters:

char str[6]; An array can be initialized with a string, when it is declared.

char str[6] = "table"; the compiler copies the characters of the "table" into the **str** array and adds the null character.

str[0] becomes 't', **str[1]** becomes 'a', and the value of the last element **str[6]** becomes '\0'.

char str[6] = {'t', 'a', 'b', 'l', 'e', '\0'};

Strings:

- The *null character* '\0' is one character.
 - The ASCII code of the *null character* is 0.
 - The \ character denotes an escape sequence.
- The ASCII code of the zero character is 48.
A common error is to confuse the '\0' and '0' characters.

Strings and arrays:

--- If the number of the characters in an array is less than the size of the array, the remaining elements are initialized to 0. *If it is greater, it is an error.*

str[0] becomes 't',

str[1] becomes 'a' and the rest elements are initialized to 0, or equivalently to '\0'.

--- **char str[8] = {0};** all **str** elements are initialized to '\0'.

--- To initialize the array is to omit its length and to let the compiler to compute it.

--- **char str[] = "table";** the compiler calculates the length of "table" and then allocates 6 bytes for **str** to store the 5 characters plus the null character.

--- **sizeof(str)** outputs its length (Leaving the compiler to compute the length, it is easier and safer)

Strings and arrays:

- By declaring an array - reserve an extra place for the null character.
- If **null** character is missing, the program uses a library function to handle the array, unpredictable results may arise because C functions assume that strings are null terminated.
- Reserve ***null character*** by adding 1 in the declaration:

```
char s[SIZE+1];
```

```
char s[] = "abc"; //size of s is 4, null character is stored – a string
```

```
char s[3] = "abc"; //size of s is 3 - no string.
```

Pointers:

--- In a computer with N bytes, the memory address of each cell is a unique number from 0 to N-1.

--- When a variable is declared, the compiler reserves the required consecutive bytes to store its value. If a variable occupies more than one byte, the variable's address is the address of the first byte:

int a = 10; the compiler reserves four consecutive unused bytes and stores the value **10** into the addresses **5000-5003**, assuming that the less significant byte of the value is stored in the lower address byte.

--- The compiler associates name of a variable with its memory address.

--- When the variable is used in the program, the compiler accesses its address.

--- There is a statement **a = 80;** the compiler knows the memory address of **a** is **5000** and sets its content to **80.**

Pointers declaration:

--- A pointer variable is a variable that holds a memory address.

```
data_type *pointer_name;
```

```
int *ptr;
```

ptr can hold the memory address of an **int** variable

```
int *ptr, i, j, k;
```

Pointers declaration, cont.,

int * ptr; declaration of a pointer

It might be confusing when multiple variables are declared such as:

int* p1, p2;

p1 is declared as pointer, p2 is not.

Pointers declaration, cont.,

The size of the operator is used to find out how many bytes **ptr** allocates:

```
#include <stdio.h>
int main(void)
{
    int *ptr;
    printf("Bytes: %u\n", sizeof(ptr));
    return 0;
}
```

Pointers declaration, cont.,

The size of the operator is used to find out how many bytes **ptr** allocates:

```
#include <stdio.h>
int main(void){
    int *ptr;
    printf("Bytes: %u\n", sizeof(ptr));
    return 0;
}
```

A pointer variable allocates the same size, no matter what data type it points to.

char*ptr; or

float *ptr; or

double *ptr; the output would be the same, that is, 4.

Pointers initialization:

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;
    ptr = &a;          //ptr "points to" the memory address of a
    printf("Address=%p\n", ptr); //Display the memory
address of a
    return 0;
}
```

Example:

```
int a, b, *ptr = &a;
```

Pointers initialization, cont.,

- It is better to initialize the pointer variables in separate statements, not together with their declarations. Due to lack of space, sometimes it is done in one line.
- The address operator & is applied only to objects in memory, such as variables, array elements, and functions.

If the type is T the type of $\&T$ is “pointer to T . ”

Null pointers:

```
#include <stdio.h>
int main(void){
int *ptr;
printf("Addr = %p\n", ptr);
ptr = NULL;
printf("Addr = %p\n", ptr);
return 0;}
```

Null pointers:

A pointer assigned the NULL value points to nothing and becomes a null pointer.

The program **first outputs the initial value of the pointer and then 0**.

```
#include <stdio.h>
int main(void){
    int *ptr;
    printf("Addr = %p\n", ptr);
    ptr = NULL;
    printf("Addr = %p\n", ptr);
    return 0; }
```

Null pointers :

The value of a pointer variable can be compared with NULL:

```
if(ptr != NULL) //Equivalent to if(ptr)
if(ptr == NULL) //Equivalent to if(!ptr)
if (!ptr)        //not preferred syntax
```

Many functions return NULL to indicate that the execution failed.

Using of * for pointers :

To access the content of a memory address referenced by a pointer variable we use the *** (*indirection* or *dereference*) operator before its name.

Example:

Read the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;
    a = 10;
    ptr = &a;
    printf(" value = %d\n", *ptr);
    return 0;
}
```

Example of using a pointer :

```
#include <stdio.h>
int main(void)
{
    int *ptr, a;
    a = *ptr;
    printf("value = %d\n", a);
    return 0;
}
```

Example:

Read the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i, j;
    j = 20;
    ptr = &j;
    i = *ptr;
    printf("value = %d\n", i);
    return 0;
}
```

Example:

Read the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr, i;
    ptr = &i;
    printf("%p %p %p\n", &i, *&ptr,
&*ptr);
    return 0;
}
```

Pointers and Two-Dimensional (2D) arrays:

The elements:

str[0], str[1], ..., str[N-1] of a two-dimensional array.

str[N][M] is treated as a pointer to an array of M elements.

str[0] can be used as a pointer to an array of 40 characters, which holds the string One.

In order to save memory space, we could use an array of pointers instead of a two-dimensional array such as:

```
char *str[] = {"One", "Two", "Three"};
```

Although some memory is allocated for the **str** pointers, the important thing is that the memory allocated for the strings is exactly as needed (close relationship between pointers and arrays) we can subscript **str** as a two-dimensional array to access a character.

Pointers and 2D arrays:

```
for(i = 0; i < 3; i++)
if(str[i][0] == 'T')
printf("%s\n", str[i]);
```

Pointers and 2D arrays:

Read the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr1, *ptr2, *ptr3, i = 10, j = 20, k = 30;
    ptr1 = &i;
    i = 100;
    ptr2 = &j;
    j = *ptr2 + *ptr1;
    ptr3 = &k;
    k = *ptr3 + *ptr2;
    printf("%d %d %d\n", *ptr1, *ptr2, *ptr3);
    return 0;
}
```

Example:

Write a C program that uses two pointers to read two float numbers first and then to swap the values they point to. Then, use the same pointers to output the greater value.

Example, cont.,

```
#include <stdio.h>
int main(void)
{
    float *ptr1, *ptr2, i, j, tmp;
//The pointers should be initialized before used in scanf()
    ptr1 = &i;
    ptr2 = &j;
    printf("Enter values: ");
    scanf("%f%f", ptr1, ptr2); //Store the input values in the
addresses pointed to by the pointers.
    tmp = *ptr2;
    *ptr2 = *ptr1;
    *ptr1 = tmp;
    if(*ptr1 > *ptr2)
        printf("%f\n", *ptr1);
    else
        printf("%f\n", *ptr2);
    return 0;
}
```

Example, cont.,

Read the output of the following program?

```
#include <stdio.h>
int main(void)
{
    int *ptr1, i = 10;
    double *ptr2, j = 1.234;
    ptr1 = &i;
    ptr2 = &j;
    *ptr1 = *ptr2;
    printf(" %d %u %u %u\n", i, sizeof(ptr1),
           sizeof(ptr2), sizeof(*ptr2));
    return 0;
}
```

Thank you

Lecture 13

Problem solving using arrays,
pointers, and functions

Objectives

- Functions
- Arrays
- Pointers
- Characters
- Writing strings
- Defining output of C programs
- Problem solving.

Example 1:

A C program subscripts a pointer **ptr** as though it were an array to display the values and the addresses of **arr** elements.

arr[i] and **ptr[i]** access the same element. To access **arr[2]**, it gets the address of **arr**, adds 8 (assume the size of **int** is 4 bytes), and goes to the resulting address to get the value. In other words, **arr[2]** resides 8 places after the address of **arr**, while **ptr[2]** resides 8 places after the address that **ptr** points to.

```
#include <stdio.h>
int main(void) {
    int *ptr, i, arr[5] = {10, 20, 30, 40, 50};
    ptr = arr;
    for(i = 0; i < 5; i++)
printf("Addr=%p  Value=%d\n", &ptr[i], ptr[i]);
    return 0; }
```

Example 2:

Declare a pointer ***ptr**,

Declare **arr[]** with initial values of 10, 20, 30, 40, 50.

Create **ptr = arr;**

ptr points to the address of **arr[0]**,

***ptr** is equal to **arr[0]**.

Make ***ptr = 3;** to be equivalent to **arr[0] = 3;**

Use the statement **ptr += 2;** to make **ptr** equal to the address of **arr[2]**, so that ***ptr == arr[2]**.

Next make ***ptr = 5;** be equivalent to **arr[2] = 5;**

Write the C program and explain the output of the program.

```
#include <stdio.h>
int main(void)
{
    int *ptr, arr[] = {10, 20, 30, 40, 50};
    ptr = arr;
    *ptr = 3;
    ptr += 2;
    *ptr = 5;
    printf(" %d\n", arr[0]+arr[2]);
    return 0; }
```

Example 3:

Write a C program for two pointers to point to the same array and the two pointers are subtracted, the result is the number of elements between them, *not their distance in memory*.

```
#include <stdio.h>
int main(void)
{
    int i, *ptr1, *ptr2, arr[] = {10, 20, 30,
40, 50, 60, 70};
    ptr1 = &arr[2];
    ptr2 = &arr[4];
    for(i = ptr2 - ptr1; i < 5; i+=2)
        printf(" %d ", ptr1[i]);
return 0; }
```

Example 4:

Use the `p` pointer and a `while` loop, and complete the following program to display the integers from 1 to 10.

```
#include <stdio.h>
int main(void)
{
    int *p, i;
    ...
    ...
}
```

Solution:

```
#include <stdio.h>
int main(void)
{
    int *p, i; //Given
    p = &i;
    *p = 1;
    while(*p <= 10)
    {
        printf("%d\n", *p);
        (*p)++; //parentheses are used for priority reasons
    }
    return 0; }
```

void * pointer

A pointer of type **void** is a generic pointer in the sense that it can point to a variable of any type. Any pointer can be cast to **void *** and back again to the original type without any loss of information.

void * is used to assign a pointer of one type to a pointer of another type, cast is necessary.

void * pointer

To access a variable using a **void * pointer**, cast is necessary in order to inform the compiler about the type of the variable.

```
#include <stdio.h>
int main(void) {
    void *ptr;
    char s[] = "abcd";
    int i = 10;
    ptr = &i;
    *(int*)ptr += 20;
    printf("%d\n", i);
    ptr = s+2;
    (*(char*)ptr)++;
    printf("%s\n", s);
    return 0; }
```

Subtracting and comparing pointers

- The result of subtracting two pointers is the number of elements between them. The pointers point to the elements of the same array or in the next place right after the end of the array.
- The type of the result is implementation dependent, it is defined as `ptrdiff_t` (typically `int`) in the **C library**.

For example: if **p1** points to the second element and **p2** points to the fifth element of the same array, the result of **p2-p1** is 3, while the result of **p1-p2** is -3.

- The result of comparing two pointers with the `==` and `!=` operators is reliable. Using the relational operators `<`, `<=`, `>`, and `>=` the result is valid if both point to parts of the same object (e.g., array, structure); otherwise, it is undefined.

For example: the result of **p2 > p1** is 1.

Pointer arithmetic

- Pointer arithmetic refers to the application of some arithmetic operations on pointers such as: adding or subtracting a pointer and an integer and subtracting and comparing pointers provided that they point to the same array.
- Assignment or comparison with 0 is also allowed.
- Pointer arithmetic can produce reliable results when applied to elements of the same array; otherwise, the behavior is undefined.
- An exception to that rule is the address of the first element past the end of the array.

Pointers and integers

Assume a pointer **ptr** points to an element of an array.

The addition of a positive integer **n** in an assignment such as:

ptr = ptr + n; increases the value of the pointer by **n * size** of the data type and makes the pointer point to the address of the **nth** element after the one it points to.

For example:

if **ptr** points to the array element **arr[i]** and **n** is 3,

ptr will point to **arr[i+3]**. If the resulting element is out of the array, with the exception of the first element past the end, the result is *undefined*.

Consider the C program:

```
#include <stdio.h>
int main(void)
{
    int *ptr, arr[] = {10, 20, 30};
    ptr = &arr[0];
    printf("Addr:%d\n", ptr);
    ptr = ptr+2;
    printf("Addr:%d Value is %d\n", ptr, *ptr);
    return 0;
}
```

Similar to addition, subtracting a positive integer from a pointer in an assignment such as:

ptr = ptr-n; decreases the pointer's value by **n * size** of its type and makes the pointer point to the address of the **nth** element before the one it points to.

For example:

We replace the code in the previous example with this one:

```
ptr = &arr[2];
printf("Addr:%d\n", ptr);
ptr = ptr-2;
printf("Addr:%d\n", ptr);
```

Example 5:

Use the p2 pointer and complete the following program to read students' grades continuously until the user enters -1. Use the p1 pointer to display how many students got a grade within [5, 10] and the p3 pointer to display the best grade.

```
#include <stdio.h>
int main(void){
    int *p1, sum;
    float *p2, *p3, grade, max;
    ...
}
```

...

Solution:

```
#include <stdio.h>
int main(void){
    int *p1, sum;
    float *p2, *p3, grade, max;
    p1 = &sum;
    *p1 = 0;
    p3 = &max;
    *p3 = 0;
    p2 = &grade;
    while(1) {
        printf("Enter grade: ");
        scanf("%f", p2);
        if(*p2 == -1)
            break;
        if(*p2 >= 5 && *p2 <= 10){
            (*p1)++;
            if(*p2 > *p3)
                *p3 = *p2; }
    }
    printf("%d students passed (max = %.2f)\n", *p1, *p3);
    return 0; }
```

Thank you

Lecture 14

Problem solving using arrays,
pointers, and strings

Objectives

- Pointers
- Characters
- Writing strings
- Problem solving.

Subtracting and comparing pointers

- The result of subtracting two pointers is the number of elements between them. The pointers point to the elements of the same array or in the next place right after the end of the array.

For example: if **p1** points to the second element and **p2** points to the fifth element of the same array, the result of **p2-p1** is 3, while the result of **p1-p2** is -3.

- The result of comparing two pointers with the == and != operators is reliable. Using the relational operators <, <=, >, and >= the result is valid if both point to parts of the same object (e.g., array, structure); otherwise, it is undefined.

For example: the result of **p2 > p1** is 1.

Pointer arithmetic

- Pointer arithmetic refers to the application of some arithmetic operations on pointers such as: adding or subtracting a pointer and an integer and subtracting and comparing pointers provided that they point to the same array.
- Assignment or comparison with 0 is also allowed.
- Pointer arithmetic can produce reliable results when applied to elements of the same array; otherwise, the behavior is undefined.
- An exception to that rule is the address of the first element past the end of the array.

Pointers and integers

Assume a pointer **ptr** points to an element of an array. The addition of a positive integer **n** in an assignment such as:

ptr = ptr + n; increases the value of the pointer by **n * size** of the data type and makes the pointer point to the address of the **nth** element after the one it points to.

For example:

if **ptr** points to the array element **arr[i]** and **n** is 3,

ptr will point to **arr[i+3]**. If the resulting element is out of the array, with the exception of the first element past the end, the result is *undefined*.

If **ptr** is declared as a pointer to an array of:

char, its value is increased by **n*1=n**, the size of **char** type is 1B.

int or **float**, its value is increased by **n*4**, the size of both types =4B.

double, its value is increased by **n*8**, the size of **double** = 8B.

Consider the C program:

```
#include <stdio.h>
int main(void) {
    int *ptr, arr[] = {10, 20, 30};
    ptr = &arr[0];
    printf("Addr:%d\n", ptr);
    ptr = ptr+2;
    printf("Addr:%d Value is %d\n", ptr, *ptr);
    return 0; }
```

Subtracting a positive integer from a pointer in an assignment such as:

ptr = ptr-n; decreases the pointer's value by *n * size* of its type and makes the pointer point to the address of the **nth** element before the one it points to.

For example:

We replace the code in the previous example with this one:

```
ptr = &arr[2];
printf("Addr:%d\n", ptr);
ptr = ptr-2;
printf("Addr:%d\n", ptr);
```

Use three pointers such that a p2 pointer in a C program reads students' grades continuously until the user enters -1. Use p1 pointer to display the number of students who received a grade within [5, 10] and use a p3 pointer to display the best grade.

```
#include <stdio.h>
int main(void){
    int *p1, sum;
    float *p2, *p3, grade, max;
    p1 = &sum;
    *p1 = 0;
    p3 = &max;
    *p3 = 0;
    p2 = &grade;
    while(1) {
        printf("Enter grade: ");
        scanf("%f", p2);
        if(*p2 == -1)
            break;
        if(*p2 >= 5 && *p2 <= 10){
            (*p1)++;
            if(*p2 > *p3)
                *p3 = *p2; }
    }
    printf("%d students passed (max = %.2f)\n", *p1, *p3);
    return 0; }
```

Array of pointers to functions:

The following statement reads two integers, checks the first one, and uses a function pointer to call the respective function. The program displays the return value.

An array of function pointers can be initialized when declared, just like an ordinary array.

```
int (*ptr[3])(int a, int b) = {test_1, test_2, test_3};
```

```
#include <stdio.h>
int test_1(int a, int b);
int test_2(int a, int b);
int test_3(int a, int b);
int main(void) {
    int (*ptr[3])(int a, int b);
    int i, j, k;
    ptr[0] = test_1;//ptr[0] points to memory address of test_1()
    ptr[1] = test_2;
    ptr[2] = test_3;
    printf("Enter numbers: ");
    scanf("%d%d", &i, &j);
```

```
if(i > 0 && i < 10)
    k = ptr[0](i, j); //Call function that ptr[0] points to.
Optional :k = (*ptr[0])(i, j)
else if(i >= 10 && i < 20)
    k = ptr[1](i, j); //Call function that ptr[1] points to.
else
    k = ptr[2](i, j); //Call the function that ptr[2] points to.
printf("Val = %d\n", k);
return 0; }
int test_1(int a, int b) {
    return a+b; }
int test_2(int a, int b) {
    return a-b; }
int test_3(int a, int b) {
    return a*b; }
```

Characters:

```
char ch = 'c';
int i;
ch++; //ch becomes 'd'.
ch = 68; //ch becomes 'D'.
i = ch-3; //i becomes 'A', that is, 65.
```

To test if a character is a digit 0-9 we write

```
if(ch >= '0' && ch <= '9')
```

and **not** `if(ch >= 0 && ch <= 9)`

Characters 1:

```
#include <stdio.h>
int main(void) {
    char ch;
    int i;
    printf("Enter number: ");
    scanf("%d", &i);
    printf("Enter character: ");
    scanf("%c", &ch);
    printf("Int = %d and Char = %c\n", i, ch);
    return 0; }
```

Characters 2:

```
#include <stdio.h>
int main(void) {
    char ch;
    int i;
    printf("Enter number: ");
    scanf("%d", &i);
    printf("Enter character: ");
    scanf(" %c", &ch);
    printf("Int = %d and Char = %c\n", i, ch);
    return 0; }
```

Writing strings 1:

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("%s\n", str);
    return 0;}
```

Writing strings 2:

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("%s\n", str+5);
    return 0;}
```

In order to display the part beginning from the sixth character, that is, *is text*, we write `printf("%s\n", str+5);` or equivalently using `printf("%s\n", &str[5]);`

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("%s\n", &str[5]);
    return 0;}
```

Writing strings 3:

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("This text is displayed \on the
same line\n");
    return 0;}
```

Writing strings 4:

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("This text is displayed "
"on the same line\n");
    return 0;
}
```

Writing strings 5:

```
#include <stdio.h>
int main(void){
    char str[] = "This is text";
    printf("\\"Test\"");
    return 0;
}
```

Writing strings 6:

```
#include <stdio.h>
int main(void)
{
    char str[] = "SampleText";
    str[4] = '\0';
    printf("%s\n", str);
    return 0;
}
```

Writing strings 7:

```
#include <stdio.h>
int main(void) {
    char str[] = "This is text";
    puts(str);
    return 0; }
```

Writing strings 8:

```
#include <stdio.h>
int main(void)
{
    char str[100];
    fgets(str, sizeof(str), stdin);
    printf(str);
    return 0;
}
```

Write strings 9:

```
#include <stdio.h>
int main(void) {
    char str[100];
    int num;
    printf("Enter number: ");
    scanf("%d", &num);
    printf("Enter text: ");
    fgets(str, sizeof(str), stdin);
    printf("%d %s\n", num, str);
    return 0;
}
```

Write strings 10:

```
#include <stdio.h>
int main(void) {
    char *p, *q, s[] = "play";
    p = s+1;
    q = s;
    p[1] = 'x';
    *s = 'a';
    printf(" %d %c\n", *q+2, *(q+2));
    return 0; }
```

Write strings 11:

```
#include <stdio.h>
int main(void)
{
    char *str = "this";
    for(; *str; printf("%s ", str++));
    return 0;
}
```

Example 3:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[10] = {0};
printf(" %c\n", *(str+strlen(strncpy(str,
"sample1", 6))/2));
    return 0;
}
```

Example 4:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
char str[30];

return !printf("%s\n", strcat(strcpy(strcpy(str,
"Water")+5, "melon ref"), "reshment")-5));
}
```

Example 5:

The characters of "Monday" are stored in the first row of arr ,
the characters of "Tuesday" are stored in the second row, and so on.

***(arr[i]+3)** is equivalent to **arr[i][3]** and

***(*(arr+i)+4)** is equivalent to ***(arr[i]+4)**, that is, **arr[i][4]**.

Make the loop to check each row of arr and to display the strings whose third, fourth, and fifth characters are 'n', 'd' and 'a', respectively.

Create a C program to display:

Monday is No.1 week day

Sunday is No.7 week day

Solution 5:

```
#include <stdio.h>
int main(void)
{
char arr[7][10] = {"Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday",
"Sunday"};
int i;
for(i = 0; i < 7; i++)
if(arr[i][2] == 'n' && *(arr[i]+3) == 'd' &&
*(arr+i+4) == 'a')
    printf("%s is No.%d week day\n", arr[i], i+1);
return 0;
}
```

Thank you

Lecture 15

Problem solving using arrays,
pointers, and recursive functions

Objectives

- Recursive function
- Searching sorted and unsorted arrays
 - Linear search algorithm
 - Binary search algorithm
- Problem solving.

Example 1:

```
#include <stdio.h>
void show(int num);
int main(void)
{ int i;
    printf("Enter a number: ");
    scanf(" %d", &i);
    show(i);
    return 0; }
void show(int num) {
    if(num > 1)
        show(num-1);
    printf("value = %d\n", num); }
```

Recursive function:

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

    printf("Enter a number: ");
    scanf(" %d", &i);
    show(i);
    return 0;
}
void show(int num)
{
    if(num > 1)
        show(num-1);
    printf("value = %d\n", num);
}
```

Example 2:

Assignment: `main()` can be called recursively. In each call, `a` is decremented by 1. The program stops calling `main()` once `a` becomes 0.

The output displays: 6 5 4 3 2 1

```
#include <stdio.h>
int a = 6;
int main(void)  {
    if(a == 0)
        return 0;
    else
    {
        printf(" %d", a--);
        main();
    }
    return 0;  }
```

Example 3:

Write a recursive function that takes as parameter an integer **n** and returns its factorial, using the formula

$$n! = n \times (n-1)!$$

The factorial of an integer n , where $n \geq 1$, is the product of the integers from **1** to **n**,
that is,

$$1 \times 2 \times 3 \times \dots \times n$$

The factorial of **0** is **1 (0!=1)**.

Write a program that reads a positive integer less than **150** and uses the function to display its factorial.

Solution to example 3:

```
#include <stdio.h>
double fact(int num);
int main(void) {
    int num;
    do {
        printf("Enter a positive integer less than 170: ");
        scanf("%d", &num); }
    while(num < 0 || num > 150);
    printf("Factorial of %d is %e\n", num, fact(num));
    return 0; }
double fact(int num) {
    if((num == 0) || (num == 1))
        return 1;
    else
        return num * fact(num-1); }
```

Example 4:

Write a C program to calculate the product of two input numbers through the use of a recursive function.

Use of recursive function:

- Create a function with two arguments
- Assume that it is called with arguments 10 and 4
- The function returns:

$$\begin{aligned} n1 + \text{unknown}(n1, n2-1 = 3) &= \\ &= n1 + n1 + \text{unknown}(n1, n2-1 = 2) = \\ &= n1 + n1 + n1 + \text{unknown}(n1, n2-1 = 1) \end{aligned}$$

- The last call of $\text{unknown}(n1, 1)$ returns $n1$, because $n2 = 1$.
- Therefore, the function returns:

$$n1+n1+n1+n1 = 4*n1 = n2*n1$$

- if - else - if series is used to find the sign of the product and the absolute values of the integers.
- Next if - else statement is used, in order to make the less recursive calls.

Solution to example 4:

```
#include <stdio.h>
int unknown(int num1, int num2);
int main(void){
    int num1, num2, sign;
    printf("Enter numbers: ");
    scanf("%d%d", &num1, &num2);
    sign = 1;
    if((num1 < 0) && (num2 > 0)) {
        num1 = -num1;
        sign = -1; }
    else if((num1 > 0) && (num2 < 0)) {
        num2 = -num2;
        sign = -1; }
    else if((num1 < 0) && (num2 < 0)) {
        num1 = -num1;
        num2 = -num2; }

    if(num1 > num2)
        printf("%d\n", sign*unknown(num1, num2));
    else
        printf("%d\n", sign*unknown(num2, num1));
    return 0; }

int unknown(int n1, int n2) {
    if(n2 == 1)
        return n1;
    else
        return n1 + unknown(n1, n2-1); }
```

To do for practicing:

Write a function that searches for a number in an array of doubles. If the number is stored, the function should return the number of its occurrences and the position of its first occurrence, otherwise -1. Write a program that reads up to 100 doubles and stores them in an array. If the user enters -1, the insertion of numbers should terminate. Then, the program should read a double number and use the function to display its occurrences and the position of its first occurrence in the array.

Use of a *Linear Search Algorithm* to solve the problem

- The algorithm is to search for a value in a *non-sorted* array.
- The searched value is compared with the value of each element until a match is found.
- In an array of **n** elements, maximum number of searches is **n**. when the searched value is not found or is equal to the last element.
- **linear_search()** function is used to implement the *linear search algorithm* or it is also known as a *sequential search*.

Solution to example 5: (LSA)

```
#include <stdio.h>
#define SIZE 5
int linear_search(double arr[], int size, double num, int *t);
int main(void){
    int i, times, pos;
    double num, arr[SIZE];
    for(i = 0; i < SIZE; i++){
        printf("Enter number: ");
        scanf("%lf", &num);
        if(num == -1)
            break;
        arr[i] = num; }
    printf("Enter number to search: ");
    scanf("%lf", &num);
pos = linear_search(arr, i, num, &times); //The variable i indicates the number of the
array's elements.
    if(pos == -1)
        printf("%f isn't found\n", num);
    else
        printf("%f appears %d times (first pos = %d)\n", num, times, pos);
return 0;}
int linear_search(double arr[], int size, double num, int *t){
    int i, pos;
    pos = -1;
    *t = 0;
    for(i = 0; i < size; i++){
        if(arr[i] == num){
            (*t)++;
            if(pos == -1) //Stores position of 1st occurrence.
                pos = i; } }
return pos; } cont. with explanation about the bynary serach alogorithmm
```

Example 6: (BSA)

Write a function that searches for a number in an array of integers. If the number is found, the function should return its position, otherwise -1. Write a program that initializes an array of integers with values are sorted in ascending order. The program should read an integer and use the function to display its position in the array.

Solution to example 6: (BSA)

```
#include <stdio.h>
int binary_search(int arr[], int size, int num);
int main(void) {
    int num, pos, arr[] = {10, 20, 30, 40, 50, 60, 70};
    printf("Enter number to search: ");
    scanf("%d", &num);
    pos = binary_search(arr, 7, num);
    if(pos == -1)
        printf("%d isn't found\n", num);
    else
        printf("%d is found in position %d\n", num, pos);
    return 0; }

int binary_search(int arr[], int size, int num) {
    int start, end, middle;
    start = 0;
    end = size-1;
    while(start <= end) {
        middle = (start+end)/2;
        if(num < arr[middle])
            end = middle-1;
        else if(num > arr[middle])
            start = middle+1;
        else
            return middle; }
    return -1; } //If the execution reaches this point it means that the number was not found.
```

Solution to example 6:

The user enters the number 45.

First iteration:

The initial value of **start** is 0 and **end** is 6

middle becomes $(\text{start}+\text{end})/2=6/2=3$

When **arr[middle]**=40 < 45, the next statement will be

start=middle+1=3+1=4

Second iteration:

middle becomes $(\text{start}+\text{end})/2=(4+6)/2=5$

When **arr[middle]**=60 > 45, the next statement will be

end=middle-1 = 4

Third iteration:

middle becomes $(\text{start}+\text{end})/2 = (4+4)/2 = 4$

When **arr[middle]**=50 > 45, the next statement will be

end = middle-1 = 3

When **start > end**, the loop terminates and the function **returns -1**.

Example 7: (Selection sort algorithm)

Write a function that takes as parameters an array of doubles and uses the selection sort algorithm to sort it in ascending order. Write a C program that reads the grades of 10 students, stores them in an array, and uses the function to sort it.

Solution to example 7 uses selection sort algorithm.

In each iteration of the inner loop, **arr[i]** is compared with the elements from **i+1** up to **SIZE-1**.

If an element is less than **arr[i]**, their values are swapped.

Thus, in each iteration of the outer loop, the smallest value of the elements from **i** up to **SIZE-1** is stored at **arr[i]**.

To sort an array in descending order, change the **if** statement to:

```
if (arr[i] < arr[j])
```

```
#include <stdio.h>
#define SIZE 10
void sel_sort(double arr[]);
int main(void) {
    int i;
    double grd[SIZE];
    for(i = 0; i < SIZE; i++) {
        printf("Enter grade of stud_%d: ", i+1);
        scanf("%lf", &grd[i]); }
    sel_sort(grd);
    printf("\n***** Sorted array *****\n");
    for(i = 0; i < SIZE; i++)
        printf("%.2f\n", grd[i]);
    return 0; }
void sel_sort(double arr[]) {
    int i, j;
    double tmp;
    for(i = 0; i < SIZE-1; i++){
        for(j = i+1; j < SIZE; j++) {
            if(arr[i] > arr[j]) {
                //Swap values.
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp; } } }
```

Thank you

Lecture 16

Problem solving using linear and binary searching
of arrays

Objectives

- Searching arrays:
 - Linear search
 - Binary search
- Problem solving.

Linear search algorithm:

Linear search algorithm is a sequential search using an algorithm to search for a value in a *non-sorted* array. The searched value is compared with the value of each element until a match is found. In an array of n elements, the maximum number of searches is n . This may occur if the searched value is not found or is equal to the last element.

Linear search algorithm, cont.:

Idea:

Start with the first element in the array or list.

Compare it with the given key, if key and value at current index are same, return the current index.

Else increase the index value and repeat step 2 until end of list is reached.

//Given a set of data

```
int [] arr = {10, 2, 7, 9, 17, 4};
```

//and a particular value

```
int val = 7;
```

//Find the first index of the value in the data.

```
return index = 2
```

//(because arrays follow 0 index structure i.e. the index of first element is 0 and so on)

Linear search algorithm, cont.:

Pseudocode:

Input:

Array arr, integer key

Output:

first index of key in arr, or -1 if not found

for i = 0 to last index of arr:

 if arr[i] equals key:

 return i

return -1

Linear search algorithm, cont.:

```
#include <stdio.h>
//A linear search algorithm to find and return index of the searched element
int recursiveLinearSearch(int array[], int arraySize, int value, int index) {
    if(index<arraySize)    {
        if(array[index] == value)
            return index;
        else
            return recursiveLinearSearch(array, arraySize, value, index+1);
    }
    else {
        return -1; } }
int main(void) {
    int array[] = {13, 22, 17, 5, 15, 30, 98, 16, 43, 33,90};
//input array
    int value = 16; //value to be searched
    int arraySize = sizeof(array)/sizeof(array[0]);
    int result = recursiveLinearSearch(array, arraySize, value, 0);
//calling function to apply linear search
    (result == -1)? printf("Element is not present in array")
                  : printf("Element is present at index %d", result);
return 0; }
```

Example 1:

Write a function that takes as parameter an integer array and returns the maximum number of the same occurrences. For example, if the array is {1, 10, -3, 5, -3, 8}, the function should return 2 because -3 appears the most times, that is, 2. Write a program that reads 10 integers, stores them in an array, and uses the function to display the maximum number of the same occurrences.

Binary search algorithm:

The binary search algorithm is used to search for and to find a value in a *sorted* array (either in ascending or descending order).

The sorted array is used to reduce the time complexity, so that the use of **size of** the elements reduces to half after each iteration.

This is achieved by comparing the middle element with the key and if they are ***unequal*** then we choose the first or second half, whichever is expected to hold the key (if available) based on the comparison.

If array is sorted in an increasing manner and the key is smaller than middle element and if key exists, it will be in the first half,

So as is it is chosen and repeats same operation again and again until key is found or no more elements are left in the array.

Binary search algorithm, cont.:

```
//initially called with low = 0, high = N - 1
BinarySearch_Right(A[0..N-1], value, low, high) {
    //invariants: value >= A[i] for all i < low
    value < A[i] for all i > high
    if (high < low)
        return low
    mid = low +((high - low) / 2) // THIS IS AN IMPORTANT
    STEP TO AVOID BUGS
    if (A[mid] > value)
        return BinarySearch_Right(A, value, low, mid-1)
    else
        return BinarySearch_Right(A, value, mid+1, high) }
```

Problem 2:

Suppose that the user enters the number 45.

First iteration. The initial value

of start is 0 and end is 6. middle becomes $(\text{start}+\text{end})/2 = 6/2 = 3$. Since $\text{arr}[\text{middle}]$ is 40, less than 45, the next statement to be executed is $\text{start} = \text{middle}+1 = 3+1 = 4$.

Second iteration. middle becomes $(\text{start}+\text{end})/2 = (4+6)/2 = 5$.

Since $\text{arr}[\text{middle}]$ is 60, greater than 45, the next statement is $\text{end} = \text{middle}-1 = 4$.

Third iteration. middle becomes $(\text{start}+\text{end})/2 = (4+4)/2 = 4$.

Since $\text{arr}[\text{middle}]$ is 50, greater than 45, the next statement is $\text{end} = \text{middle}-1 = 3$. Since start is greater than end, the loop terminates and the function returns -1.

Solution:

```
#include <stdio.h>
int binary_search(int arr[], int size, int num);
int main(void) {
    int num, pos, arr[] = {11, 11, 33, 44, 55, 66, 77};
    printf("Enter number to search: ");
    scanf("%d", &num);
    pos = binary_search(arr, 7, num);
    if(pos == -1)
        printf("%d isn't found\n", num);
    else
        printf("%d is found in position %d\n", num, pos);
    return 0; }
int binary_search(int arr[], int size, int num) {
    int start, end, middle;
    start = 0;
    end = size-1;
    while(start <= end) {
        middle = (start+end)/2;
        if(num < arr[middle])
            end = middle-1;
        else if(num > arr[middle])
            start = middle+1;
        else
            return middle;}
    return -1; /* If execution reaches this point it that is the number
was not found.
```

Problem 3:

Write a C program to read an integer within [0, 100] and uses the binary search algorithm to “guess” that number. The program asks questions to define if the number that is searched for is less or greater than the middle of the examined interval. The answers must be given in the form of 0 (No) or 1 Yes). The C program is to display in the number of tries that were used to find the number.

Thank you

Lecture 17

Problem solving using searching arrays
and sorting of arrays

Objectives

- Searching arrays
- Sorting arrays
- Problem solving.

Example 1:

Write a function that searches for a number in an array of integers. If the number is found, the function should return its position, otherwise -1. Write a program that initializes an array of integers with values sorted in ascending order. The program should read an integer and use the function to display its position in the array.

```
#include <stdio.h>
int binary_search(int arr[], int size, int num);
int main(void) {
    int num, pos, arr[] = {11, 22, 33, 44, 55, 66, 77};
    printf("Enter number to search: ");
    scanf("%d", &num);
    pos = binary_search(arr, 7, num);
    if(pos == -1)
        printf("%d isn't found\n", num);
    else
        printf("%d is found in position %d\n", num, pos);
    return 0; }

int binary_search(int arr[], int size, int num) {
    int start, end, middle;
    start = 0;
    end = size-1;
    while(start <= end) {
        middle = (start+end)/2;
        if(num < arr[middle])
            end = middle-1;
        else if(num > arr[middle])
            start = middle+1;
        else
            return middle; }
```

Explanation to problem 1:

Suppose that the user enters the number 45.

First iteration.

The initial value of start is 0 and end is 6.

middle becomes $(\text{start}+\text{end})/2 = 6/2 = 3$.

Since $\text{arr}[\text{middle}]$ is 40, less than 45, the next statement to be executed is $\text{start} = \text{middle}+1 = 3+1 = 4$.

Second iteration.

middle becomes $(\text{start}+\text{end})/2 = (4+6)/2 = 5$.

Since $\text{arr}[\text{middle}]$ is 60, greater than 45, the next statement is $\text{end} = \text{middle}-1 = 4$.

Third iteration.

middle becomes $(\text{start}+\text{end})/2 = (4+4)/2 = 4$.

Since $\text{arr}[\text{middle}]$ is 50, greater than 45, the next statement is $\text{end} = \text{middle}-1 = 3$.

Since start is greater than end, the loop terminates and function returns -1.

Selection sort algorithm: (ascending order)

At first, we find the element with the smallest value and we swap it with the first element of the array, thus the smallest value is stored in the first position.

Then, we find the smallest value among the remaining elements, except the first one. Again, we swap that element with the second element of the array, thus the second smallest value is stored in the second position.

This procedure is repeated with the rest of the elements and the algorithm terminates once the last two elements are compared.

To sort the array in *descending order*, we find the largest value instead of the smallest.

Example 2:

Write the `add_sort()` function to insert a number into a sorted array, so that the array remains sorted. Create a C program to read 9 integers, stores them in an array of 10 integers, and sorts these 9 elements in ascending order. Then, the program is to read the last integer, use the `add_sort()` to insert it in the array, and display the sorted array.

```
#include <stdio.h>
#define SIZE 10
void sel_sort(int arr[], int size); // implements
selection sort algorithm to sort an array in ascending order.
void add_sort(int arr[], int size, int num);
int main(void) {
    int i, num, a[SIZE];
    for(i = 0; i < SIZE-1; i++) { //Read 9 int and store
them in the array.
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    sel_sort(a, SIZE-1); //Sort the 9 elements .
    printf("Insert number in sorted array: ");
    scanf("%d", &num);
add_sort(a, SIZE-1, num); //Insert the last integer in the array.
    for(i = 0; i < SIZE; i++)
        printf("%d\n", a[i]);
return 0; }
```

```
void add_sort(int arr[], int size, int num) {  
    int i, pos;  
    if(num <= arr[0])  
        pos = 0;  
    else if(num >= arr[size-1]) { //If it greater  
than the last one, store it in the last position  
and return.  
        arr[size] = num;  
        return; }  
    else  
{
```

```
for(i = 0; i < size-1; i++)
{
    //Check all adjacent pairs up to the last one at positions SIZE-3 and
    //SIZE-2 to find the position to insert the number .
    if(num >= arr[i] && num <= arr[i+1])
        break;
    }
    pos = i+1;
}
for(i = size; i > pos; i--)
    arr[i] = arr[i-1]; //Elements are shifted one
position to the right, starting from the last position of the array,
that is [SIZE-1],
```

```
//up to the position in which the new number will be inserted,  
//such that, in the last iteration:  
//i = pos+1, so, arr[pos+1] = arr[pos]//  
    arr[pos] = num; } //Store the number  
void sel_sort(int arr[], int size) {  
    int i, j, temp;  
    for(i = 0; i < size-1; i++) {  
        for(j = i+1; j < size; j++) {  
            if(arr[i] > arr[j]) {  
                //Swap the values.  
                temp = arr[i];  
                arr[i] = arr[j];  
                arr[j] = temp;  
            }  
        }  
    }  
}
```

Insertion sort algorithm: (in ascending order)

A sequential comparisons between each element (starting from the second and up to the last element) and the elements on its left is completed.

The elements on its right form the “unsorted sub-array.” The leftmost element of the unsorted sub-array (i.e., the examined element) is compared against the elements of the sorted sub-array from right to left such that:

1. The examined element is stored in a temporary variable.
- 2i. If it is not less than the rightmost element of the sorted sub-array, its position does not change, the testing of the next most left element continuous.
- 2ii. If it is less than the rightmost element of the sorted sub-array, the latter is shifted one position to the R while the examined element is compared against the R-most but one element. If it is not less, the examined element is stored at the position of the last shifted element; otherwise, the same procedure is repeated (the rightmost but one element is shifted one position to the R and the examined element is compared against the R-most but two elements) .
Step 2ii terminates either if the examined element is not less than an element of the sorted sub-array or the first element of the sorted sub-array is reached.
Then, Step 1 is repeated for the new L-most element.

Insertion sort algorithm: (in descending order)

To sort the array in descending order, an element of the sorted sub-array is shifted one position to the R if it is less than the examined element.

Example 3:

Create a C function to take as parameter an array of integers and use the insertion sort algorithm to sort it in ascending order. Write a C program to read five integers, to store them in an array, and to use the function to sort it.

```
#include <stdio.h>
#define SIZE 5
void insert_sort(int arr[]);
int main(void) {
    int i, a[SIZE];
    for(i = 0; i < SIZE; i++) {
        printf("Enter number: ");
        scanf("%d", &a[i]);
    }
    insert_sort(a);
    printf("\n-- The sorted array is: --\n");
    for(i = 0; i < SIZE; i++)
        printf(" %d\n", a[i]);
    return 0;
}
void insert_sort(int arr[]) {
    int i, j, temp;
    for(i = 1; i < SIZE; i++) {
        temp = arr[i];
        j = i;
        while((j > 0) && (arr[j-1] > temp)) {
            arr[j] = arr[j-1];      //Shift this element one position to R
            j--;
        }
        arr[j] = temp;
    }
}
```

The **for** loop compares the elements starting from the second one. In each iteration, temp holds the examined element.

The **while** loop shifts one position to the right the elements being on the left of the examined element and are greater than it.

Suppose that the array elements are 7, 3, 1, 9, 4.

First **for** loop iteration (i = 1 and temp = arr[1] = 3)

(j = 1) First **while** loop iteration: 7 --> 3 1 9 4

So, the array is transformed to: 3 7 1 9 4

Second **for** loop iteration (i = 2 and temp = arr[2] = 1)

(j = 2) First **while** loop iteration: 3 7 --> 1 9 4

(j = 1) Second **while** loop iteration: 3 --> 7 1 9 4

So, the array is transformed to: 1 3 7 9 4

Third **for** loop iteration (i = 3 and temp = arr[3] = 9)

(j = 3) No shifting takes place because the fourth array element (i.e., arr[3] = 9) is greater than the rightmost element (i.e., arr[2] = 7).

So, the array remains the same: 1 3 7 9 4

Cont: So, the array remains the same: 1 3 7 9 4

Fourth for loop iteration ($i = 4$ and $\text{temp} = \text{arr}[4] = 4$)

($j = 4$) First **while** loop iteration: 1 3 7 9 \rightarrow 4

($j = 3$) Second **while** loop iteration: 1 3 7 \rightarrow 9 4

The sorting is completed and the array is transformed

to: 1 3 4 7 9

To sort the array in descending order, change the **while** statement to

while (($j > 0$) && ($\text{arr}[j-1] < \text{temp}$))

in order to shift one position to the right, the elements of the sorted subarray, which are less than the examined element.

Problems to solve for practicing:

- 1. Create a C program that reads 5 integers and stores them in an array. The C programsis to use qsort() to sort the array in ascending order. Then, the program reads an integer and use bsearch() to check if it is stored in the array.**

- 2. Create a C program to read the populations of 50 cities and stores them in ascending order in an array when entered. The program should display the array before it ends. For example, if the users enters 2000 and then 1000, the first two elements should be 1000, 2000. If the next input value is 1500, the first three elements should be1000, 1500, and 2000.**

Thank you

Lecture 18

Structures, unions, and problem solving

Objectives

- Structures
- Unions
- Problem solving.

Structure:

A structure is a collection of related data items of different types.

A structure type in C is called **struct**.

A **struct** can be composed of data of different types.

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly, structure is another user-defined data type that allows to combine data items of different kinds.

Structures hold data that belong together. In database applications, structures are called records.

Examples:

Student record: student id, name, major, gender, start year, ...

Bank account: account number, name, currency, balance, ...

Address book: name, address, telephone number, ...

Structure definition :

To define a structure, a **struct** statement is used.

The **struct** statement defines a new data type, with more than one **member**. Individual members are named using field identifiers.

The format of the **struct** statement is as follows:

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

Examples:

```
struct date {  
    int day;  
    int month;  
    int year;  
}; //The "date" structure has 3 members, day, month and year.
```

```
struct studentInfo {  
    int id;  
    int age;  
    char gender;  
    double gpa;  
}; //The "StudentInfo" structure has 4 members of different types.
```

```
struct studentGrade{  
    char name[15];  
    char course[9];  
    int lab[5];  
    int homework[3];  
    int exam[2];  
}; // "StudentGrade" structure has 5 members of different array types.
```

Examples:

```
struct bankAccount{  
    char name[15];  
    int account_No[10];  
    double balance;  
    double date_birthday;
```

}//“BankAccount” structure has simple, array and structure types as members.

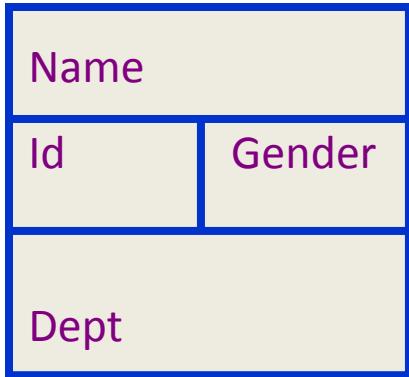
```
struct studentRecord{  
    char name[15];  
    int id;  
    char dept[5];  
    char gender;
```

//The “StudentRecord” structure has 4 members.

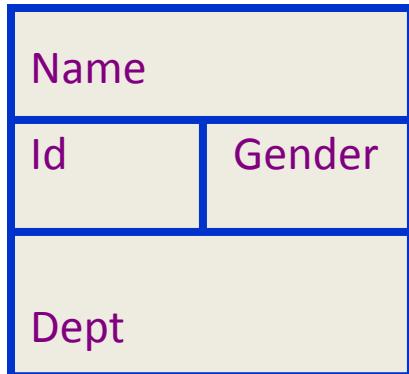
Declaration of a struct type:

```
struct data-type identifier_list;  
studentRecord student1, student2;
```

student1



student2



student1 and **student2** are part of **studentRecord** type

Basics of struct:

The structure tag is optional and each member definition is a normal definition, such as `int i;` or `float f;` or any other valid variable definition.

At the end of the structure's definition, before the final semicolon, one can specify one or more structure variables but it is optional. Book structure:

```
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```

Accessing structure members:

To access any member of a structure, the member access is done through a dot operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. **struct** is used to define variables of structure type.

The use of a structure in a C program:

```
#include <stdio.h>
#include <string.h>
struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

int main( )
{
    struct Books Book1; //Declare Book1 of type Book
    struct Books Book2; //Declare Book2 of type Book
    strcpy( Book1.title, "C Programming"); //book 1 specification
    strcpy( Book1.author, "Denis Kernighan");
    strcpy( Book1.subject, "Programming");

    Book1.book_id = 6495407;

    strcpy( Book2.title, "Linear Algebra"); //book 2 specification
    strcpy( Book2.author, "Adam Smith");
    strcpy( Book2.subject, "Mathematics");

    Book2.book_id = 6495700;
```

```
Book2.book_id = 6495700;
printf( "Book 1 title : %s\n", Book1.title);
    //print Book1 info
printf( "Book 1 author : %s\n", Book1.author);
printf( "Book 1 subject : %s\n", Book1.subject);
printf( "Book 1 book_id : %d\n", Book1.book_id);

    //print Book2 info
printf( "Book 2 title : %s\n", Book2.title);
printf( "Book 2 author : %s\n", Book2.author);
printf( "Book 2 subject : %s\n", Book2.subject);
printf( "Book 2 book_id : %d\n", Book2.book_id);
return 0;
}
```

Structure as a function argument:

A structure can be passed as a function argument in the same way as one passes any other variable or pointer.

```
#include <stdio.h>
#include <string.h>
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};
void printBook( struct Books book ); //function declaration
```

```
int main( ) {
    struct Books Book1; //Declare Book1 of type Book
    struct Books Book2; //Declare Book2 of type Book
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "D. Kernighan");
    strcpy( Book1.subject, "Programming");
    Book1.book_id = 6495407;
    strcpy( Book2.title, "Linear Algebra"); //book 2
specifications
    strcpy( Book2.author, "John Petroff");
    strcpy( Book2.subject, "Mathematics");
    Book2.book_id = 6495700;
    printBook( Book1 );
    printBook( Book2 ); //Print Book2 info
    return 0; }
void printBook( struct Books book ) {
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id); }
```

Pointers to structure:

- Pointers to structures work as one defines pointer to a variable:

```
struct Books *struct_pointer;
```

- One can store the address of a structure in the above-defined pointer ***struct_pointer**
- To find the address of a structure, place the ‘&’ operator before the structure as follows:

```
struct_pointer = &Book1;
```

- To access the members of a structure using a pointer to that structure, one must use the **-> operator** as follows:

```
struct_pointer->title;
```

Pointers to structure:

```
#include<stdio.h>
#include<string.h>
struct Books
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

//function declaration
void printBook( struct Books *book );
int main( ) {
    struct Books Book1; //Declare Book1 of type Book
    struct Books Book2; //Declare Book2 of type Book
    //book 1 specification
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Kernighan");
    strcpy( Book1.subject, "Programming");
    Book1.book_id = 6495407;
```

cont.

Pointers to structure, cont.:

```
//book 2 specification
strcpy( Book2.title, "Differential Equations");
strcpy( Book2.author, "Zill");
strcpy( Book2.subject, "Applied Science");
Book2.book_id = 6495700;
//print Book1 info by passing address of Book1
printBook( &Book1 );
//print Book2 info by passing address of Book2
printBook( &Book2 );
return 0; }

void printBook( struct Books *book ) {
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id); }
```

Bit fields:

When the C program contains a number of TRUE/FALSE variables grouped in a structure called *status*, such as:

```
struct
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

Bit fields, cont.:

The ***status*** structure requires 8 bytes of memory space where either 0 or 1 will be stored in each of the variables. In C, the width of the variable can be defined so that the C compiler can use only those number of bytes.

```
struct
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for ***status*** variable, it will only use 2 bits to store the values.

If 32 variable are used, each one with a width of 1 bit, then the ***status*** structure uses 4 bytes. As soon as there will be the 33 variable, the compiler will allocate the next slot of the memory and it will start using 8 bytes such as:

```
#include <stdio.h>
#include <string.h>
struct //define simple structure
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;
struct //define a structure with bit fields
{
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;
int main( )
{
    printf("Memory size used by status1:%d\n", sizeof(status1));
    printf("Memory size used by status2:%d\n", sizeof(status2));
    return 0;
}
```

Bit fields, cont.:

The variables defined with a predefined width are called ***bit fields***. A ***bit field*** can hold more than a single bit; for example, if one needs a variable to store a value from 0 to 7, then one can define a bit-field with a width of 3 bits such as:

```
struct { unsigned int age : 3; } Age;
```

The keyword **typedef** can be used to give the type, a new name.

For example:

```
typedef unsigned char BYTE;
```

```
BYTE b1, b2; //BYTE: Symbolic abbreviation
```

For example:

```
typedef unsigned char byte;
```

typedef can be used with structure to define a new data type and then use that data type to define structure variables directly such as:

```
#include <stdio.h>
#include <string.h>
//typedef can be used with structure to define a new data type and then use
that data type to define structure variables directly
typedef struct Books {
    char title[30];
    char author[30];
    char subject[50];
    int book_id;
} Book;
int main( ) {
    Book book;
    strcpy( book.title, "Programming in C");
    strcpy( book.author, "Kernighan");
    strcpy( book.subject, "Programming");
    book.book_id = 6495407;
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
    return 0; }
```

typedef v.s #define:

#define is a C-directive,

typedef is limited to giving symbolic names to types only,

#define is used to define alias for values such as define 1 as ONE,

typedef interpretation is performed by a compiler,

#define statements are processed by a preprocessor.

Example:

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main( )
{
    printf( "Value of TRUE : %d\n", TRUE);
    printf( "Value of FALSE : %d\n", FALSE);
    return 0;
}
```

Union:

The union is used to store different data types in the same memory location.

The union can have many members, but only one member can contain a value at any given time.

Unions are efficient in using the same memory for multiple purpose.

Define an union:

Use an union statement to define a new data type with more than one member for the C program.

The union tag is optional and each member definition is a normal variable definition, such as int i; or float f; At the end of the union, before the final semicolon, it is optional that one can specify one or more union variables.

One can use any built-in or user-defined data types inside a union based on the requirement.

The memory occupied by a union is large enough to hold the largest member of the union.

Example 1:

The data type occupies 20 bytes of memory space that is the maximum space which can be used by a character string. Examine the C program that displays the total memory size used by the union:

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    printf("Memory size used by data is %d.\n", sizeof(data));
    return 0;
}
```

Example 2:

A member access operator (.) between the *union* variable and the union member is used to access it. The keyword *union* is used to define variables of *union* type.

```
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0; }
```

Example 3:

Use 1 variable at a time in the union to satisfy the purpose of the union.

```
#include <string.h>
union Data
{
    int i;
    float f;
    char str[34];
};
int main( )
{
    union Data data;
    data.i = 55;
    printf( "data.i : %d\n", data.i);
    data.f = 340.00;
    printf( "data.f : %.3f\n", data.f);
    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);
    return 0; }
```

Pointer to a structure member:

A pointer to a structure member is used like an ordinary pointer. The C program below uses pointers to display the values of the b members:

```
#include <stdio.h>
#include <string.h>
struct book {
    char title[100];
    int year;
    float price;};
int main(void) {
    char *ptr1;
    int *ptr2;
    float *ptr3;
    struct book b;
    strcpy(b.title, "C Programming Language ");
    b.year = 2016;
    b.price = 14.75;
    ptr1 = b.title;
    ptr2 = &b.year;
    ptr3 = &b.price;
    printf("%s %d %.2f\n", ptr1, *ptr2, *ptr3);
    return 0; }
```

Structure operations

```
#include <stdio.h>
struct student {
    int code;
    float grd; };
int main(void) {
    struct student s1, s2;
    s1.code = 1234;
    s1.grd = 6.7;
    s2 = s1; //Copy the structure
    printf("C:%d G:%.2f\n", s2.code, s2.grd);
    return 0; }
```

Structure operations, cont.,

To test whether two structures are equal, it is to compare their members one by one such as:

```
if((s1.code == s2.code) && (s1.grd == s2.grd))
```

The operators `==` and `!=` cannot be used to test whether two structures are equal or not.

Not allowed to write:

```
if(s1 == s2) or if(s1 != s2)
```

Structure containing pointers

A structure may contain one or more pointer members.

```
#include <stdio.h>
struct student {
    char *name;
    float *avg_grd; };
int main(void) {
    float grd = 8.5;
    struct student s1, s2;
    s1.name = "somebody";
    s1.avg_grd = &grd;
    printf("%s %.2f\n", s1.name+3, *s1.avg_grd);
    s2 = s1;
    grd = 3.4;
    printf("%s %.2f\n", s2.name, *s2.avg_grd);
    return 0; }
```

Structure as function argument

A structure variable can be passed to a function like any other variable, either the variable itself or its address.

```
#include <stdio.h>
#include <string.h>
struct student {
    char name[50];
    int code;
    float grd; };
```

cont.,

Structure as function argument

```
int main(void) {  
    struct student stud = {"somebody", 20, 5};  
    test(stud); //According to the function  
//declaration, it must be called with an argument of type  
student.  
    printf("N:%s C:%d G:%.2f\n",  
stud.name, stud.code, stud.grd);  
    return 0; }  
void test(struct student stud_1) {  
    strcpy(stud_1.name, "new_name");  
    stud_1.code = 30;  
    stud_1.grd = 7; }
```

Structure as function argument:

When the address is passed, the function may change the values of the members.

For example:

test() is changed in order to modify the members of **stud**

Structure as function argument:

```
#include <stdio.h>
#include <string.h>
struct student {
    char name[50];
    int code;
    float grd; }
void test(struct student *ptr);
int main(void) {
    struct student stud = {"somebody", 20, 5};
    test(&stud);
    printf("N:%s C:%d G:%.2f\n", stud.name, stud.code, stud.grd);
    return 0; }
void test(struct student *ptr) {
    strcpy(ptr->name, "new_name");
    ptr->code = 33;
    ptr->grd = 77; }
```

Thank you

The review for the final examination includes the following topics per chapters:

Chapter 1: Variables, Arithmetic Expressions; “for” statement; Symbolic constants; Character Input and Output; File Copying; Character, line, and word counting; Introduction to Arrays, Functions, and Pointers; Arguments - Call by value; External Variables and Scope.

Chapter 2: Types, Operators and Expressions: Names; Data Types and Sizes; Constants; Declarations; Arithmetic Operators; Relational and Logical Operators; Conversions; Increment Decrement Operators; Bitwise Operator; Assignment Operators and Expression; Conditional Expressions; Precedence and Order of Evaluation

Chapter 3: Control Flow; Statements and Blocks; If-Else; Else-If; Loops - While and For Loops - Do-While; Break and Continue; Goto and labels.

Chapter 4: Functions and Program Structure; Basics of Functions; Functions Returning Non-integers; External Variables; Rules; Header Files; Variables; Block Structure; Use of Macros and Conditional Inclusion.

Chapter 5: Pointers and Arrays; Pointers and Functions; Pointers to Pointers; Multi-dimensional Arrays; Pointers to Functions.

Chapter 6: Structures; Structures and Functions; Arrays of Structures; Pointers to Structures; Typedef; Unions; Bit-fields.

Chapter 7: Input and Output; Formatted Output – printf; Argument Lists; and File Access.

Problem solving

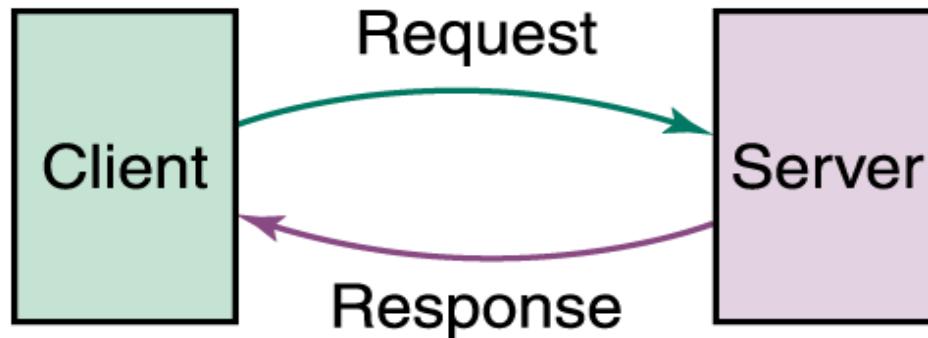
Networking

Computer network A collection of computing devices that are physically connected to communicate and share resources, using physical wires or cables. There are also wireless connections, using radio waves or infrared signals.

The generic term **node** or **host** refers to any device on a network
Data transfer rate (key issue in computer network) is the speed with which data is moved from one place on a network to another.

Networking

Computer networks consists of a **client/server model**.



File server is a computer that stores and manages files for multiple users on the network.

Web server is a computer which responds to requests (from the browser client) for web pages.

Problem solving

Problem 1:

Write a C program that reads an Internet Protocol (IP) version 4 address (IPv4) and checks if it is valid. The form of a valid IPv4 address is x.x.x.x, where each x must be an integer within [0, 255].

Solution:

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

Solution:

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

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
    {
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        }
    }
}
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)    {
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)  {
printf("Error: The value of each byte should be in [0,
255]\n");
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)    {
printf("Error: The value of each byte should be in [0,
255]\n");
            return 0;      }
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)    {
printf("Error: The value of each byte should be in [0,
255]\n");
            return 0;    }
bytes++;temp = -1;}//The value-1 means current address byte is checked.
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)  {
printf("Error: The value of each byte should be in [0,
255]\n");
                return 0;    }
bytes++;temp = -1;}//The value-1 means current address byte is checked.
}
        else
{

```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)    {
printf("Error: The value of each byte should be in [0,
255]\n");
            return 0;    }
bytes++;temp = -1;}//The value-1 means current address byte is checked.
    }
    else
{
printf("Error: Acceptable chars are only digits and dots\n");
}
```

Solution:

```
#include <stdio.h>
int main(void) {
int ch, dots, bytes, temp;
dots = bytes = temp = 0;
printf("Enter IP address (x.x.x.x): ");
while((ch = getchar()) != '\n' && ch != EOF)    {
    if(ch < '0' || ch > '9')    {
        if(ch == '.')
        {
            dots++;
        if(temp != -1)  {
            if(temp > 255)    {
printf("Error: The value of each byte should be in [0,
255]\n");
            return 0;    }
bytes++;temp = -1;}//The value-1 means current address byte is checked.
    }
    else
{
printf("Error: Acceptable chars are only digits and dots\n");
return 0; }
```

Solution, cont.:

```
 }  
 else  
{
```

Solution, cont.:

```
 }  
 else  
 {  
     if(temp == -1)
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');
} }
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');
}  }
if(temp != -1) //Check the value of the last address byte.
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');
} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
{
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');
} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
    {
printf("Error: The value of each byte should be in [0, 255]\n");
        return 0; }
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');

} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
    {
printf("Error: The value of each byte should be in [0, 255]\n");
        return 0; }
    bytes++;
}
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');

} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
    {
printf("Error: The value of each byte should be in [0, 255]\n");
        return 0; }

bytes++;
    }

    if(dots != 3 || bytes != 4)
printf("Error: The IP format should be x.x.x.x\n");
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');

} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
    {
printf("Error: The value of each byte should be in [0, 255]\n");
        return 0; }

bytes++;
    }

    if(dots != 3 || bytes != 4)
printf("Error: The IP format should be x.x.x.x\n");
    else
printf("The input address is a valid IPv4 address\n");
```

Solution, cont.:

```
}

else
{
    if(temp == -1)
temp = 0;      //Make it 0, to start checking the next address byte.
    temp = 10*temp + (ch-'0');

} }

if(temp != -1) //Check the value of the last address byte.
{
    if(temp > 255)
    {
printf("Error: The value of each byte should be in [0, 255]\n");
        return 0; }

bytes++;
    }

    if(dots != 3 || bytes != 4)
printf("Error: The IP format should be x.x.x.x\n");
    else
printf("The input address is a valid IPv4 address\n");
    return 0; }
```

Problem 2:

Write a C program to determine the class, network and the host ID of an IPv4 address.

Solution:

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

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
    while (i >= 0) {
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
    while (i >= 0) {
        ip = ip + (str[i] - '0') * j;
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
    while (i >= 0) {
        ip = ip + (str[i] - '0') * j;
        j = j * 10;
        i--; }
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
    while (i >= 0) {
        ip = ip + (str[i] - '0') * j;
        j = j * 10;
        i--; }
    // Write for Class A
    if (ip >=1 && ip <= 126)
        return 'A';
```

```
#include<stdio.h>
#include<string.h>
//Create a function to find out the class
char findClass(char str[]) {
    // Store the first octet in an arr[]
    char arr[4];
    int k;
    int i = 0;
    while (str[i] != '.') {
        arr[i] = str[i];
        i++; }
    i--;
    // Convert the array str[] into number for comparison
    int ip = 0, j = 1;
    while (i >= 0) {
        ip = ip + (str[i] - '0') * j;
        j = j * 10;
        i--; }
    // Write for Class A
    if (ip >=1 && ip <= 126)
        return 'A';
    // Write for Class B
    else if (ip >= 128 && ip <= 191)
        return 'B';
```

//Write for Class C

//Write for Class C

```
else if (ip >= 192 && ip <= 223)
```

//Write for Class C

```
else if (ip >= 192 && ip <= 223)  
return 'C';
```

```
//Write for Class C  
else if (ip >= 192 && ip <= 223)  
return 'C';  
//Write for Class D
```

```
//Write for Class C  
else if (ip >= 192 && ip <= 223)  
return 'C';  
//Write for Class D  
else if (ip >= 224 && ip <= 239)
```

```
//Write for Class C  
else if (ip >= 192 && ip <= 223)  
return 'C';  
  
//Write for Class D  
else if (ip >= 224 && ip <= 239)  
return 'D';
```

```
//Write for Class C  
else if (ip >= 192 && ip <= 223)  
return 'C';  
  
//Write for Class D  
else if (ip >= 224 && ip <= 239)  
return 'D';  
  
//Write for Class E
```

```
//Write for Class C
else if (ip >= 192 && ip <= 223)
return 'C';
//Write for Class D
else if (ip >= 224 && ip <= 239)
return 'D';
//Write for Class E
else
```

```
//Write for Class C
else if (ip >= 192 && ip <= 223)
return 'C';
//Write for Class D
else if (ip >= 224 && ip <= 239)
return 'D';
//Write for Class E
else
return 'E'; }
```

```
//Write for Class C
else if (ip >= 192 && ip <= 223)
return 'C';
//Write for Class D
else if (ip >= 224 && ip <= 239)
return 'D';
//Write for Class E
else
return 'E'; }
// Create a function to separate network ID and host ID and to
print them
```

```
//Write for Class C
else if (ip >= 192 && ip <= 223)
return 'C';
//Write for Class D
else if (ip >= 224 && ip <= 239)
return 'D';
//Write for Class E
else
return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
```

```
//Write for Class C
    else if (ip >= 192 && ip <= 223)
        return 'C';
//Write for Class D
    else if (ip >= 224 && ip <= 239)
        return 'D';
//Write for Class E
    else
        return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
    // Initialize network and host array to NULL
```

```
//Write for Class C
    else if (ip >= 192 && ip <= 223)
        return 'C';
//Write for Class D
    else if (ip >= 224 && ip <= 239)
        return 'D';
//Write for Class E
    else
        return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
    // Initialize network and host array to NULL
    char network[12], host[12];
```

```
//Write for Class C
    else if (ip >= 192 && ip <= 223)
        return 'C';
//Write for Class D
    else if (ip >= 224 && ip <= 239)
        return 'D';
//Write for Class E
    else
        return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
    // Initialize network and host array to NULL
    char network[12], host[12];
    int k;
```

```
//Write for Class C
    else if (ip >= 192 && ip <= 223)
        return 'C';
//Write for Class D
    else if (ip >= 224 && ip <= 239)
        return 'D';
//Write for Class E
    else
        return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
    // Initialize network and host array to NULL
    char network[12], host[12];
    int k;
    for (k = 0; k < 12; k++)
```

```
//Write for Class C
    else if (ip >= 192 && ip <= 223)
        return 'C';
//Write for Class D
    else if (ip >= 224 && ip <= 239)
        return 'D';
//Write for Class E
    else
        return 'E'; }
// Create a function to separate network ID and host ID and to
print them
void separate(char str[], char ipClass) {
    // Initialize network and host array to NULL
    char network[12], host[12];
    int k;
    for (k = 0; k < 12; k++)
        network[k] = host[k] = '\0';
```

// For class A, only first octet is Network ID and rest are host ID

```
// For class A, only first octet is Network ID and rest are host ID  
if (ipClass == 'A') {
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {
    int i = 0, j = 0;
    while (str[j] != '.')
        network[i++] = str[j++];
    i = 0;
    j++;
    while (str[j] != '\0')
        host[i++] = str[j++];
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

// For class B, first two octets are network ID and rest are host ID

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

// For class B, first two octets are network ID and rest are host ID

```
else if (ipClass == 'B') {
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

// For class B, first two octets are network ID and rest are host ID

```
else if (ipClass == 'B') {  
    int i = 0, j = 0, dotCount = 0;
```

// For class A, only first octet is Network ID and rest are host ID

```
if (ipClass == 'A') {  
    int i = 0, j = 0;  
    while (str[j] != '.')  
        network[i++] = str[j++];  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

// For class B, first two octets are network ID and rest are host ID

```
else if (ipClass == 'B') {  
    int i = 0, j = 0, dotCount = 0;
```

//Storing in network[] up to 2nd dot, dotCount keeps track of
number of dots or octets that are passed.

```
while (dotCount < 2) {
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }  
// for class C, first three octet are Network ID and rest are Host ID
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }  
// for class C, first three octet are Network ID and rest are Host ID  
else if (ipClass == 'C') {
```

```
while (dotCount < 2) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
    printf("Network ID is %s\n", network);  
    printf("Host ID is %s\n", host); }  
// for class C, first three octet are Network ID and rest are Host ID  
else if (ipClass == 'C') {  
    int i = 0, j = 0, dotCount = 0;
```

// storing in network[] up to 3rd dot, dotCount keeps track of
number of dots or octets passed.

// storing in network[] up to 3rd dot, dotCount keeps track of
number of dots or octets passed.

```
while (dotCount < 3) {
```

// storing in network[] up to 3rd dot, dotCount keeps track of number of dots or octets passed.

```
while (dotCount < 3) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }
```

// storing in network[] up to 3rd dot, dotCount keeps track of number of dots or octets passed.

```
while (dotCount < 3) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++;}  
    i = 0;  
    j++;
```

// storing in network[] up to 3rd dot, dotCount keeps track of number of dots or octets passed.

```
while (dotCount < 3) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++;}  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];
```

// storing in network[] up to 3rd dot, dotCount keeps track of number of dots or octets passed.

```
while (dotCount < 3) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++;}  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
printf("Network ID is %s\n", network);
```

// storing in network[] up to 3rd dot, dotCount keeps track of number of dots or octets passed.

```
while (dotCount < 3) {  
    network[i++] = str[j++];  
    if (str[j] == '.')  
        dotCount++; }  
    i = 0;  
    j++;  
    while (str[j] != '\0')  
        host[i++] = str[j++];  
printf("Network ID is %s\n", network);  
printf("Host ID is %s\n", host); }
```

// Class D and E are not divided in network and host ID

// Class D and E are not divided in network and host ID
else

```
// Class D and E are not divided in network and host ID  
else  
printf("In this Class, IP address is not"  
" divided into Network and Host ID\n"); }
```

```
// Class D and E are not divided in network and host ID
else
printf("In this Class, IP address is not"
" divided into Network and Host ID\n"); }
// Create a driver function is to test above function
```

```
// Class D and E are not divided in network and host ID
else
printf("In this Class, IP address is not"
" divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
```

```
// Class D and E are not divided in network and host ID
    else
        printf("In this Class, IP address is not"
        " divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
    char str[] = "192.226.12.11";
```

```
// Class D and E are not divided in network and host ID
    else
        printf("In this Class, IP address is not"
        " divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
    char str[] = "192.226.12.11";
    char ipClass = findClass(str);
```

```
// Class D and E are not divided in network and host ID
    else
        printf("In this Class, IP address is not"
        " divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
    char str[] = "192.226.12.11";
    char ipClass = findClass(str);
printf("The given IP address is from Class %c\n", ipClass);
```

```
// Class D and E are not divided in network and host ID
    else
        printf("In this Class, IP address is not"
        " divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
    char str[] = "192.226.12.11";
    char ipClass = findClass(str);
printf("The given IP address is from Class %c\n", ipClass);
separate(str, ipClass);
```

```
// Class D and E are not divided in network and host ID
    else
        printf("In this Class, IP address is not"
        " divided into Network and Host ID\n"); }
// Driver function is to test above function
int main() {
    char str[] = "192.226.12.11";
    char ipClass = findClass(str);
printf("The given IP address is from Class %c\n", ipClass);
    separate(str, ipClass);
    return 0; }
```

Thank you

Problems to solve

1. Create a C program to read a natural number and to convert the natural number to multiples of 50, 20, 10, and 1.
2. Write a function that takes an integer parameter and returns the result of $1^4 + 2^4 + 3^4 + \dots + n^4$. Write a C program that reads a positive integer up to 1000 and uses the function to display the result of the expression.
3. Write a function that searches for a number in an array of integers. If the number is found, the function should return its position, otherwise -1. Write a program that initializes an array of integers with values that are already sorted in ascending order. The program should read an integer and use the function to display its position in the array.
4. Write a C program to read an IPv4 address and to check its validity.
5. Use a recursive function to calculate the sum and the product of two numbers.
6. Build a calculator that can add, multiply, divide, and find the square root of two numbers.
7. Write a function that takes as parameters two strings and returns a pointer to the longer string. If the strings have the same number of characters, it should return NULL. Write a C program that reads two strings of less than 20 characters and uses the function to display the longer one.
8. Write a C program to evaluate the mathematical function: $f(x) = 0.5 x^3 - x \ln x$
9. Write a function that takes as parameters an integer and a character and displays the character as many times as the value of the integer. Write a C program that reads an integer and a character and uses the function to display the character.
10. Write a C program that reads a two-digit positive integer and duplicates its digits. For example, if the user enters 12, the program should display 1122.
11. Enter the current PIN code of a mobile phone to change the PIN code of it and the device checks if that code is equal to the one stored in the SIM card. If they are the same, the user is asked to enter the new PIN code once more for verification, and if it is entered correctly, the PIN is stored in the SIM card. Write a C program that simulates this process. Assume that the current code stored in SIM is 5678.
12. In a C Programming exam, each exam is graded by two GTAs. If the difference of their grades is less than z, the final grade is their average. Otherwise, the test is reviewed by a third GTA such as: i) If the grade of the third GTA is equal to the average of the first two grades, that is the final grade. ii) If the grade is less than the minimum of the first two grades, the final grade is minimal. iii) Otherwise, the final grade is the average of the grade of the third reviewer and the one of the first two grades closest to it. Write a C program to read the two grades and the difference z and to display the final grade according to that method.
13. Create a C program to read an integer and to display a message to indicate whether it is positive or negative. If it is 0, the program is to display zero. **Use the conditional operator.**

14. Create a C program to display the area of a square or a circle based on the user's choice. If the user enters 0, the program is to read the side of the square and to display its area. If the user enters 1, the program is to read the radius of the circle and to display its area.
15. A machine shop produces small and big metal discs. A small one costs \$0.007 and a big one \$0.03. For orders of more than \$310 or more than 3200 discs in total, a discount of 5% is offered. For orders of more than \$700, the discount is 15%. Create a C program to read the number of small and big metal discs ordered and to display the total cost.
16. Create a C program to read the initial population of a country and its annual population growth (as a percentage). Then, the C program reads the number of years and is to display the new population for each year.
17. Create a C program to read an integer and to display the number of its digits and their sum. For example, if the user enters 2345, the program should display 5 and 14 ($2+3+4+5=14$).
18. Create a C program to read an integer in $[0, 255]$ continuously and displays it in a binary form. For any value out of $[0, 255]$ the program is to terminate.
19. Create a C program to read an integer that corresponds to a number of lines. Assume the user enters 4 so that the C program adds spaces and '*' in accordance to the line number, as shown below (three spaces at the left of one '*' in the first line, two spaces at the left of two '*' in the second line, and so on). When the last line is reached, the program displays in each line one '*' less, and the spaces at the right of '*', until one '*' is displayed.

```
*  
* *  
* * *  
* * * *  
* * *  
* *  
*
```

20. Write a C program to read 100 integers and to display the two different higher values.
21. Write a C program to read the grades of 5 students and to store them in an array. Then, the program displays the average, the greatest, and the worst grade, as well as the positions in the array of their first occurrences. The program is an user defined program and requires the user to enter grades within $[50, 100]$.
22. Write a C program to read an integer and to display the digits that appear more than once and the number of their appearances. If no digit appears more than once, the program is to display a message about it.
23. Create a C program to read integers and to store them in a square matrix. Then, the program checks whether the array is a square; that is, the sum of each row, column, and diagonal is the same.

24. Write a C program to use three pointers to read the grades of a student in three exams. If all grades are greater than or equal to 6, the program displays them in ascending order. Otherwise, the program displays their average.

25. Write a C program to read the grades of 5 students, to store them in an array, and to display the best and the worst grade and the positions of their first occurrences in the array. The program is an user defined program, the user needs to enter grades within [50, 100]. Use pointer arithmetic to process the array.

To be continued