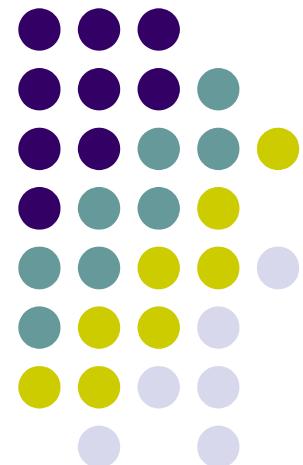


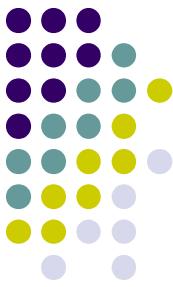
# Introduction to C Programming

---

## Chapter 2

### Constant ,Variables and Data Types





# Character Set

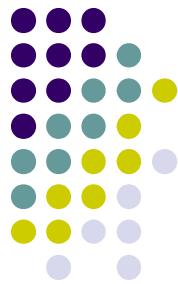
- The characters that can be used to form words, numbers and expressions depend upon the computer on which the program is run.
- The characters in C are grouped into the following categories:
  - Letters
  - Digits
  - Special characters
  - White spaces
- The entire character set is given in Table 2.1.



# Table 2.1

**Table 2.1 C Character Set**

<i>Letters</i>	<i>Digits</i>
Uppercase A.....Z	All decimal digits 0 .....9
Lowercase a.....z	
<b>Special Characters</b>	
,	& ampersand
.	^ caret
;	* asterisk
:	- minus sign
?	+ plus sign
'	< opening angle bracket
"	( or less than sign)
!	> closing angle bracket
	( or greater than sign)
/	( left parenthesis
\	) right parenthesis
~	[ left bracket
_	] right bracket
\$	{ left brace
%	} right brace
	# number sign
<b>White Spaces</b>	
Blank space	
Horizontal tab	
Carriage return	
New line	
<b>Form feed</b>	
Revised programming in ANSI C by E.Balagurusamy	



# Trigraph Characters

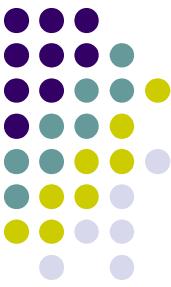
- Many non-English keyboards do not support all the characters mentioned in Table 2.1. ANSI C introduces the concept of "trigraph" sequences to provide a way to enter certain characters that are not available on some keyboards.
- **Each trigraph sequence consists of three characters (two question marks followed by another character) as shown in Table 2.2**



## Table 2.2

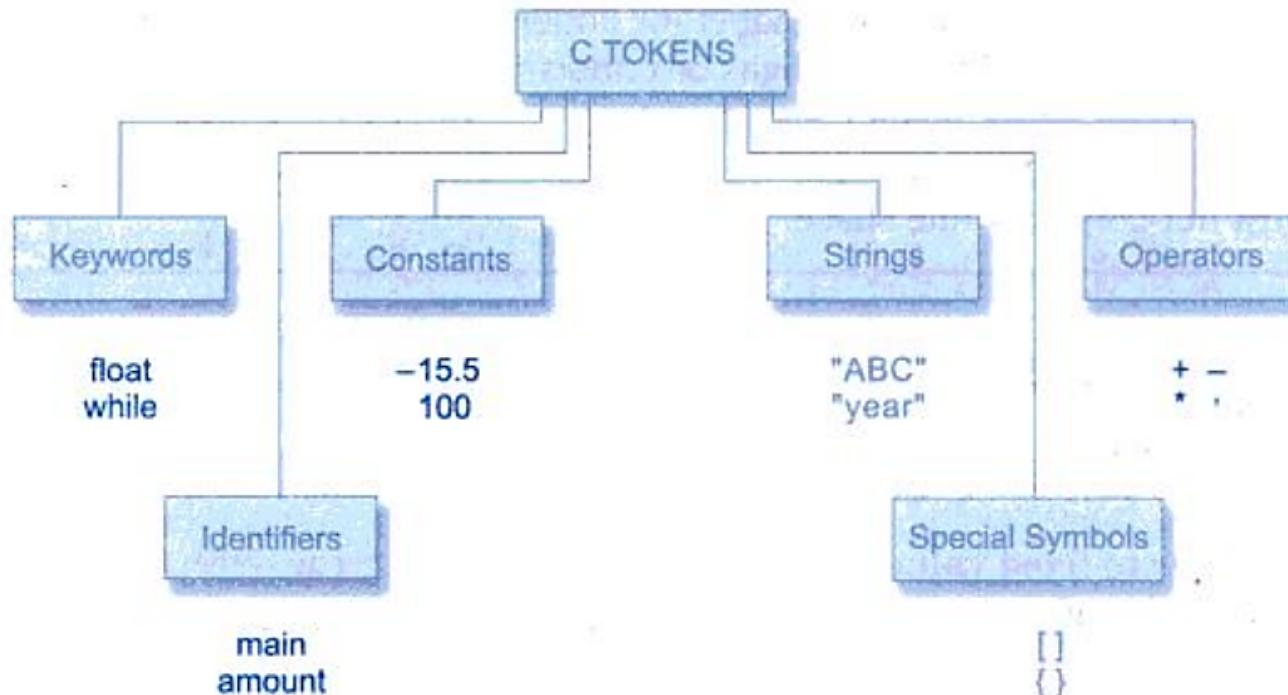
**Table 2.2 ANSI C Trigraph Sequences**

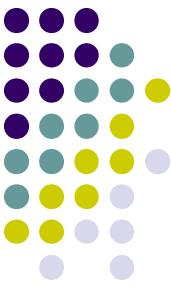
<i>Trigraph sequence</i>	<i>Translation</i>
??=	# number sign
??(	[ left bracket
??)	] right bracket
??<	{ left brace
??>	} right brace
??!	vertical bar
??/	\ back slash
??^	^ caret
??~	~ tilde



# C Tokens

- In a passage of text, individual words and punctuation marks are called tokens. Similarly, **in a C program the smallest individual units are known as C tokens.**



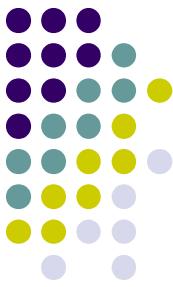


# Keywords and Identifier

- Every C word is classified as either a keyword or an identifier. All keywords have fixed meanings and these meanings cannot be changed.
- Keywords serve as basic building blocks for program statements. All key-words must be written in lowercase.

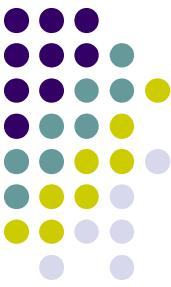
**Table 2.3 ANSI C Keywords**

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



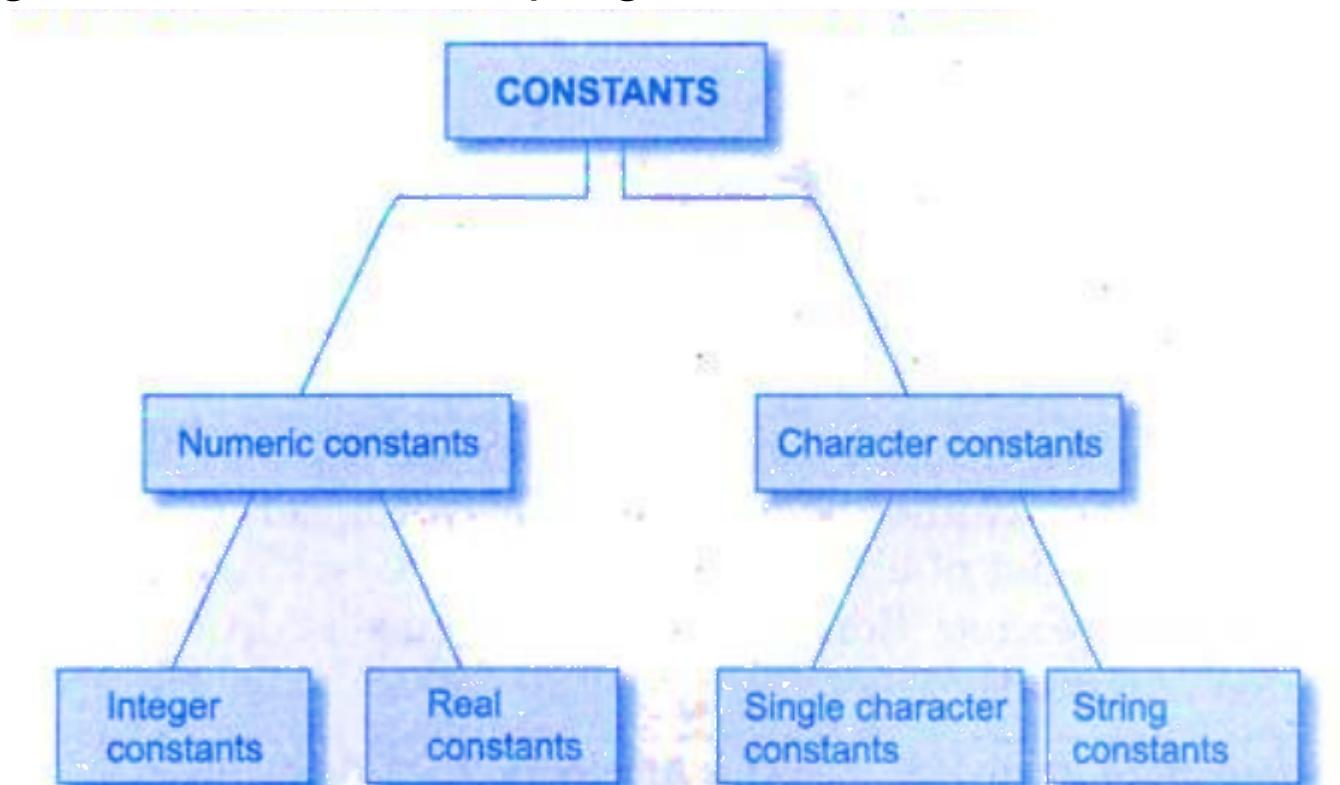
# Identifier

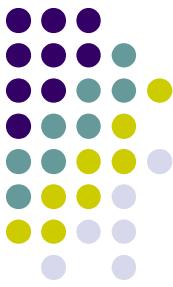
- Identifiers refer to the **names of variables, functions and arrays**. These are user-defined names and consist of a sequence of letters and digits, with a letter as a first character.
- Both uppercase and lowercase letters are permitted.
- Rules for identifiers
  - First character must be an alphabet (or underscore).
  - Must consist of only letters, digits or underscore.
  - Only first 31 characters are significant.
  - Cannot use a keyword.
  - Must not contain white space.



# Constants

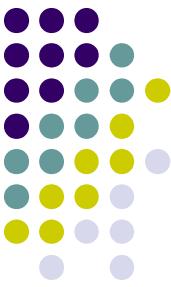
- Constants in C refer to fixed values that do not change during the execution of a program.





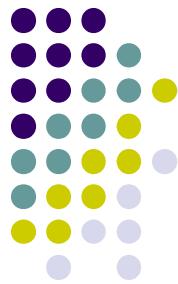
# Integer Constants

- Decimal Integer
  - Example : **123 -321 0 654321 +78**
  - Spaces, commas and non-digits are not permitted between digits
    - For example : **15 750 20,000 \$1000**
- An octal integer constant consists of any combination of digits from the set 0 through 7, with a leading 0.
  - examples : **037 0 0435 0551**
- A sequence of digits preceded by Ox or OX is considered as hexadecimal integer.
- The letter A through F represent the numbers 10 through 15.
- Following are the examples of valid hex integers:
  - Examples: **OX2 Ox9F OXbcd Ox**



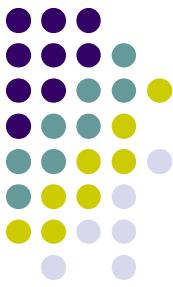
# Integer Constants

- The largest integer value that can be stored is machine-dependent. It is **32767** on **16-bit** machines and **2,147,483,647** on **32-bit** machines. It is also possible to store larger integer constants on these machines by appending qualifiers such as U,L and UL to the constants.
- Examples:
  - 56789**U** or 56789**u** (**unsigned integer**)
  - 987612347**UL** or 98761234u1 (**unsigned long integer**)
  - 9876543**L** or 98765431 (**long integer**)



# Real Constants

- These quantities are represented by numbers containing fractional parts like 17.548. Such numbers are called real (or floating point) constants. Further examples of real constants are:
  - 0.0083 -0.75 435.36 +247.0
- It is possible to omit digits before the decimal point, or digits after the decimal point. That is, 215. .95 -.71 +.5 are all valid real numbers.
- A real number may also be expressed in exponential (or scientific) notation. For example, the value 215.65 may be written as 2.1565e2 in exponential notation. e2 means multiply by 10<sup>2</sup>



# Real Constants

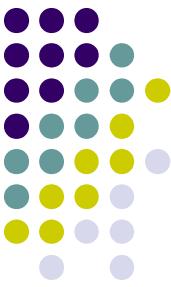
- The general form is:
  - mantissa e exponent
- The mantissa is either a real number expressed in decimal notation or an integer. The exponent is an integer number with an optional plus or minus sign. Since the exponent causes the decimal point to "float", this notation is said to represent a real number in floating point form. Examples of legal floating-point constants are: 0.65e4 12e-2 1.5e+5 3.18E3 -1.2E-1



# Real Constants

**Table 2.4 Examples of Numeric Constants**

<i>Constant</i>	<i>Valid ?</i>	<i>Remarks</i>
698354L	Yes	Represents long integer
25,000	No	Comma is not allowed
+5.0E3	Yes	(ANSI C supports unary plus)
3.5e-5	Yes	
7.1e 4	No	No white space is permitted
-4.5e-2	Yes	
1.5E+2.5	No	Exponent must be an integer
\$255	No	\$ symbol is not permitted
0X7B	Yes	Hexadecimal integer



# Single Character Constants

- A single character constant (or simply character constant) contains a single character enclosed within a pair of single quote marks.
- Example of character constants are: '5' , 'X'. ';' ''

## String Constants

- A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special characters and blank space.
- Examples are: "Hello!" "1987" "WELL DONE" "?...!" "5+3" "X"

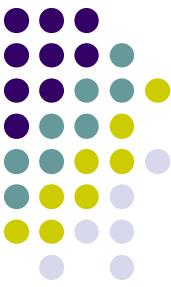


# Backslash Character

- C supports some special backslash character constants that are used in output functions.
- For example, the symbol '`\n`' stands for newline character.

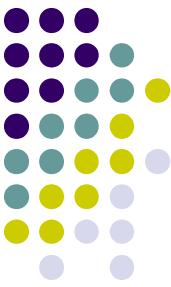
**Table 2.5 Backslash Character Constants**

<i>Constant</i>	<i>Meaning</i>
<code>'\a'</code>	audible alert (bell)
<code>'\b'</code>	back space
<code>'\f'</code>	form feed
<code>'\n'</code>	new line
<code>'\r'</code>	carriage return
<code>'\t'</code>	horizontal tab
<code>'\v'</code>	vertical tab
<code>'\'</code>	single quote
<code>'\"'</code>	double quote
<code>'?'</code>	question mark
<code>'\\'</code>	backslash
<code>'\0'</code>	null



# Variables

- A variable is a data name that may be used to store a data value. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times during execution.
- Some examples of such names are:
  - Average
  - height
  - Total
  - Counter\_I
  - class\_strength
- variable names may consist of **letters, digits, and the underscore( ) character,**



# Variables

## Conditions:

- They must begin with a letter. Some systems permit underscore as the first character.
- ANSI standard recognizes a length of 31 characters. However, length should not be normally more than eight characters.
- Uppercase and lowercase are significant. That is, the variable **Total** is not the same as **total** or **TOTAL**.
- It should not be a **keyword**.
- **White space** is not allowed.
- Some examples of valid variable names are:

▪ John	Value	T_raise
▪ Delhi	xl	ph_value
▪ mark	sum1	distance

- Invalid examples include:

▪ 123	(area)
▪ %	25th



# Variables

**Table 2.6 Examples of Variable Names**

<i>Variable name</i>	<i>Valid ?</i>	<i>Remark</i>
First_tag	Valid	
char	Not valid	char is a keyword
Price\$	Not valid	Dollar sign is illegal
group one	Not valid	Blank space is not permitted
average_number	Valid	First eight characters are significant
int_type	Valid	Keyword may be part of a name

# Data Types



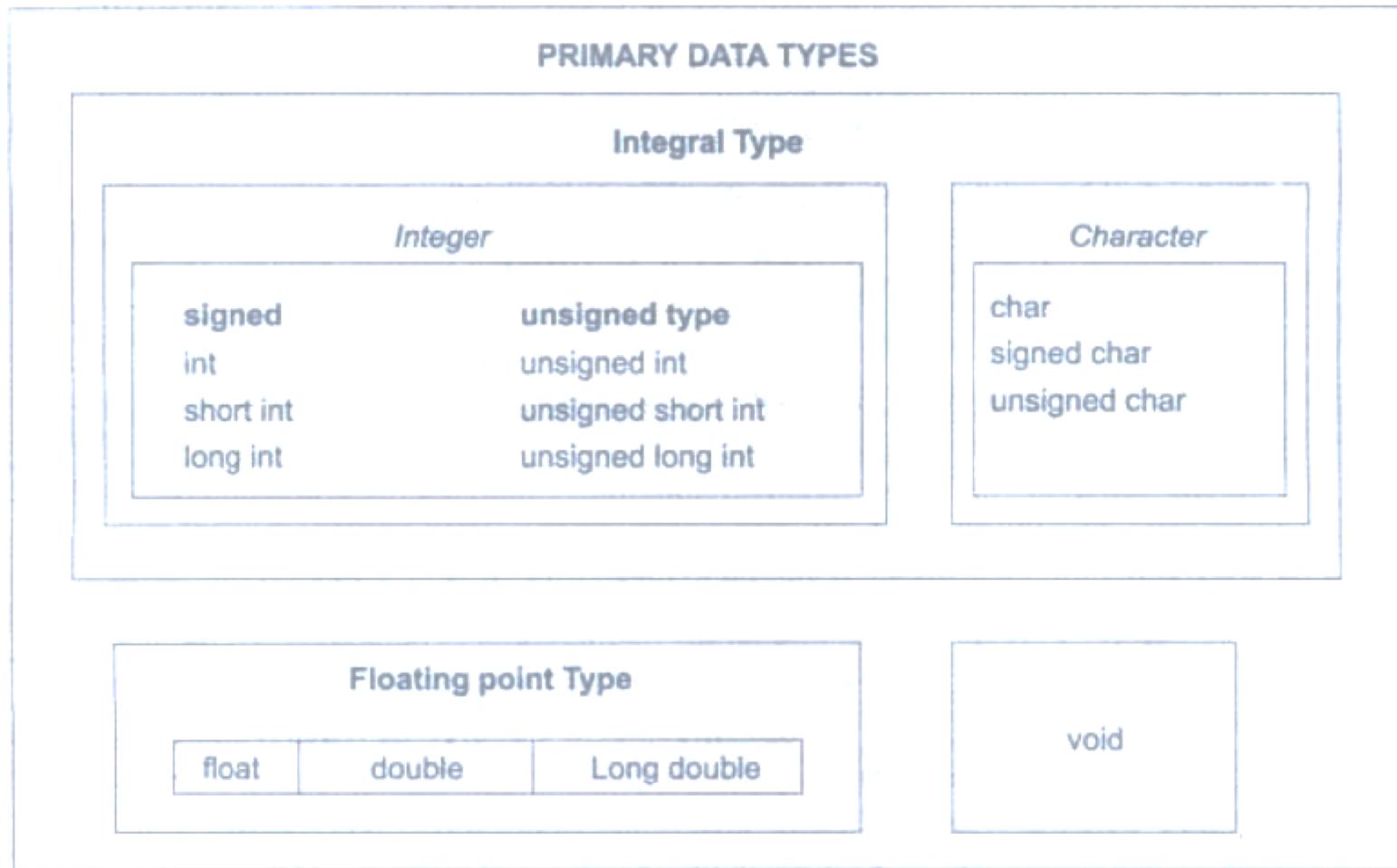
- ANSI C supports three classes of data types:
  - Primary (or fundamental) data types
  - Derived data types
  - User-defined data types

All C compilers support five fundamental data types, namely

- integer (int),
- character (char),
- floating point (float),
- double-precision floating point (double)
- and void.

Many of them also offer extended data types such as long int and long double.

# Data Types



# Data Types



**Table 2.7** *Size and Range of Basic Data Types on 16-bit Machines*

<i>Data type</i>	<i>Range of values</i>
char	-128 to 127
int	-32,768 to 32,767
float	3.4e-38 to 3.4e+38
double	1.7e-308 to 1.7e+308

# Data Types



- ANSI C supports three classes of data types:
  - Primary (or fundamental) data types
  - Derived data types
  - User-defined data types

All C compilers support five fundamental data types, namely

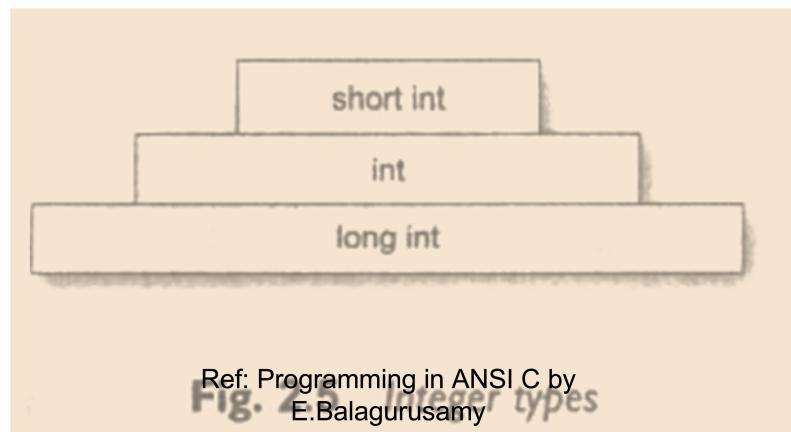
- integer (int),
- character (char),
- floating point (float),
- double-precision floating point (double)
- and void.

Many of them also offer extended data types such as long int and long double.



# Integer Types

- Integers are whole numbers with a range of values supported by a particular machine. The size of the integer value is limited to the range **-32768 to +32767** (that is,  $-2^{15}$  to  $+2^{15} - 1$ ).
- A signed integer uses one bit for sign and 15 bits for the magnitude of the number.
- Similarly, a **32 bit** word length can store an integer ranging from **-2,147,483,648 to 2,147,483,647**.
- C has three classes of integer storage,



# Integer Types



- Unlike signed integers, **unsigned** integers use all the bits for the magnitude of the number and are always positive. Therefore, for a **16 bit** machine, the range of unsigned integer numbers will be from **0 to 65,535**.
- We declare **long** and **unsigned** integers to increase the range of values. The use of qualifier **signed** on integers is optional because the default declaration assumes a signed number..

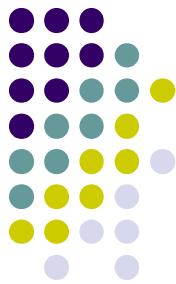
# Integer Types



**Table 2.8** Size and Range of Data Types on a 16-bit Machine

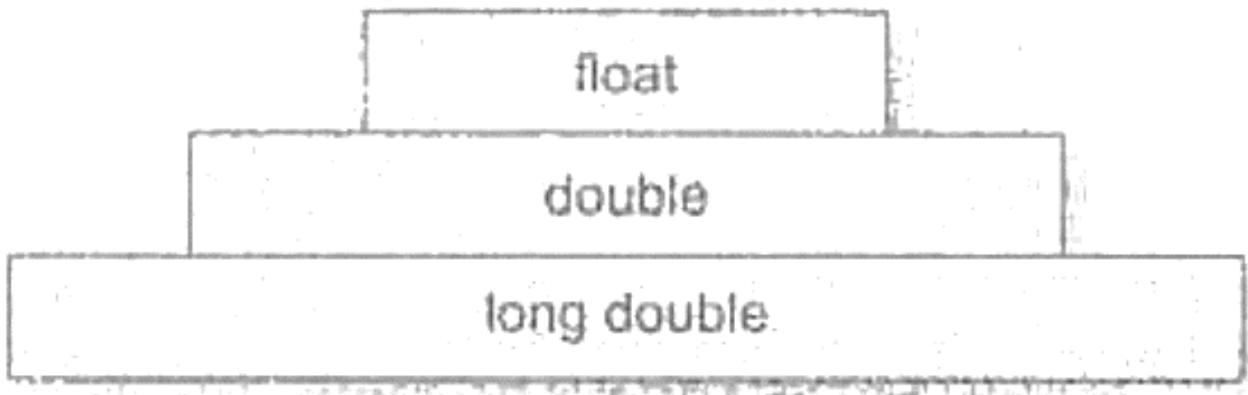
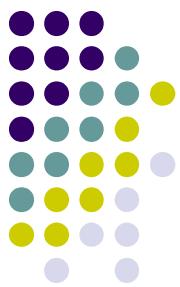
Type	Size (bits)	Range
char or signed char	8	-128 to 127
unsigned char	8	0 to 255
int or signed int	16	-32,768 to 32,767
unsigned int	16	0 to 65535
short int or signed short int	8	-128 to 127
unsigned short int	8	0 to 255
long int or signed long int	32	-2,147,483,648 to 2,147,483,647
unsigned long int	32	0 to 4,294,967,295
float	32	3.4E - 38 to 3.4E + 38
double	64	1.7E - 308 to 1.7E + 308
long double	80	3.4E - 4932 to 1.1E + 4932

# Floating Point Types



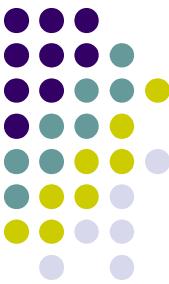
- Floating point (or real) numbers are stored in **32 bits** (on all 16 bit and 32 bit machines), with **6 digits** of precision.
- Floating point numbers are defined in C by the keyword **float**.
- When the accuracy provided by a float number is not sufficient, the type **double** can be used to define the number.
- A double data type number uses **64 bits** giving a precision of **14 digits**. These are known as **double precision numbers**.
- To extend the precision further, we may use **long double** which uses **80 bits**.

# Floating Point Types



**Fig. 2.6 Floating-point types**

# Void and Character Types

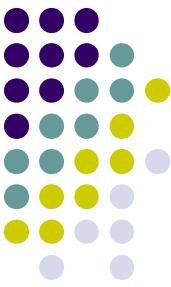


- The **void** type has no values. This is usually used to specify the type of functions. The type of a function is said to be **void** when it does not return any value to the calling function. It can also play the role of a generic type, meaning that it can represent any of the other standard types.
- A single character can be defined as a **character(char)** type data. Characters are usually stored in 8 bits (one byte) of internal storage. The qualifier **signed** or **unsigned** may be explicitly applied to char. While **unsigned chars** have values between 0 and 255, **signed chars** have values from -128 to 127.

# Assigning values to variables



- Variables are created for use in program statements such as,
- $\text{value} = \text{amount} + \text{inrate} * \text{amount};$
- In the first statement, the numeric value stored in the variable `inrate` is multiplied by the value stored in `amount` and the product is added to `amount`. The result is stored in the variable `value`. This process is possible only if the variables `amount` and `inrate` have already been given values. The variable `value` is called the target variable.



# Assignment Statement

- Values can be assigned to variables using the assignment operator `=` as follows:
  - `variable_name = constant;`
- Further examples are:
  - initial value = 0;
  - final value = 100;
  - balance = 75.84;
  - yes = 'x' ;
- C permits multiple assignments in one line.
- For example
  - `initial value = 0; final value = 100;`  
are valid statements.

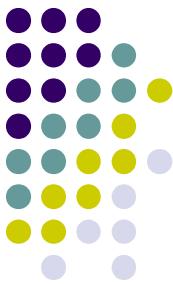
# Assignment Statement



- An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right. The statement `year = year + 1;` means that the 'new value' of `year` is equal to the 'old value' of `year` plus 1.
- During assignment operation, C converts the type of value on the right-hand side to the type on the left. This may involve truncation when real value is converted to an integer.
- It is also possible to assign a value to a variable at the time the variable is declared. This takes the following form:

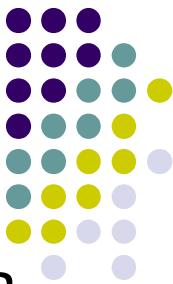
`data-type variable_name = constant;`

# Assignment Statement



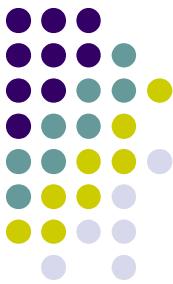
- Some examples are:
  - int final value = 100;
  - char yes = 'x';
  - double balance = 75.84;
- The process of giving initial values to variables is called **initialization**. C permits the initialization of **more than one variables** in one statement using multiple assignment operators.
- For example the statements
  - p = q = s = 0;
  - x = y = z = MAX; are valid.
- The first statement initializes the variables p, q, and s to zero while the second initializes x, y, and z with MAX.  
**MAX is a symbolic constant defined at the beginning.**

# Reading Data from Keyboard



- Another way of giving values to variables is to input data through keyboard using the **scanf** function. It is a general **input function** available in C and is very similar in concept to the **printf** function.
- It works much like an **INPUT statement** in BASIC. The general format of scanf is as follows:  
`scanf("control string", &variable1 ,&variable2,.. );`
- The control string contains the format of data being received.
- The ampersand symbol **&** before each variable name is an operator that specifies the variable name's **address**.  
**We must always use this operator, otherwise unexpected results may occur.**

# Reading Data from Keyboard

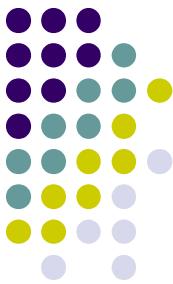


- Let us look at an example:

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

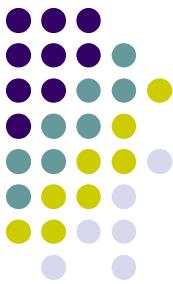
- When this statement is encountered by the computer, the execution stops and waits for the value of the variable number to be typed in.
- Since the control string "%d" specifies that an integer value is to be read from the terminal, we have to type in the value in integer form. Once the number is typed in and the 'Return' Key is pressed, the computer then proceeds to the next statement.
- Thus, the use of scanf provides an interactive feature and makes the program 'user friendly'. The value is assigned to the variable number.

# Defining Symbolic Constants



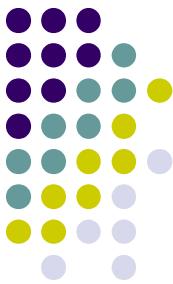
- We often use certain unique constants in a program. These constants may appear repeatedly in a number of places in the program.
- One example of such a constant is 3.142, representing the value of the mathematical constant "pi".
- Another example is the total number of students whose mark-sheets are analysed by a 'test analysis program'. The number of students, say 50, may be used for calculating the class total, class average, standard deviation, etc. **We face two problems in the subsequent use of such programs.**
- These are
  - problem in modification of the program and
  - problem in understanding the program.

# Defining Symbolic Constants



- **Modifiability** We may like to change the value of “pi” from 3.142 to 3.14159 to improve the accuracy of calculations or the number 50 to 100 to process the test results of another class. In both the cases, we will have to search throughout the program and explicitly change the value of the constant wherever it has been used. If any value is left unchanged, the program may produce disastrous outputs.
- **Understandability** When a numeric value appears in a program, its use is not always clear, especially when the same value means different things in different places.
- **For example, the number 50 may mean the number of students at one place and the 'pass marks' at another place of the same program.**

# Defining Symbolic Constants



- We may forget what a certain number meant, when we read the program some days later.
- Assignment of such constants to a symbolic name frees us from these problems.
- For example, we may use the name **STRENGTH** to define the number of students and **PASS\_MARK** to define the pass marks required in a subject.
- Constant values are assigned to these names at the beginning of the program. Subsequent use of the names **STRENGTH** and **PASS\_MARK** in the program has the effect of causing their defined values to be automatically substituted at the appropriate points. A constant is defined as follows:

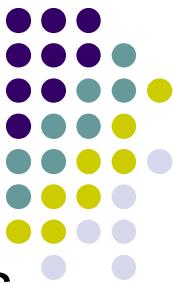
**#define symbolic-name value of constant**

# Defining Symbolic Constants



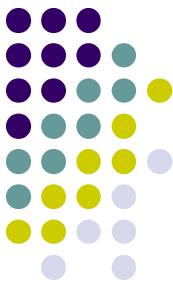
- Valid examples of constant definitions are:
  - `#define STRENGTH 100`
  - `#define PASS MARK 50`
  - `#define MAX 700`
  - `#define PI 3.14159`
- Symbolic names are sometimes called constant identifiers. Since the symbolic names are constants (not variables), they do not appear in declarations. The following rules apply to a **#define statement** which define a symbolic constant:

# Defining Symbolic Constants



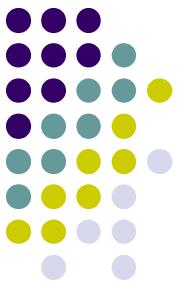
- Symbolic names have the same form as variable names. (Symbolic names are written in CAPITALS to visually distinguish them from the normal variable names, which are written in lowercase letters. This is only a convention, not a rule.)
- No blank space between the **pound sign '#** and the word **define** is permitted.
- **#** must be the first character in the line.
- A blank space is required between **#define** and **symbolic name** and between the **symbolic name** and the constant.
- **#define** statements must not end with a semicolon.

# Defining Symbolic Constants



- After definition, the symbolic name should not be assigned any other value within the program by using an assignment statement. **For example, `STRENGTH = 200;` is illegal.**
- **Symbolic names are NOT declared for data types.** Its data type depends on the type of constant.
- **#define statements may appear anywhere** in the program but before it is referenced in the program (the usual practice is to place them in the beginning of the program).
- **#define statement** is a pre-processor compiler directive and is much more powerful than what has been mentioned here.

# Defining Symbolic Constants



**Table 2.11 Examples of Invalid #define Statements**

<i>Statement</i>	<i>Validity</i>	<i>Remark</i>
#define X = 2.5	Invalid	'=' sign is not allowed
# define MAX 10	Invalid	No white space between # and define
#define N 25;	Invalid	No semicolon at the end
#define N 5, M 10	Invalid	A statement can define only one name.
#Define ARRAY 11	Invalid	define should be in lowercase letters
#define PRICES 100	Invalid	\$ symbol is not permitted in name

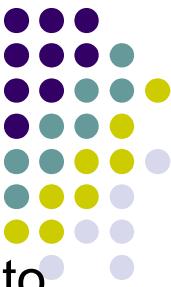


# Declaring a variable as constant

- We may like the value of certain variables to remain constant during the execution of a program. We can achieve this by declaring the variable with the qualifier **const** at the time of initialization.
- Example:

```
const int class size = 40;
```

- **const** is a new data type qualifier defined by ANSI standard. This tells the compiler that the value of the **int** variable **class\_size** must not be modified by the program.



# Declaring a variable as volatile

- ANSI standard defines another qualifier **volatile** that could be used to tell explicitly the compiler that a variable's value may be changed at any time by some external sources (from outside the program).
- For example:

```
volatile int date;
```

- The value of date may be altered by some external factors even if it does not appear on the left-hand side of an assignment statement.
- When we declare a variable as volatile, the compiler will examine the value of the variable each time it is encountered to see whether any external alteration has changed the value. Remember that the value of a variable declared as volatile can be modified by its own program as well. If we wish that the value must not be modified by the program while it may be altered by some other process, then we may declare the variable as both const and volatile as shown below:

```
volatile const int location = 100;
```

Ref: Programming in ANSI C by  
E.Balagurusamy



# Overflow and Underflow of Data

- Problem of data overflow occurs when the value of a variable is either too big or too small for the data type to hold. The largest value that a variable can hold also depends on the machine.
- Since floating-point values are rounded off to the number of significant digits allowed (or specified), an overflow normally results in the largest possible real value, whereas an underflow results in zero.
- Integers are always exact within the limits of the range of the integral data types used. However, an overflow which is a serious problem may occur if the data type does not match the value of the constant.
- C does not provide any warning or indication of integer overflow. It simply gives incorrect results. (Overflow normally produces a negative number.) We should therefore exercise a greater care to define correct data types for handling the input/ output values.



# Introduction to C Programming

## Chapter 3

### Operators and Expressions

# Introduction

- C supports a rich set of built-in operators. We have already used several of them, **such as =, +, \*, & and <**.
- An operator is a symbol that tells the computer to perform certain mathematical or logical manipulations.
- Operators are used in programs **to manipulate data and variables**. They usually form a part of the mathematical or logical expressions.

# Introduction

- C operators can be classified into a number of categories. They include:
  - Arithmetic operators
  - Relational operators
  - Logical operators
  - Assignment operators
  - Increment and decrement operators
  - Conditional operators
  - Bitwise operators
  - Special operators
- An **expression** is a sequence of operands and operators that reduces to a single value. For example,
- $10 + 15$  is an expression whose value is 25. The value can be any type other than void.

# Arithmetic Operators

- C provides all the basic arithmetic operators. The operators **+**, **\***, and **/** all work the same way as they do in other languages.
- Integer division truncates any fractional part. The modulo division operation produces the remainder of an integer division.
- Examples of use of arithmetic operators are:  
**a-b      a+b      a \* b      a/b      a % b      -a \* b**
- Here **a** and **b** are variables and are known as **operands**.
- The modulo division operator **%** cannot be used on floating point data. Note that C does not have an operator for exponentiation.

# Arithmetic Operators

**Table 3.1 Arithmetic Operators**

<i>Operator</i>	<i>Meaning</i>
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division

# Integer Arithmetic

- When both the operands in a single arithmetic expression such as  $a+b$  are integers, the expression is called an integer expression, and the operation is called integer arithmetic.
- Integer arithmetic always yields an integer value. The largest integer value depends on the machine, as pointed out earlier.
- In the above examples, if a and b are integers,
- then for  $a = 14$  and  $b = 4$  we have the following results:
  - $a - b = 10$
  - $a + b = 18$
  - $a * b = 56$
  - $a / b = 3$  (decimal part truncated)
  - $a \% b = 2$  (remainder of division)

# Integer Arithmetic

- During integer division, if both the operands are of the same sign, the result is truncated towards zero. If one of them is negative, the direction of truncation is implementation dependent.
- That is,  $6/7 = 0$  and  $-6/-7 = 0$  but  $-6/-7$  may be zero or -1. (Machine dependent)
- Similarly, during modulo division, the sign of the result is always the sign of the first operand (the dividend). That is
  - $-14 \% 3 = -2$
  - $-14 \% -3 = -2$
  - $14 \% -3 = 2$

# Real Arithmetic

- An arithmetic operation involving only real operands is called real arithmetic.
- A real operand may assume values either **in decimal** or **exponential** notation. Since floating point values are rounded to the number of significant digits permissible, the final value is an approximation of the correct result.
- If  $x$ ,  $y$ , and  $z$  are floats, then we will have:
  - $x = 6.0/7.0 = 0.857143$
  - $y = 1.0/3.0 = 0.333333$
  - $z = -2.0/3.0 = -0.666667$

The operator % cannot be used with real operands.

# Mixed-mode Arithmetic

- When one of the operands is real and the other is integer, the expression is called a mixed-mode arithmetic expression.
- If either operand is of the real type, then only the real operation is performed and the result is always a real number.
- Thus       $15/10.0 = 1.5$
- whereas

$$15/10 = 1$$

# Relational Operators

- We often compare two quantities and depending on their relation, take certain decisions. For example, we may compare the age of two persons, or the price of two items, and so on.
- These comparisons can be done with the help of relational operators. An expression such as
- $a < b$  or  $1 < 20$  containing a relational operator is termed as a **relational expression**.
- **The value of a relational expression is either one or zero.** It is one if the specified relation is true and zero if the relation is false. For example
  - $10 < 20$  is true
  - but  $20 < 10$  is false
  - C supports six relational operators in all.

# Relational Operators

**Table 3.2 Relational Operators**

<i>Operator</i>	<i>Meaning</i>
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
==	is equal to
!=	is not equal to

# Relational Operators

- some examples of simple relational expressions and their values:
  - $4.5 \leq 10$  TRUE
  - $4.5 < -10$  FALSE
  - $-35 \geq 0$  FALSE
  - $10 < 7+5$  TRUE
  - $a+b = c+d$  TRUE
- only if the sum of values of a and b is equal to the sum of values of c and d. When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared. That is, arithmetic operators have a higher priority over relational operators. Relational expressions are used in decision statements such as if and while to decide the course of action of a running program.

# Relational Operators

## Relational Operator Complements

Among the six relational operators, each one is a complement of another operator.

>	is complement of	<=
<	is complement of	>=
==	is complement of	!=

We can simplify an expression involving the *not* and the *less than* operators using the complements as shown below:

Actual one	Simplified one
$!(x < y)$	$x \geq y$
$!(x > y)$	$x \leq y$
$!(x \neq y)$	$x == y$
$!(x \leq y)$	$x > y$
$!(x \geq y)$	$x < y$
$!(x == y)$	$x \neq y$

# Logical Operators

- In addition to the relational operators, C has the following three logical operators.

&& meaning logical AND

|| meaning logical OR

! meaning logical NOT

- The logical operators && and || are used when we want to test more than one condition and make decisions.
- An example is:  $a > b \&& x = 1$
- An expression of this kind, which combines two or more relational expressions, is termed as a **logical expression** or a **compound relational expression**. Like the simple relational expressions, a logical expression also yields a value of one or zero, according to the truth table shown in Table 3.3.

# Logical Operators

- The logical expression given above is true only if  $a > b$  is true and  $x == 10$  is true. If either (or both) of them are false, the expression is false

**Table 3.3 Truth Table**

op-1	op-2	Value of the expression	
		op-1 && op-2	op-1    op-2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

# Logical Operators

- Some examples of the usage of logical expressions are: 1.  
`if (age > 55 && salary < 1000)`
- 2. `if (number < 0 || number > 100)`

Relative precedence of the relational and logical operators is as follows:

Highest        !

    > >=    < <=

    ==    !=

    &&

Lowest        ||

# Assignment Operators

- Assignment operators are used to assign the result of an expression to a variable. C has a set of 'shorthand' assignment operators of the form

$v \text{ op} = \text{exp};$

Where  $v$  is a variable,  $\text{exp}$  is an expression and  $\text{op}$  is a binary arithmetic operator. The operator  $\text{op}=$  is known as the shorthand assignment operator.

The assignment statement

$v \text{ op} = \text{exp};$

Is equivalent to

$v = v \text{ op} (\text{exp});$

with  $v$  evaluated only once. Consider an example

$x += y+1;$

# Assignment Operators

This is same as the statement

`x = x + (y+1);`

The shorthand operator `+=` means 'add y+1 to x or increment x by y+1'. For `y =2`, the above statement becomes

`x +=3;`

And when this statement is executed, 3 is added to x. if the old value of x is, say 5, then the new value of x is 8.

Statement with simple assignment operator	Statement with shorthand operator
<code>a= a+1</code>	<code>a +=1</code>
<code>a= a-1</code>	<code>a -= 1</code>
<code>a= a * (n-1)</code>	<code>a *= n+1</code>
<code>a= a / (n+1)</code>	<code>a /= n+1</code>
<code>a= a % b</code>	<code>a %= b</code>

# Assignment Operators

- The use of shorthand assignment operators has three advantages:
  - What appears on the left-hand side need not be repeated and therefore it becomes easier write.
  - The statement is more concise and easier to read
  - The statement is more efficient.

These advantages may be appreciated if we consider a slightly more involved statement like

value (5\*j - 2) = value (5\*j-2) + delta;

With the help of the `+=` operator this can be written as follows:

value (5\*j -2) += delta;

It is easier to read and understand and is more efficient because the expression `5*j-2` is evaluated only once.

# Increment & decrement operator

- C allows two very useful operators not generally found in other languages. These are the increment and decrement operators:

++ and --

- The operator **++ adds 1 to the operand**, while **-- subtracts 1**. Both are unary operators and takes the following form:
  - `++m`; is equivalent to `m = m+1`; (or `m += 1`);
  - `--m`; is equivalent to `m = m-1`; (or `m -= 1`);
- We use the increment and decrement statements in for and while loops extensively.
- While `++m` and `m++` mean the same thing when they form statements independently, they behave differently when they are used in expressions on the right-hand side of an assignment statement.

# Increment & decrement operator

- Consider the following:
  - $m = 5;$
  - $y = ++m;$
- In this case, the value of  $y$  and  $m$  would be 6. Suppose, if we rewrite the above statements as
  - $m = 5;$
  - $y = m++;$
- then, the value of  $y$  would be 5 and  $m$  would be 6.
- *A prefix operator first adds 1 to the operand and then the result is assigned to the variable on left.*
- *On the other hand, a postfix operator first assigns the value to the variable on left and then increments the operand.*

# Increment & decrement operator

- Similar is the case, when we use ++ (or - -) in subscripted variables. That is, the statement

`a[i++] =10;`

- is equivalent to

`a[i] =10;`

`i = i+1;`

- The increment and decrement operators can be used in complex statements. Example:

`m = n++ -j+10;`

- Old value of `n` is used in evaluating the expression. `n` is incremented after the evaluation. Some compilers require a space on either side of `n++` or `++n`.

# Increment & decrement operator

## Rules for ++ and -- Operators

- Increment and decrement operators are unary operators and they require variable as their operands.
- When postfix ++ (or --) is used with a variable in an expression, the expression is evaluated first using the original value of the variable and then the variable is incremented (or decremented) by one.
- When prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable.
- The precedence and associativity of ++ and -- operators are the same as those of unary + and unary -.

# Conditional operator

- A ternary operator pair "? :" is available in C to construct conditional expressions of the form

`exp1 ? exp2 : exp3`

- where exp1, exp2, and exp3 are expressions.
- The operator ? : works as follows: exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression.
- If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Note that only one of the expressions (either exp2 or exp3) is evaluated

# Conditional Operator

- For example, consider the following statements.

```
a = 10;  
b = 15;  
x= (a > b) ?a :b;
```

- In this example, x will be assigned the value of b. This can be achieved using the if. Else statements as follows:

```
if (a > b)  
    x = a;  
else  
    x = b;
```

# Bitwise Operator

- C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table 3.5 lists the bitwise operators and their meanings.

**Table 3.5 Bitwise Operators**

<i>Operator</i>	<i>Meaning</i>
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

# Bitwise operator

- C has a distinction of supporting special operators known as bitwise operators for manipulation of data at bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table 3.5 lists the bitwise operators and their meanings.

**Table 3.5 Bitwise Operators**

<i>Operator</i>	<i>Meaning</i>
&	bitwise AND
	bitwise OR
^	bitwise exclusive OR
<<	shift left
>>	shift right

# Special operator

- The **comma operator** can be used to link the related expressions together. A comma-linked list of expressions are evaluated left to right and the value of right-most expression is the value of the combined expression. For example, the statement

`value = (x = 10, y = 5, x+y);`

- first assigns the value 10 to x, then assigns 5 to y, and finally assigns 15 (i.e.  $10 + 5$ ) to value.
- Since comma operator has the lowest precedence of all operators, the parentheses are necessary. Some applications of comma operator are:

# Special operator

- The **sizeof** is a compile time operator and, when used with an operand, it returns the number of bytes the operand occupies.
- The operand may be a variable, a constant or a data type qualifier.
- Examples:

m = **sizeof** (sum);

n = **sizeof** (long int);

k = **sizeof** (235L);

- The **sizeof** operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of a program.

# Arithmetic Expressions

- An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.
- C can handle any complex mathematical expressions. Some of the examples of C expressions are shown in Table 3.6

**Table 3.6 Expressions**

<i>Algebraic expression</i>	<i>C expression</i>
$a \times b - c$	$a * b - c$
$(m+n)(x+y)$	$(m+n) * (x+y)$
$\left( \frac{ab}{c} \right)$	$a * b/c$
$3x^2 + 2x + 1$	$3 * x * x + 2 * x + 1$
$\left( \frac{x}{y} \right) + c$	$x/y+c$

# Evaluation of Expressions

- Expressions are evaluated using an assignment statement of the form:
  - variable = expression;
  - Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and the result then replaces the previous value of the variable on the left-hand side. All variables used in the expression must be assigned values before evaluation is attempted.
- Examples of evaluation statements are

x = a \* b - c;

y = b / c \* a;

z = a-b/ c+d;

# Cont.

- The blank space around an operator is optional and adds only to improve readability. When these statements are used in a program, the variables a, b, c, and d must be defined before they are used in the expressions.

# Precedence of arithmetic operators

- An arithmetic expression without parentheses will be evaluated from left to right using the rules of precedence of operators. There are two distinct priority levels of arithmetic operators in C:
- High priority \* / %
- Low priority + —
- The basic evaluation procedure includes 'two' left-to-right passes through the expression.
- During the first pass, the high priority operators (if any) are applied as they are encountered.
- During the second pass, the low priority operators (if any) are applied as they are encountered.

# Cont.

- Consider the following evaluation statement

$$x = a - b / 3 + c * 2 - 1$$

- When  $a = 9$ ,  $b = 12$ , and  $c = 3$ , the statement becomes

$$x = 9 - 12 / 3 + 3 * 2 - 1$$

- and is evaluated as follows

- **First pass**

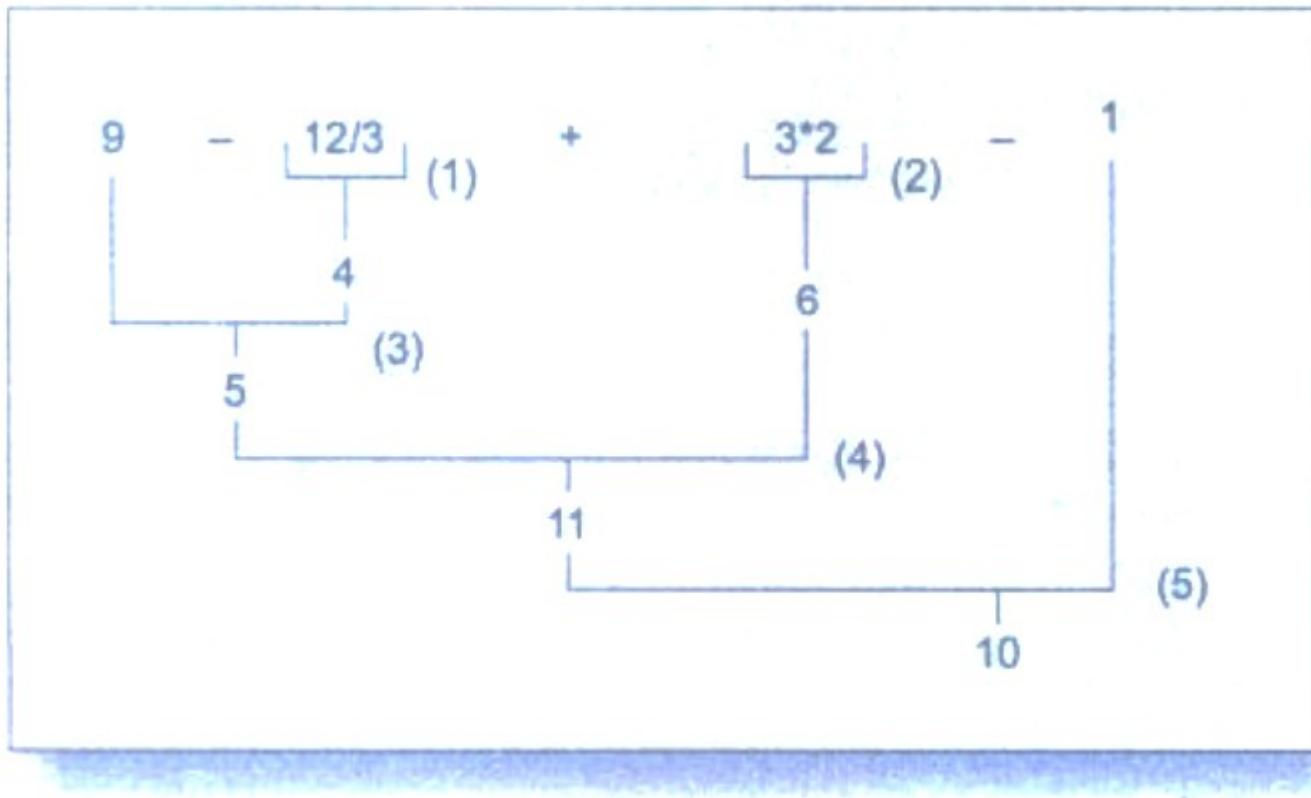
- Step1:  $x = 9 - 4 + 3 * 2 - 1$
- Step2:  $x = 9 - 4 + 6 - 1$

- **Second pass**

- Step3:  $x = 5 + 6 - 1$
- Step4:  $x = 11 - 1$
- Step5:  $x = 10$

# Cont.

- These steps are illustrated in Fig. 3.5. The numbers inside parentheses refer to step numbers.



**Fig. 3.5 Illustration of hierarchy of operations**

# Cont.

- However, the order of evaluation can be changed by introducing parentheses into an expression. Consider the same expression with parentheses as shown below:

$$9 - 1 \ 2/(3+3)*(2-1)$$

- Whenever parentheses are used, the expressions within parentheses assume highest priority.
- If two or more sets of parentheses appear one after another as shown above, the expression contained in the left-most set is evaluated first and the right-most in the last. Given below are the new steps.

# Cont.

- First pass

**Step1:**  $9-12/6 * (2-1)$

**Step2:**  $9-12/6 * 1$

- Second pass

**Step3:**  $9-2 * 1$

**Step4:**  $9-2$

- Third pass

**Step5:**  $7$

- This time, the procedure consists of three left-to-right passes. However, the number of evaluation steps remains the same as 5 (i.e. equal to the number of arithmetic operators).

# Cont.

- Parentheses may be nested, and in such cases, evaluation of the expression will proceed outward from the innermost set of parentheses. Just make sure that every opening parenthesis has a matching closing parenthesis.  
For example

$$9 - (12/(3+3) * 2) - 1 = 4$$

whereas

$$9 - ((12/3) + 3 * 2) - 1 = -2$$

- While parentheses allow us to change the order of priority, we may also use them to improve understandability of the program. When in doubt, we can always add an extra pair just to make sure that the priority assumed is the one we require.

## Rules for Evaluation of Expression

- First, parenthesized sub expression from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- The precedence rule is applied in determining the order of application of operators in evaluating sub-expressions
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.
- Arithmetic expressions are evaluated from left to right using the rules of precedence.
- When parentheses are used, the expressions within parentheses assume highest priority.

# Some computational problems

- When expressions include real values, then it is important to take necessary precautions to guard against certain computational errors.
- We know that the computer gives approximate values for real numbers and the errors due to such approximations may lead to serious problems.
- For example, consider the following statements:

$a = 1.0/3.0;$

$b = a * 3.0;$

- We know that  $(1.0/3.0) * 3.0$  is equal to 1. But there is no guarantee that the value of b computed in a program will equal 1.

# Cont.

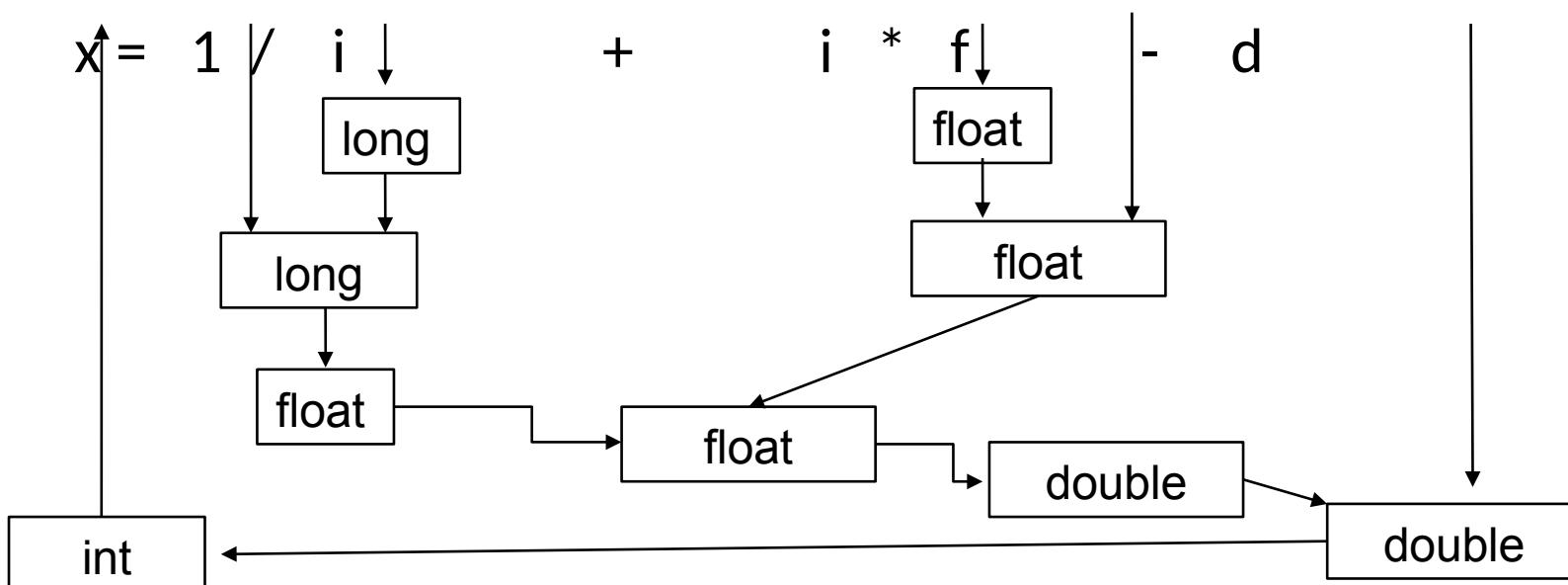
- **Another problem is division by zero.** On most computers, any attempt to divide a number by zero will result in abnormal termination of the program. In some cases such a division may produce meaningless results. Care should be taken to test the denominator that is likely to assume zero value and avoid any division by zero.
- **The third problem is to avoid overflow or underflow errors.** It is our responsibility to guarantee that operands are of the correct type and range, and the result may not produce any overflow or underflow.

# Implicit type conversion

- C permits mixing of constants and variables of different types in an expression.
- C automatically converts any intermediate values to the proper type so that the expression can be evaluated without losing any significance.
- During evaluation it adheres to very strict rules of type conversion. If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds.
- The result is of the higher type. A typical type conversion process is illustrated in Fig 3.8

# Implicit type conversion

int	i,x;
float	f;
double	d;
long int	1;



# Cont.

Given below is the sequence of rules that are applied while evaluating expressions.

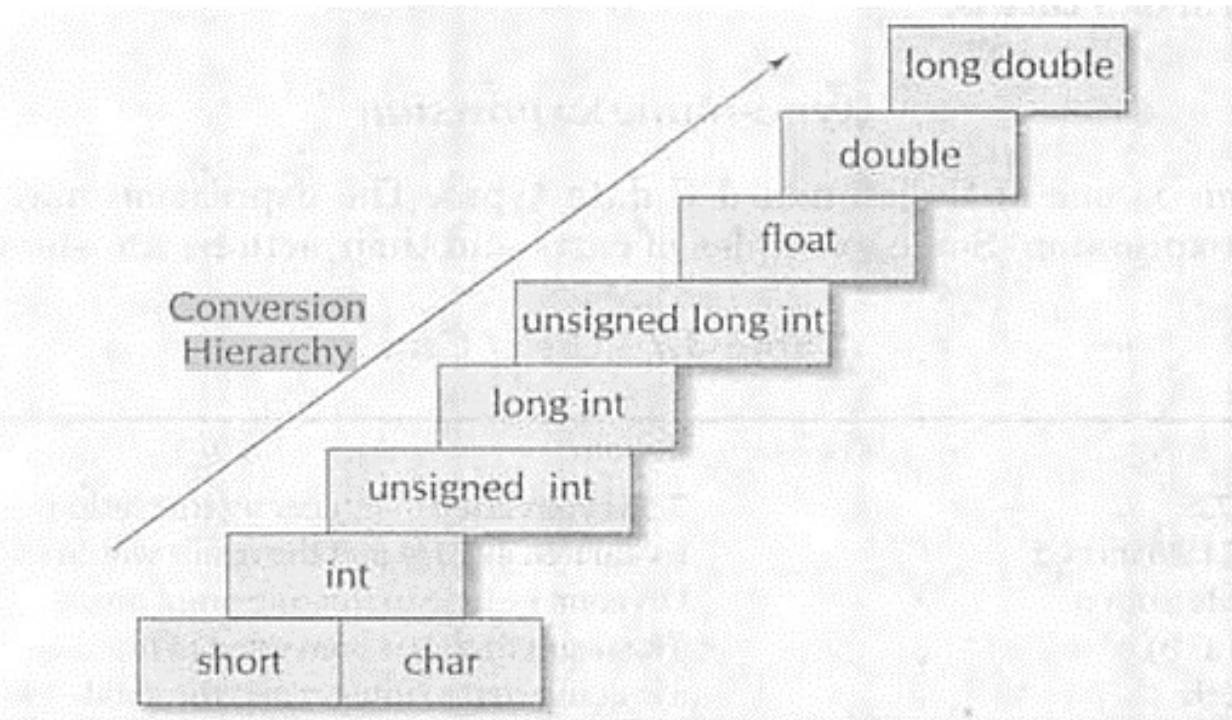
- All **short** and **char** are automatically converted to **int**; then
  1. If one of the operands is **long double**, the other will be converted to **long double** and the result will be **long double**.
  2. Else if one of the operands is **double**, the other will be converted to **double** and the result will be **double**;
  3. Else, if one of the operands is **float**, the other will be converted to **float** and the result will be **float**;
  4. Else, if one of the operands is **unsigned long int**, the other will be converted to **unsigned long int** and the result will be **unsigned long int**;

# Cont.

5. Else, if one of the operands is **long int** and the other is **unsigned int**, then
  - a) If **unsigned int** can be converted to **long int**, the **unsigned int** operand will be converted as such and the result will be **long int**;
  - b) Else, both operands will be converted to **unsigned long int** and the result will be **unsigned long int**;
6. Else, if one of the operands is **long int**, the other will be converted to **long int** and the result will be **long int**.
7. Else, if one of the operands is **unsigned int**, the other will be converted to **unsigned int** and the result will be **unsigned int**.

# Conversion Hierarchy

- C uses the rule that, in all expressions except assignments, any implicit type conversion are made from a lower size type to a higher size type as shown below:



# Conversion Hierarchy

- some versions of C automatically convert all floating-point operands to double precision.
- The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it. However, the following changes are introduced during the final assignment.
  1. **float** to **int** causes truncation of the fractional part.
  2. **double** to **float** causes rounding of digits.
  3. **long int** to **int** causes dropping of the excess higher order bits.

# Explicit Conversion

- There are instances when we want to force a type conversion in a way that is different from the automatic conversion.
- Consider, for example, the calculation of ratio of females to males in a town.

`ratio = female_number/male_number`

- Since `female_number` and `male_number` are declared as `integers` in the program, the decimal part of the result of the division would be lost and `ratio` would represent a wrong figure.
  - This problem can be solved by converting locally one of the variables to the floating point as shown below:
- `ratio = (float) female_number/male_number`

# Explicit Conversion

- The operator (`float`) converts the `female_number` to **floating point** for the purpose of evaluation of the expression.
- Then using the rule of **automatic conversion**, the division is performed in floating point mode, thus retaining the fractional part of result.
- Note that in no way does the operator (`float`) affect the value of the variable `female_number`. And also, the type of `female_number` remains as `int` in the other parts of the program.
- *The process of such a local conversion is known as explicit conversion or casting a value.*

# Explicit Conversion

- The general form of a cast is:

*(type-name)expression*

- where type-name is one of the standard C data types. The expression may be a constant, variable or an expression.

**Table 3.7 Use of Casts**

<i>Example</i>	<i>Action</i>
<code>x = (int) 7.5</code>	7.5 is converted to integer by truncation.
<code>a = (int) 21.3/(int)4.5</code>	Evaluated as 21/4 and the result would be 5.
<code>b = (double)sum/n</code>	Division is done in floating point mode.
<code>y = (int) (a+b)</code>	The result of a+b is converted to integer.
<code>z = (int)a+b</code>	a is converted to integer and then added to b.
<code>p = cos((double)x)</code>	Converts x to double before using it.

# Operator Precedence & Associativity

- Each operator, in C has a precedence associated with it. This precedence is used to determine how an expression involving more than one operator is evaluated.
- There are distinct levels of precedence and an operator may belong to one of these levels. The operators at the higher level of precedence are evaluated first. The operators of the same precedence are evaluated either from 'left to right' or from 'right to left', depending on the level.
- *This is known as the associativity property of an operator.*
- Table 3.8 provides a complete list of operators, their precedence levels, and their rules of association.

# Operator Precedence & Associativity

- The groups are listed in the order of decreasing precedence. Rank 1 indicates the highest precedence level and 15 the lowest. Consider the following conditional statement:

if (x == 10 + 15 && y < 10)

- The precedence rules say that the **addition operator** has a **higher priority** than the **logical operator** (**&&**) and the **relational operators** (**==** and **<**).
- Therefore, the addition of 10 and 15 is executed first. This is equivalent to :

if (x == 25 && y < 10)

- The next step is to determine whether a is equal to 25 and y is less than 10.

# Operator Precedence & Associativity

- If we assume a value of 20 for x and 5 for v, then

x = 25 is FALSE (0)

y < 10 is TRUE (1)

- Note that since the operator < enjoys a **higher priority** compared to ==, y < 10 is tested first and then x == 25 is tested.
- Finally we get:

if (FALSE && TRUE)

# Operator Precedence & Associativity

Operator	Description	Associativity	Rank
( )	Function call	Left to right	1
[ ]	Aray element reference		
+	Unary plus		
-	Unary minus	Right to left	2
++	Increment		
--	Decrement		
!	Logical negation		
~	Ones complement		
*	Pointer reference (indirection)		
&	Address		
sizeof	Size of an object		
(type)	Type cast (conversion)		
*	Multiplication	Left to right	3
/	Division		
%	Modulus		
+	Addition	Left to right	4
-	Subtraction		
<<	Left shift	Left to right	5
>>	Right shift		
<	Less than	Left to right	6
<=	Less than or equal to		
>	Greater than		
>=	Greater than or equal to		

# Operator Precedence & Associativity

<code>==</code>	Equality	Left to right	7
<code>!=</code>	Inequality		
<code>&amp;</code>	Bitwise AND	Left to right	8
<code>^</code>	Bitwise XOR	Left to right	9
<code> </code>	Bitwise OR	Left to right	10
<code>&amp;&amp;</code>	Logical AND	Left to right	11
<code>  </code>	Logical OR	Left to right	12
<code>?:</code>	Conditional expression	Right to left	13
<code>=</code> <code>* = /= %=</code> <code>+= -= &amp;=</code> <code>^=  =</code> <code>&lt;&lt;= &gt;&gt;=</code>	Assignment operators	Right to left	14
<code>,</code>	Comma operator	Left to right	15

# Mathematical Functions

**Table 3.9 Math functions**

<i>Function</i>	<i>Meaning</i>
<b>Trigonometric</b>	
acos(x)	Arc cosine of x
asin(x)	Arc sine of x
atan(x)	Arc tangent of x
atan 2(x,y)	Arc tangent of x/y
cos(x)	Cosine of x
sin(x)	Sine of x
tan(x)	Tangent of x
<b>Hyperbolic</b>	
cosh(x)	Hyperbolic cosine of x
sinh(x)	Hyperbolic sine of x
tanh(x)	Hyperbolic tangent of x
<b>Other functions</b>	
ceil(x)	x rounded up to the nearest integer
exp(x)	e to the x power ( $e^x$ )
fabs(x)	Absolute value of x.
floor(x)	x rounded down to the nearest integer
fmod(x,y)	Remainder of x/y
log(x)	Natural log of x, $x > 0$
log10(x)	Base 10 log of x, $x > 0$
pow(x,y)	x to the power y ( $x^y$ )
sqr(x)	Square root of x, $x \geq 0$

# Mathematical Functions

## Note:

- `x` and `y` should be declared as `double`.
- In trigonometric and hyperbolic functions. `x` and `y` are in radians.
- All the functions return a `double`.

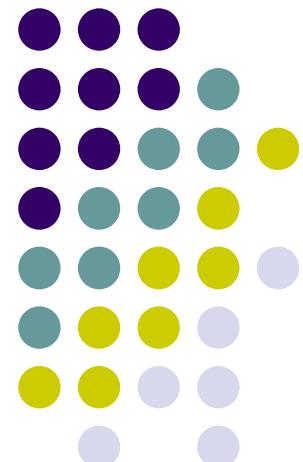


# Introduction to C Programming

---

Chapter 4

Managing Input and Output  
Operations





# Introduction

- **Reading, processing, and writing** of data are the three essential functions of a computer program. Most programs take some data as input and display the processed data, often known as **information** or results, on a suitable medium.
- So far we have seen two methods of providing data to the program variables. One method is to assign values to variables through the assignment statements such as `x =5;` `a = 0;` and so on. Another method is to use the input function **scanf** which can read data from a keyboard.
- For **outputting** results we have used extensively the function **printf** which sends results out to a terminal. All input/output operations are carried out through function calls such as **printf** and **scanf**.



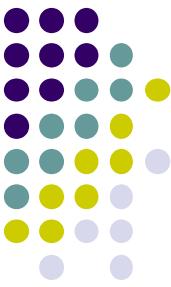
# Introduction

- There exist several functions that have more or less become standard for input and output operations in C. These functions are collectively known as the standard I/O library.
- each program that uses a standard input/output function must contain the statement **# include <stdio.h>** at the beginning.
- The file name **stdio.h** is an abbreviation for **standard input-output** header file. The instruction `#include <stdio.h>` tells the compiler `to search for a file named stdio.h and place its contents at this point in the program'. The contents of the header file become part of the source code when it is compiled.



# Reading a Character

- The simplest of all input/output operations is reading a character from the '**standard input**' unit (usually the keyboard) and writing it to the '**standard output**' unit (usually the screen).
- Reading a single character can be done by using the function **getchar**. (This can also be done with the help of the **scanf** function. The **getchar** takes the following form:  
**variable name getchar( );**
- **variable\_name** is a valid C name that has been declared as **char** type. When this statement is encountered, the computer waits until a key is pressed and then assigns this character as a value to **getchar** function.

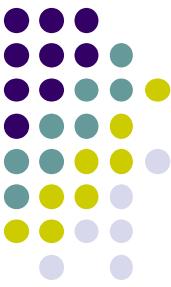


# Reading a Character

- Since **getchar** is used on the right-hand side of an assignment statement, the character value of getchar is in turn assigned to the variable name on the left.
- For example

```
char name;  
name = getchar();
```

- Will assign the character 'H' to the variable name when we press the key H on the keyboard. Since getchar is a function, it requires a set of parentheses as shown.



# Reading a Character

- Since **getchar** is used on the right-hand side of an assignment statement, the character value of getchar is in turn assigned to the variable name on the left.
- For example

```
char name;  
name = getchar();
```

- Will assign the character 'H' to the variable name when we press the key H on the keyboard. Since getchar is a function, it requires a set of parentheses as shown.

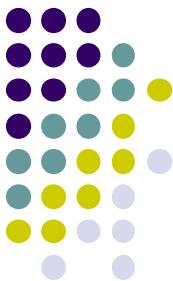


# Reading a Character

- C supports many functions, which are given in Table 4.1. These character functions are contained in the file **ctype.h** and therefore the statement
- `#include<ctype.h>` must be included in the program.

**Table 4.1 Character Test Functions**

<i>Function</i>	<i>Test</i>
<code>isalnum(c)</code>	Is c an alphanumeric character?
<code>isalpha(c)</code>	Is c an alphabetic character?
<code>isdigit(c)</code>	Is c a digit?
<code>islower(c)</code>	Is c lower case letter?
<code>isprint(c)</code>	Is c a printable character?
<code>ispunct(c)</code>	Is c a punctuation mark?
<code>isspace(c)</code>	Is c a white space character?
<code>isupper(c)</code>	Is c an upper case letter?



# Writing a Character

- Like `getchar`, there is an analogous function `putchar` for writing characters one at a time to the terminal. It takes the form as shown below:

```
putchar (variable_name);
```

- where `variable_name` is a type `char` variable containing a character. This statement `displays the character` contained in the `variable_name` at the terminal.
- For example, the statements

```
answer = 'Y';
```

```
putchar (answer);
```

will display the character Y on the screen. The statement

```
putchar (' \n');
```

would cause the cursor on the screen to move to the beginning of the next line.



# Formatted Input

- Formatted input refers to an input data that has been arranged in a particular format. For example, consider the following data:

15.75 123 John

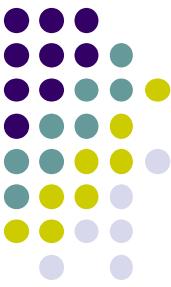
- This line contains three pieces of data, arranged in a particular form. Such data has to be read conforming to the format of its appearance.
- For example, the first part of the data should be read into a variable **float**, the second into **int**, and the third part into **char**. This is possible in C using the **scanf** function.  
**(scanf means scan formatted.)** The general form of **scanf** is:

**scanf("control string", arg1, arg2,... argn);**



# Cont.

- The control string specifies the **field format** in which the data is to be entered and the arguments **arg1, arg2, argn** specify the **address of locations** where the data is stored.
- Control string and arguments are separated by commas. Control string (also known as format string) contains field specifications, which direct the interpretation of input data. It may include:
- *Field (or format) specifications, consisting of the conversion character %, a data type character (or type specifier), and an optional number, specifying the field width.*



# Cont.

- Blanks, tabs, or newlines.
- Blanks, tabs and newlines are ignored. The data type character indicates the type of data that is to be assigned to the variable associated with the corresponding argument. The field width specifier is optional.



# Inputting Integer Numbers

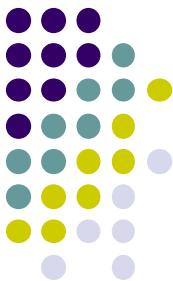
- The field specification for reading an integer number is:  
 $\% w \ sd$
- *The percentage sign (%) indicates that a conversion specification follows. w is an **integer number** that specifies the field width of the number to be read and d, known as **data type character**, indicates that the number to be read is in integer mode.*
- Consider the following example:

`scanf (%2d %5d", &num1, &num2);`

- Data line:

50 31426

- The value 50 is assigned to `num1` and 31426 to `num2`.



# Inputting Integer Numbers

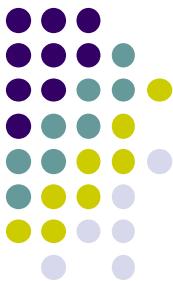
- Suppose the input data is as follows:

31426 50

- The variable num1 will be assigned 31 (because of %2d) and num2 will be assigned 426 (unread part of 31426).
- The value 50 that is unread will be assigned to the first variable in the next scanf' call.
- This kind of errors may be eliminated if we use the field specifications without the field width specifications. That is, the statement

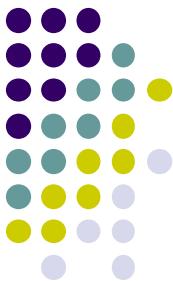
```
scanf ("%d %d", &num1, &num2);
```

- will read the data 31426 50 correctly and assign 31426 to num1 and 50 to num2.



# Inputting Integer Numbers

- Input data items must be separated by spaces, tabs or newlines. Punctuation marks do not count as separators. When the **scanf** function searches the input data line for a value to be read, it will always bypass any white space characters.
- *What happens if we enter a floating point number instead of an integer? The fractional part may be stripped away! Also, scanf may skip reading further input.*
- When the **scanf** reads a particular value, reading of the value will be terminated as soon as the number of characters specified by the field width is reached (if specified) or until a character that is not valid for the value being read is encountered. In the case of integers, valid characters are an optionally signed sequence of digits.



# Inputting Integer Numbers

- An input field may be skipped by specifying \* in the place of field width. For example, the statement :

`scanf("%d %*d %d", &a, &b)`

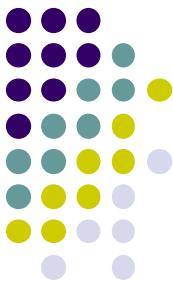
- will assign the data      123 456 789
- as follows

123 to a

456 skipped (because of \*)

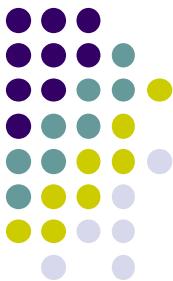
789 to b

- The data type character **d** may be preceded by **'l'** (letter ell) to read long integers and **h** to read short integers.



# Inputting Real Numbers

- Unlike integer numbers, the field width of real numbers is not to be specified and therefore scanf reads real numbers using the simple specification **%f** for both the notations, namely, decimal point notation and exponential notation.
- For example, the statement  
`scanf("%f %f %f", &x, &y, &z);`
- with the input data
- 475.89 43.21E-1 678 will assign the value 475.89 to x, 4.321 to y, and 678.0 to z. The input field specifications may be separated by any arbitrary blank spaces.
- If the number to be read is of **double** type, then the specification should be **%lf** instead of simple **%f**. A number may be skipped using **%\*f** specification.

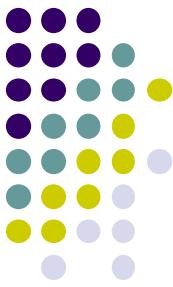


# Inputting Character Strings

- We have already seen how a single character can be read from the terminal using the **getchar** function. The same can be achieved using the **scanf** function also.
- In addition, a **scanf** function can input strings containing more than one character.
- Following are the specifications for reading character strings:

**%ws or %wc**

- The corresponding argument should be a pointer to a character array. However, **%c** may be used to read a single character when the argument is a pointer to a **char** variable.



# Inputting Character Strings

- Some versions of **scanf** support the following conversion specifications for strings:
  - % (characters)
  - % [^characters]
- *The specification %[characters] means that only the characters specified within the brackets are permissible in the input string. If the input string contains any other character, the string will be terminated at the first encounter of such a character.*
- *The specification %[^characters] does exactly the reverse. That is, the characters specified after the circumflex (^) are not permitted in the input string. The reading of the string will be terminated at the encounter of one of these characters.*



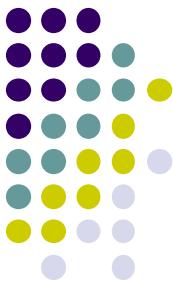
# Reading Mixed Data Types

- It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items **match the control specifications in order and type**. When an attempt is made to read an item that does not match the type expected, the scanf function does not read any further and immediately returns the values read.
- The statement  
**scanf ("%d %c %f %s", &count, &code, &ratio, name);**
- will read the data **15 p 1.575 coffee**
- correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.*



# Reading Mixed Data Types

- It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items **match the control specifications in order and type**. When an attempt is made to read an item that does not match the type expected, the scanf function does not read any further and immediately returns the values read.
- The statement  
`scanf ("%d %c %f %s", &count, &code, &ratio, name);`
- will read the data **15 p 1.575 coffee**
- correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.*



# Reading Mixed Data Types

- It is possible to use one **scanf** statement to input a data line containing mixed mode data. In such cases, care should be exercised to ensure that the input data items **match the control specifications in order and type**. When an attempt is made to read an item that does not match the type expected, the scanf function does not read any further and immediately returns the values read.
- The statement  
**scanf ("%d %c %f %s", &count, &code, &ratio, name);**
- will read the data **15 p 1.575 coffee**
- correctly and assign the values to the variables in the order in which they appear. Some systems accept integers in the place of real numbers and vice versa, and the input data is converted to the type specified in the control string.*



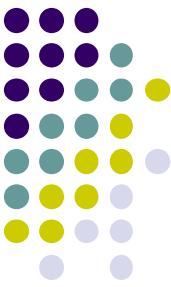
# Reading Mixed Data Types

**Table 4.2 Commonly used scanf Format Codes**

<b>Code</b>	<b>Meaning</b>
%c	read a single character
%d	read a decimal integer
%e	read a floating point value
%f	read a floating point value
%g	read a floating point value
%h	read a short integer
%i	read a decimal, hexadecimal or octal integer
%o	read an octal integer
%s	read a string
%u	read an unsigned decimal integer
%x	read a hexadecimal integer
%[..]	read a string of word(s)

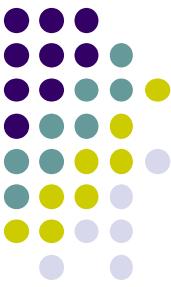
The following letters may be used as prefix for certain conversion characters.

- h for short integers
- l for long integers or double
- L for long double



# Rules for Scanf

- Each variable to be read must have a filed specification.
- For each field specification, there must be a variable address of proper type.
- Any non-whitespace character used in the format string must have a matching character in the user input.
- Never end the format string with whitespace. It is a fatal error!
- The **scanf** reads until:
  - A whitespace character is found in a numeric specification, or
  - The maximum number of characters have been read or An error is detected, or
  - The end of file is reached

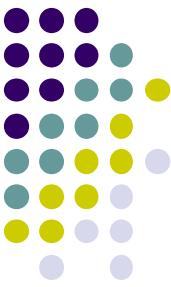


# Formatted Output

- The `printf` statement provides certain features that can be effectively exploited to control the alignment and spacing of print-outs on the terminals.
- The general form of `printf` statement is:  
`printf ("control string", arg1, arg2,... argn);`

Control string consists of three types of items:

- *Characters that will be printed on the screen as they appear.*
- *Format specifications that define the output format for display of each item.*
- *Escape sequence characters such as \n, \t, and b.*



# Formatted Output

- The control string indicates how many arguments follow and what their types are. The arguments arg1, arg2, argn are the variables whose values are formatted and printed according to the specifications of the control string. **The arguments should match in number, order and type with the format specifications.** A simple format specification has the following form:

% w.p type-specifier

- where **w** is an integer number that specifies the total number of columns for the output value and **p** is another integer number that specifies the number of digits to the right of the decimal point (of a real number) or the number of characters to be printed from a string.

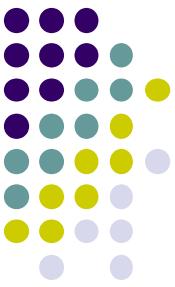


# Formatted Output

- Both w and p are optional. Some examples of formatted printf statement are:

```
printf("Programming in C");
printf(" ");
printf("\n");
printf("%d", x);
printf("a = %f\n b = %f", a, b);
printf("sum = %d", 1234);
printf("\n\n");
```

- printf never supplies a **newline** automatically and therefore multiple printf statements may be used to build one line of output. A **newline** can be introduced by the help of a newline character '\n' as shown in some of the examples above.*



# Output of integer numbers

- The format specification for printing an integer number is:

%w d

- where **w** specifies the **minimum field width** for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification.
- **d** specifies that the value to be printed is an integer. The number is written right-justified in the given field width. Leading blanks will appear as necessary.
- The following examples illustrate the output of the number 9876 under different formats:



# Output of integer numbers

## Format

```
printf("%d", 9876)
```

## Output

9	8	7	6
---	---	---	---

```
printf("%6d", 9876)
```

		9	8	7	6
--	--	---	---	---	---

```
printf("%2d", 9876)
```

9	8	7	6
---	---	---	---

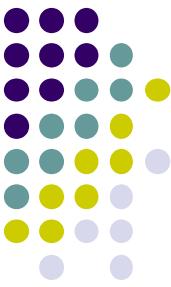
```
printf("%-6d", 9876)
```

9	8	7	6		
---	---	---	---	--	--

```
printf("%06d", 9876)
```

0	0	9	8	7	6
---	---	---	---	---	---

- It is possible to force the printing to be left-justified by placing a minus sign directly after the % character, as shown in the fourth example above. It is also possible to pad with zeros the leading blanks by placing a 0 (zero) before the field width specifier as shown in the last item above. The minus (—) and zero (0) are known as flags. Long integers may be printed by specifying *l*d in the place of d in the format specification. Similarly, we may use *h*d for printing short integers.

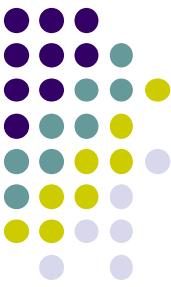


# Output of Real numbers

- The output of a real number may be displayed in decimal notation using the following format specification:

*%w.p f*

- where **w** specifies the **minimum field width** for the output. However, if a number is greater than the specified field width, it will be printed in full, overriding the minimum specification.
- **d** specifies that the value to be printed is an integer. The number is written right-justified in the given field width. Leading blanks will appear as necessary.
- The following examples illustrate the output of the number 9876 under different formats:



# Output of Real numbers

- The integer **w** indicates the minimum number of positions that are to be used for the display of the value and the integer **p** indicates the number of digits to be displayed after the decimal point (precision).
- *The value, when displayed, is rounded to p decimal places and printed right-justified in the field of w columns.*
- Leading blanks and trailing zeros will appear as necessary. The default precision is 6 decimal places. The negative numbers will be printed with the minus sign. The number will be displayed in the form  
**[-] mmm-nnn**
- We can also display a real number in exponential notation by using the specification:

**% w.p e**

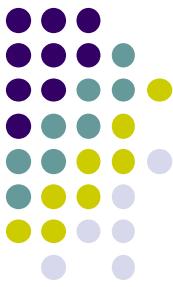


# Output of Real numbers

- The display takes the form
- [ - ] m.nnnn [ $\pm$ ]xx
- where the length of the string of n's is specified by the precision **p**. The default precision is 6.
- The field width **w** should satisfy the condition.

$$W \geq p+7$$

- The value will be rounded off and printed right justified in the field of w columns. Padding the leading blanks with zeros and printing with left-justification are also possible by using flags 0 or - before the field width specifier w.
- The following examples illustrate the output of the number **y = 98.7654** under different format specifications:



# Output of Real numbers

- The display takes the form
- [ - ] m.nnnn [ $\pm$ ]xx
- where the length of the string of n's is specified by the precision **p**. The default precision is 6.
- The field width **w** should satisfy the condition.

$$W \geq p+7$$

- The value will be rounded off and printed right justified in the field of w columns. Padding the leading blanks with zeros and printing with left-justification are also possible by using flags 0 or - before the field width specifier w.
- The following examples illustrate the output of the number **y = 98.7654** under different format specifications:



# Output of Real numbers

## *Format*

```
printf("%7.4f",y)
```

```
printf("%7.2f",y)
```

```
printf("%-7.2f",y)
```

```
printf("%f",y)
```

```
printf("%10.2e",y)
```

```
printf("%11.4e",-y)
```

```
printf("%-10.2e",y)
```

```
printf("%e",y)
```

## *Output*

9	8	.	7	6	5	4
---	---	---	---	---	---	---

		9	8	.	7	7
--	--	---	---	---	---	---

9	8	.	7	7		
---	---	---	---	---	--	--

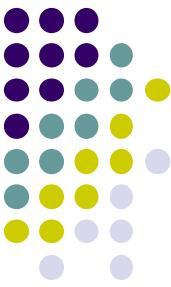
9	8	.	7	6	5	4
---	---	---	---	---	---	---

		9	.	8	8	e	+	0	1
--	--	---	---	---	---	---	---	---	---

-	9	.	8	7	6	5	e	+	0	1
---	---	---	---	---	---	---	---	---	---	---

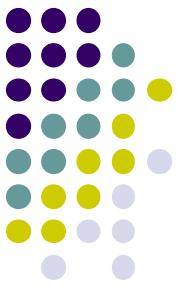
9	.	8	8	e	+	0	1		
---	---	---	---	---	---	---	---	--	--

9	.	8	7	6	5	4	0	e	+	0	1
---	---	---	---	---	---	---	---	---	---	---	---



# Output of Real numbers

- Some systems also support a special field specification character that lets the user define the field size at run time. This takes the following form:  
`printf("%.*f", width, precision, number);`
- In this case, both the **field width** and the **precision** are given as arguments which will supply the values for w and p. For example,  
`printf("%.*f", 7, 2, number);`
- is equivalent to  
`printf ("%7.2f", number);`
- The advantage of this format is that the values for width and precision may be supplied at run time, thus making the format a dynamic one.



# Output of Real numbers

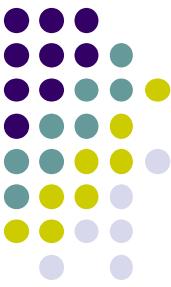
- For example, the above statement can be used as follows:

```
int width = 7;  
int precision = 2;  
printf("%.*P," width, precision, number);
```



# Printing of a Single Character

- A single character can be displayed in a desired position using the format:  
`%wc`
- The character will be displayed right-justified in the field of **w** columns. We can make the display **left-justified** by placing a **minus sign** before the integer **w**. The default value for **w** is 1.
- Printing of Strings The format specification for outputting strings is similar to that of real numbers. It is of the form
- `%w.ps` where **w** specifies the field width for display and **p** instructs that only the first **p** characters of the string are to be displayed. The display is right-justified. The following examples show the effect of variety of specifications in printing a string "NEW DELHI 110001", containing 16 characters (including blanks).



# Printing of Strings

- The format specification for outputting strings is similar to that of real numbers. It is of the form

*%w.ps*

- where **w** specifies the **field width** for display and **p** instructs that only the first **p** characters of the string are to be displayed.
- The display is **right-justified**. The following examples show the effect of variety of specifications in printing a string "**NEW DELHI 110001**", containing 16 characters (including blanks).



# Printing of Strings

## *Specification*

## *Output*

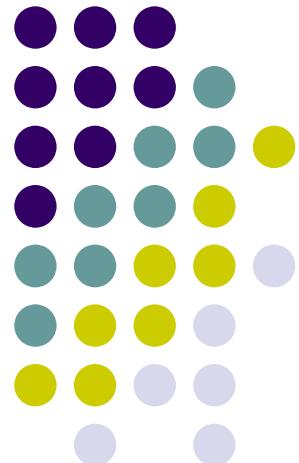
	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
%s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				
%20s					N	E	W	.	D	E	L	H	I		1	1	0	0	0	1
%20.10s											N	E	W		D	E	L	H	I	
.%5s	N	E	W		D															
%-20.10s	N	E	W		D	E	L	H	I											
%5s	N	E	W		D	E	L	H	I		1	1	0	0	0	1				

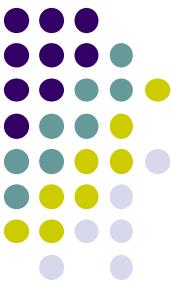
# Introduction to C Programming

---

## Chapter 5

### Decision Making and Branching





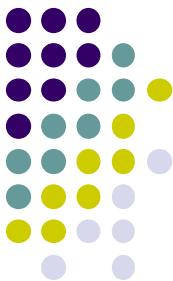
# Introduction

- We have seen that a C program is a set of statements which are normally executed sequentially in the order in which they appear. This happens when no options or no repetitions of certain calculations are necessary.
- However, in practice, we have a number of situations where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met.
- *This involves a kind of decision making to see whether a particular condition has occurred or not and then direct the computer to execute certain statements accordingly.*



# Introduction

- C language possesses such decision-making capabilities by supporting the following statements:
  - if statement
  - switch statement
  - Conditional operator statement
  - goto statement
- These statements are popularly known as **decision-making statements**. Since these statements 'control' the flow of execution, they are also known as **control statements**.



# Decision making with if statement

- The **if** statement is a powerful decision-making statement and is used to **control the flow of execution** of statements. It is basically a two-way decision statement and is used in **conjunction with an expression**. It takes the following form:

**if (test expression)**

- It allows the computer to evaluate the expression first and then, depending on whether the value of the expression (relation or condition) is '**true**' (or non-zero) or '**false**' (zero), it transfers the control to a particular statement.
- This point of program has two paths to follow, one for the true condition and the other for the false condition as shown in Fig. 5.1.



# Cont.

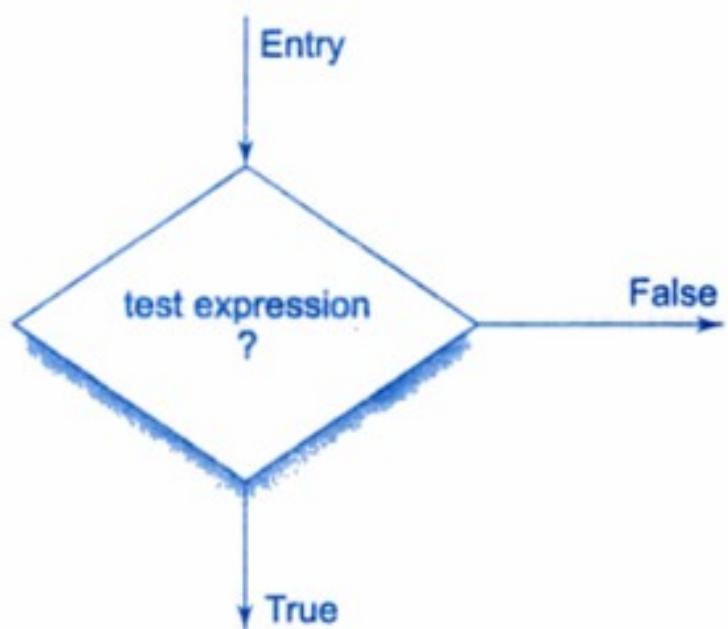
- Some examples of decision making, using **if** statements are:

**if** (bank balance is zero)  
    borrow money

**if** (room is dark)  
    put on lights

**if** (code is 1)  
    person is male

**if** (age is more than 55)  
    person is retired

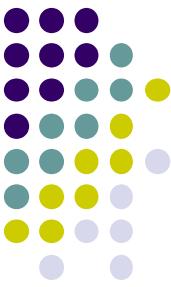


**Fig. 5.1 Two-way branching**



# Introduction

- The if statement may be implemented in different forms depending on the complexity of conditions to be tested.
- The different forms are:
  1. Simple **if** statement
  2. **if else** statement
  3. Nested **if....else** statement
  4. **else if ladder**.



# Simple IF Statement

- The general form of a simple **if** statement is

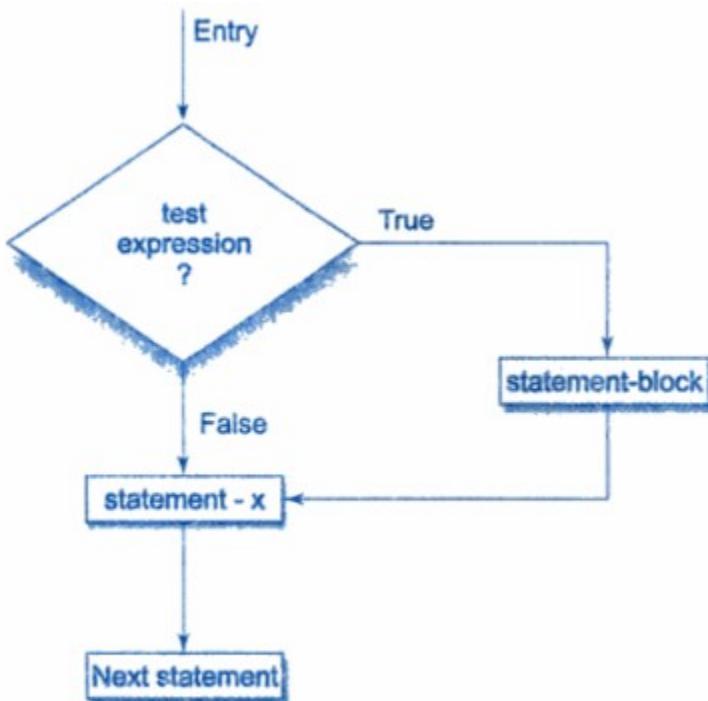
```
if (test expression)
{
    statement-block;
}
statement-x;
```

- The '**statement-block**' may be a single statement or a group of statements. If the *test expression is true*, the *statement-block* will be executed; otherwise the *statement-block* will be skipped and the execution will jump to the *statement-x*.



# Cont.

- Remember, when the condition is true both the statement block and the statement-x are executed in sequence. This is illustrated in Fig 5.2



**Fig. 5.2 Flowchart of simple if control**



# Cont.

- Consider the following segment of a program that is written for processing of marks obtained in an entrance examination.

```
if (category == SPORTS)
{
    marks = marks + bonus marks;
}
printf("%f", marks);
```

- The program tests the type of category of the student. If the student belongs to the **SPORTS** category, then additional **bonus\_marks** are added to his marks before they are printed. For others, **bonus\_marks** are not added.



# The IF...Else Statement

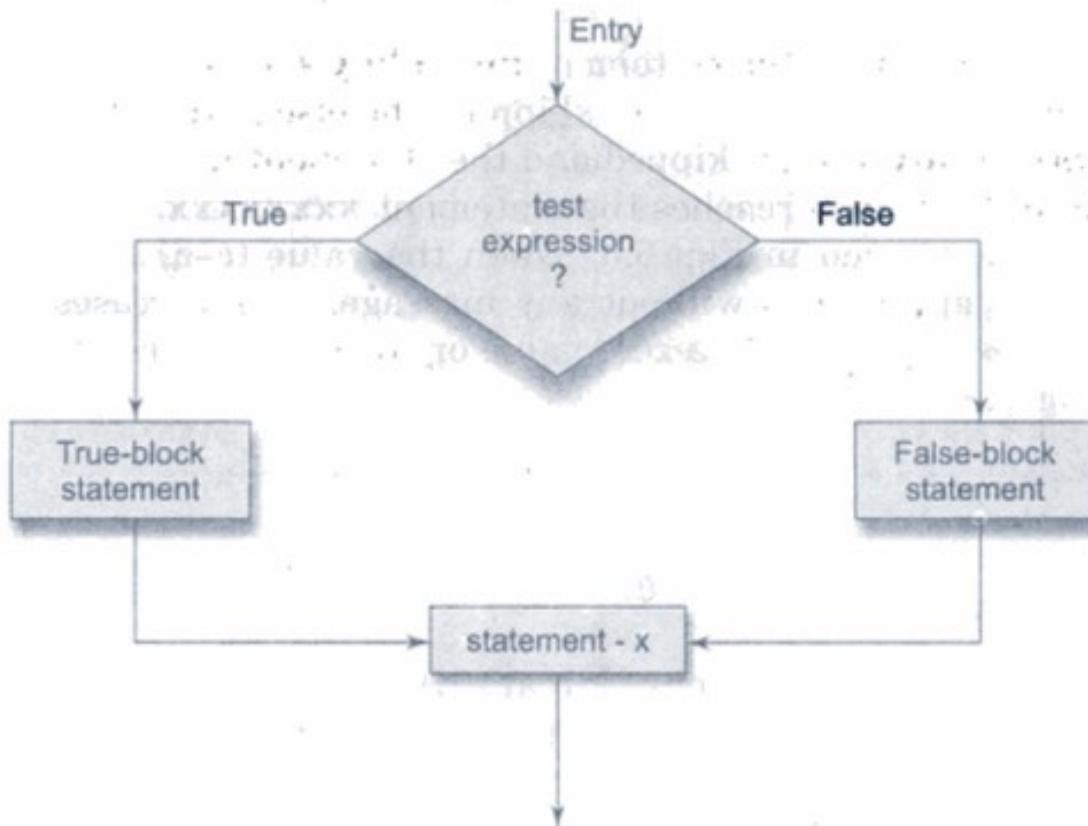
- The **if...else** statement is an extension of the simple if statement. The general form is

```
If (test expression)
{
    True-block statement(s)
}
else {
    false-block statement(s)
}
statement-x
```

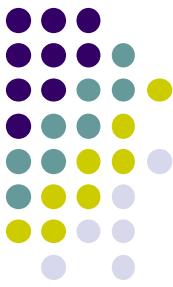
- *If the test expression is true, then the true-block statement(s), immediately following the if statements are executed; otherwise, the false-block statement(s) are executed. In either case, either true-block or false-block will be executed, not both. This is illustrated in Fig. 5.5. In both the cases, the control is transferred subsequently to the statement-x.*



# The IF...Else Statement



**Fig. 5.5 Flowchart of if.....else control**

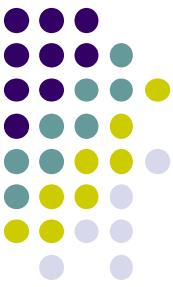


# The IF....Else Statement

- Let us consider an example of counting the number of boys and girls in a class. We use code 1 for a boy and 2 for a girl. The program statement to do this may be written as follows:

```
if(code ==1)  
    boy = boy + 1;  
  
if(code == 2)  
    girl = girl+1;
```

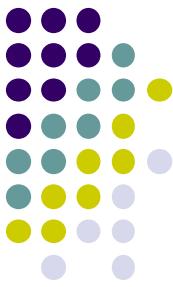
- The first test** determines whether or not the student is a boy. If yes, the number of boys is increased by 1 and the program continues to the second test. **The second test** again determines whether the student is a girl. This is unnecessary. Once a student is identified as a boy, there is no need to test again for a girl. A student can be either a boy or a girl, not both



# The IF...Else Statement

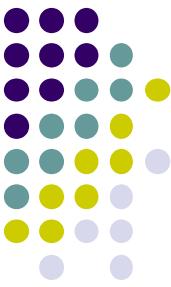
- The above program segment can be modified using the else clause as follows:

```
if (code == 1)  
    boy = boy + 1;  
  
else  
    girl = girl + 1;  
  
XXXXXXXXXXXX
```



# The IF....Else Statement

- Here, if the code is equal to 1, the statement  
`boy = boy + 1;` is executed and the control is transferred to the statement **xxxxxx**, after skipping the else part.
- If the code is not equal to 1, the statement `boy = boy + 1;` is skipped and the statement in the else part `girl = girl + 1;` is executed before the control reaches the statement **xxxxxx**.



# Cont.

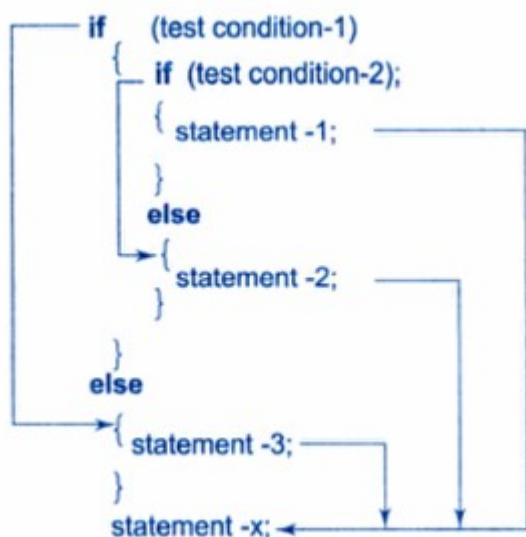
- Consider the following program
- When the value ( $c-d$ ) is zero, the ratio is not calculated and the program stops without any message.
- In such cases we may not know whether the program stopped due to a zero value or some other error. This program can be improved by adding the else clause as follows:

```
if (c-d != 0)
{
    ratio = (float)(a+b)/(float)(c-d);
    printf("Ratio = %f\n", ratio);
}
else
    printf("c-d is zero\n");
```

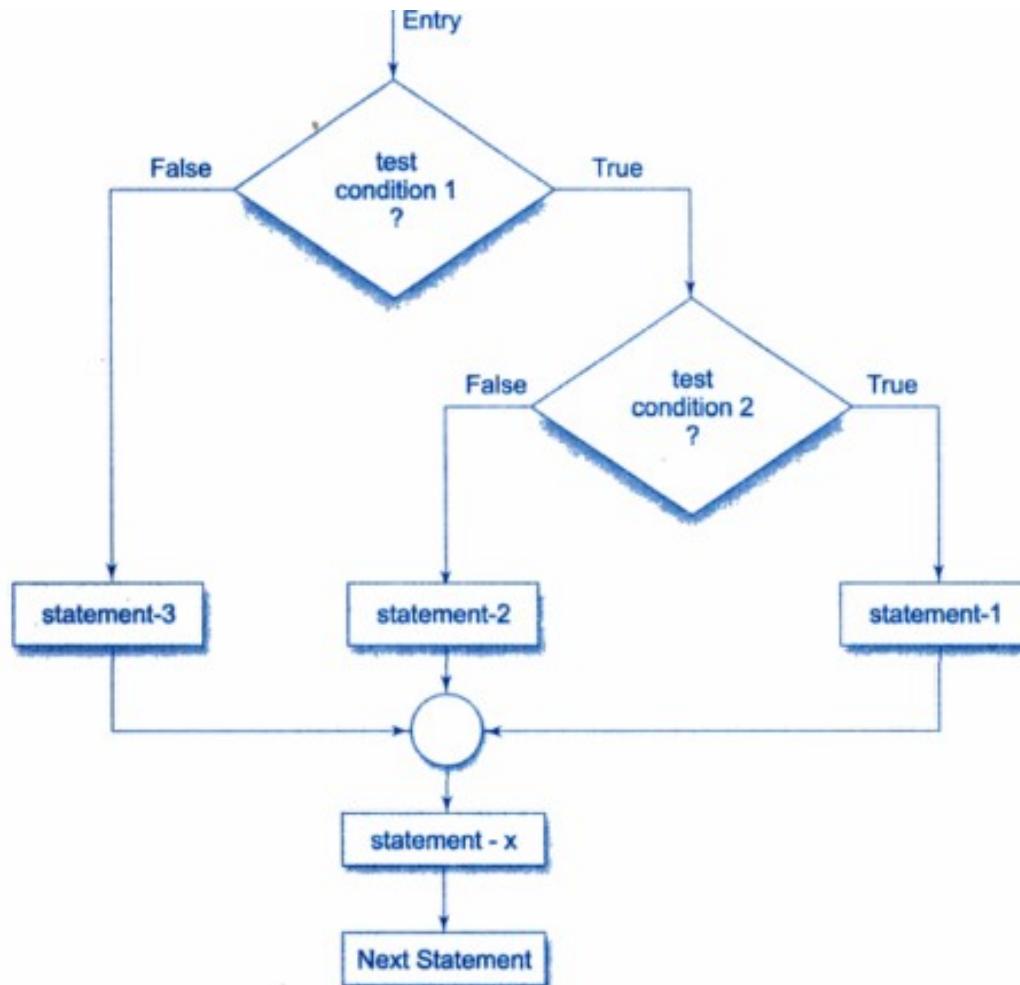


# Cont.

- When a series of decisions are involved, we may have to use more than one **if...else** statement in *nested* form as shown below:
- The logic of execution is illustrated in Fig. 5.7. If the *condition-1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *condition-2* is true, the statement-1 will be evaluated; otherwise the statement-2 will be evaluated and then the control is transferred to the statement-x



# Cont.



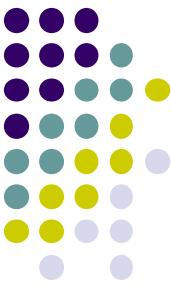
**Fig. 5.7 Flow chart of nested if...else statements**



# Cont.

- A commercial bank has introduced an incentive policy of giving bonus to all its deposit holders. The policy is as follows: A bonus of 2 percent of the balance held on 31<sup>st</sup> December is given to every one, irrespective of their balance, and 5 percent is given to female account holders if their balance is more than Rs. 5000. This logic can be coded as follows:

```
.....
if (sex is female)
{
    if (balance > 5000)
        bonus = 0.05 * balance;
    else
        bonus = 0.02 * balance;
}
else
{
    bonus = 0.02 * balance;
}
balance = balance + bonus;
.....
.....
```

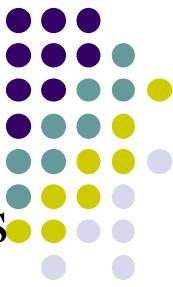


# Cont.

- When nesting, care should be exercised to match every if with an else. Consider the following alternative to the above program (which looks right at the first sight):

```
if (sex is female)
    if (balance > 5000)
        bonus = 0.05 * balance;
else
    bonus = 0.02 * balance;
balance = balance + bonus;
```

- There is an ambiguity as to over which if the else belongs to. In C, an else is linked to the closest non-terminated if.



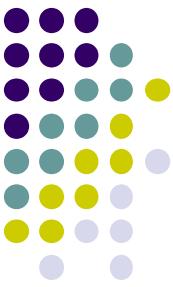
# Cont.

- Therefore, the **else** is associated with the inner **if** and there is no **else** option for the outer **if**.
- This means that the computer is trying to execute the statement

balance = balance + bonus;

- without really calculating the bonus for the male account holders. Consider another alternative, which also looks correct:

```
if (sex is female)
{
    if (balance > 5000)
        bonus = 0.05 * balance;
    }
else
    bonus = 0.02 * balance;
balance = balance + bonus;
```



# Cont.

- In this case, **else** is associated with the outer **if** and therefore bonus is calculated for the male account holders. However, bonus for the female account holders, whose balance is equal to or less than 5000 is not calculated because of the missing **else** option for the inner **if**.



# The ELSE IF Ladder

- The general form of **ELSE IF Ladder**

```
if ( condition 1)  
    statement-1;
```

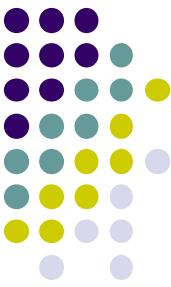
```
else if ( condition 2)  
    statement-2;
```

```
else if ( condition 3)  
    statement-3;
```

```
else if ( condition n)  
    statement-n;
```

```
else  
    default-statement;
```

```
statement-x;
```



# Cont.

- This construct is known as the **else if ladder**. The conditions are evaluated from the **top (of the ladder), downwards**.
- As soon as a true condition is found, the statement associated with it is executed and the control is transferred to the statement-x (skipping the rest of the ladder).
- When all the n conditions become false, then the final else containing the default-statement will be executed..



# Cont.

Average marks	Grade
80 to 100	Honours
60 to 79	First Division
50 to 59	Second Division
40 to 49	Third Division
0 to 39	Fail

This grading can be done using the **else if** ladder as follows:

```
if (marks > 79)
    grade = "Honours";
else if (marks > 59)
    grade = "First Division";
else if (marks > 49)
    grade = "Second Division";
else if (marks > 39)
    grade = "Third Division";
else
    grade = "Fail";
printf ("%s\n", grade);
```



# Cont.

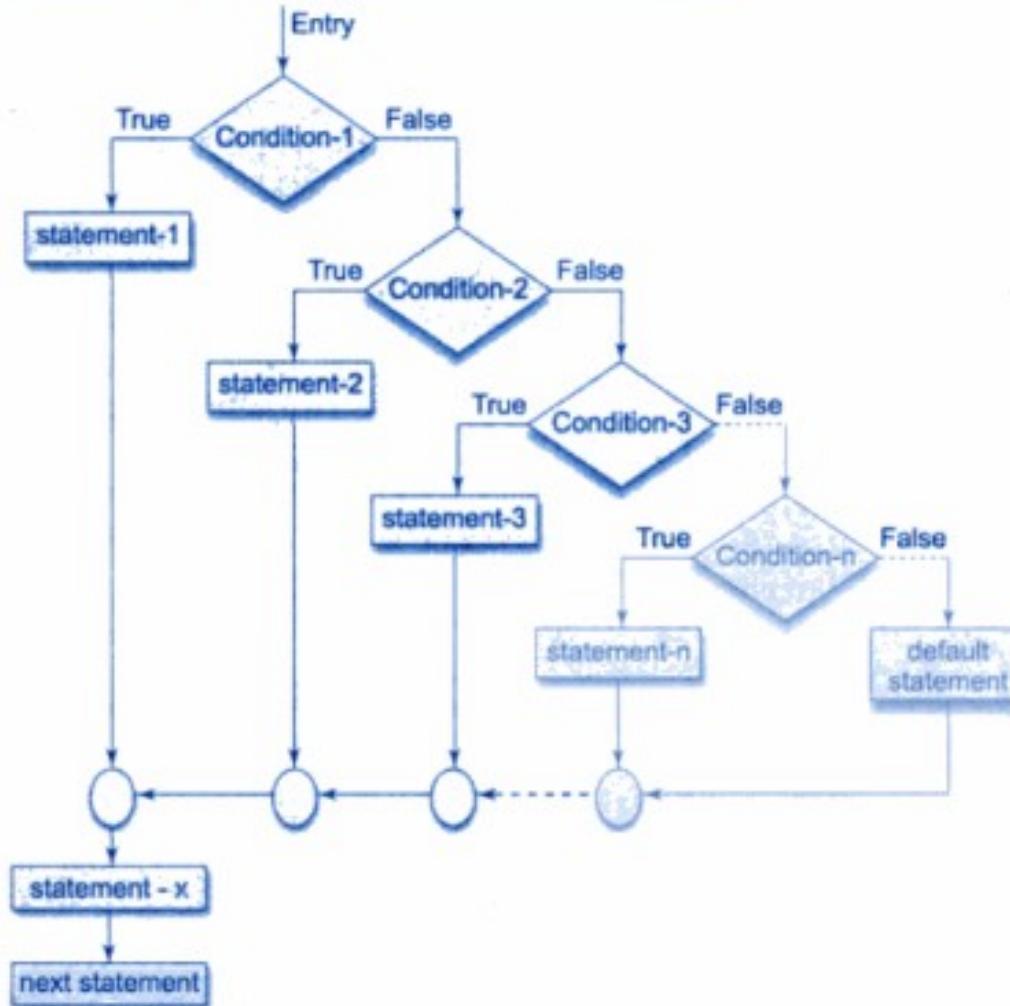
Consider another example given below:

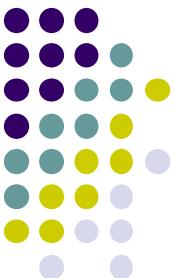
```
-----
-----
if (code == 1)
    colour = "RED";
else if (code == 2)
    colour = "GREEN";
else if (code == 3)
    colour = "WHITE";
else
    colour = "YELLOW";
-----
-----
```



# Cont.

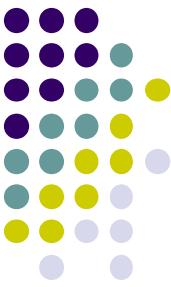
Code numbers other than 1, 2 or 3 are considered to represent YELLOW colour. The same results can be obtained by using nested if...else statements.





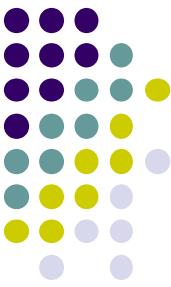
# Cont.

```
if (code != 1)
    if (code != 2)
        if (code != 3)
            colour = "YELLOW";
        else
            colour = "WHITE";
    else
        colour = "GREEN";
else
    colour = "RED";
```



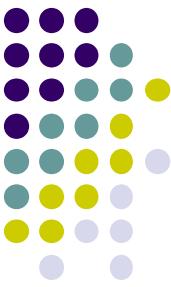
# Rules for Indentation

- When using control structures, a statement often controls many other statements that follow it. In such situations it is a good practice to use indentation to show that the indented statements are dependent on the preceding controlling statement.
- Indent statements that are dependent on the previous statements; provide at least three spaces of indentation.
- Align vertically else clause with their matching if clause.
- Use braces on separate lines to identify a block of statements.
- Indent the statements in the block by at least three spaces to the right of the braces.



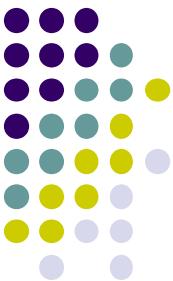
# Rules for Indentation

- Align the opening and closing braces.
- Use appropriate comments to signify the beginning and end of blocks.
- Indent the nested statements as per the above rules.
- Code only one clause or statement on each line.



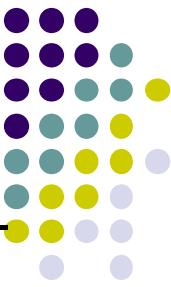
# Switch Statement

- We have seen that when one of the many alternatives is to be selected, we can use an if statement to control the selection. However, the complexity of such a program increases dramatically when the number of alternatives increases.
- The program becomes difficult to read and follow. At times, it may confuse even the person who designed it. Fortunately, C has a built-in multiway decision statement known as a switch.
- The switch statement tests the value of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form of the switch statement is as shown below:



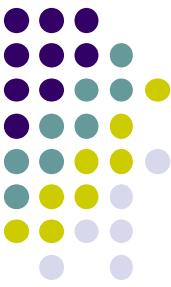
# Switch Statement

```
switch (expression)
{
    case value-1:
        block-1
        break;
    case value-2:
        block-2
        break;
    . . .
    . . .
    default:
        default-block
        break;
}
statement-x;
```



# Switch Statement

- The expression is an integer expression or characters. Value-I, value-2 ... are constants or constant expressions (evaluable to an integral constant) and are known as case labels. Each of these values should be unique within a switch statement. block-1, block-2 .... are statement lists and may contain zero or more statements.
- There is no need to put braces around these blocks. Note that case labels end with a colon (:). When the switch is executed, the value of the expression is successfully compared against the values value-I, value-2,... If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.



# Switch Statement

- The break statement at the end of each block signals the end of a particular case and causes an exit from the switch statement, transferring the control to the statement-i following the switch. The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action takes place if all matches fail and the control goes to the statement-x. (ANSI C permits the use of as many as 257 case labels).



# Switch Statement

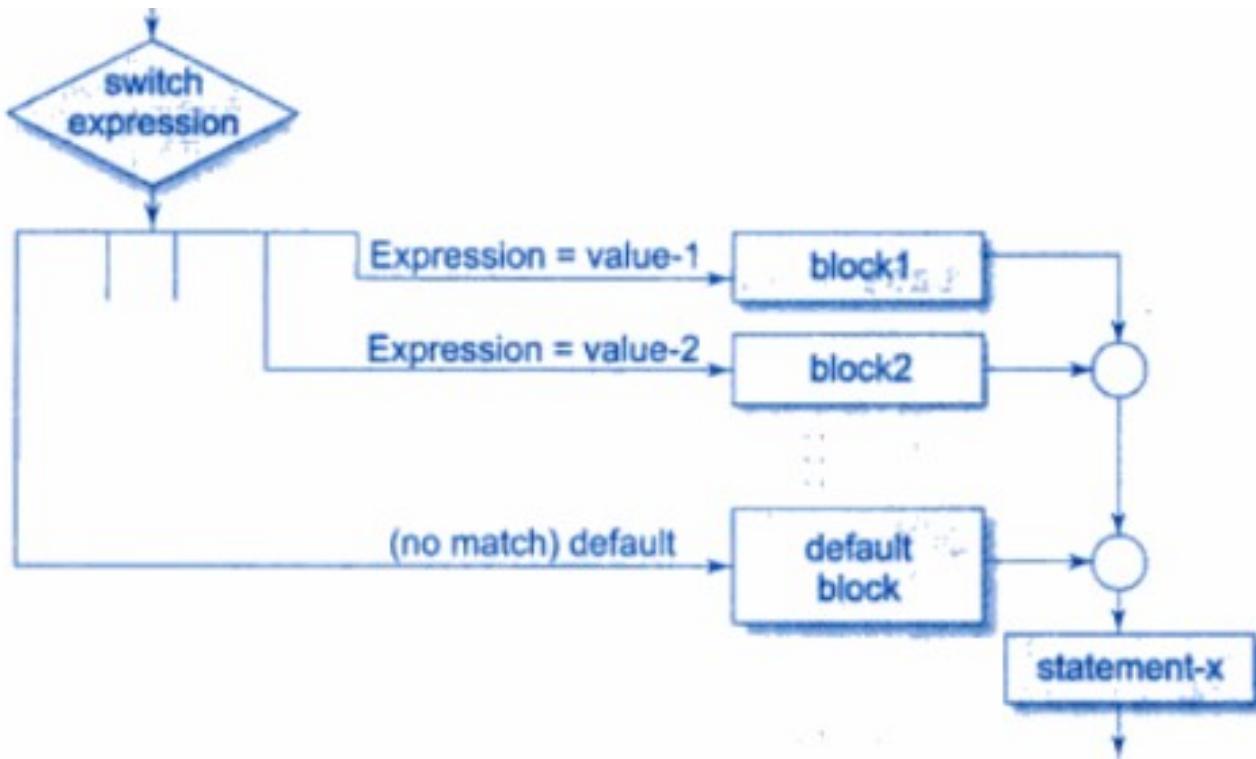
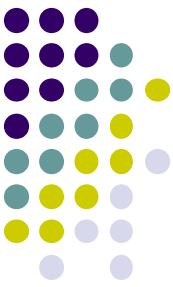
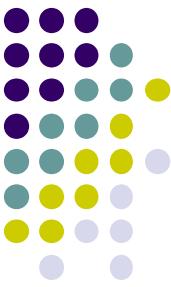


Fig. 5.11 Selection process of the switch statement



# Switch Statement

```
---  
---  
index = marks/10  
switch (index)  
{  
    case 10:  
    case 9:  
    case 8:  
        grade = "Honours";  
        break;  
    case 7:  
    case 6:  
        grade = "First Division";  
        break;  
    case 5:  
        grade = "Second Division";  
        break;  
    case 4:  
        grade = "Third Division";  
        break;  
    default:  
        grade = "Fail";  
        break;  
}  
printf("%s\n", grade);
```



# Rules for switch statement

- The switch expression must be an integral type.
- Case labels must be constants or constant expressions.
- Case labels must be unique. No two labels can have the same value.
- Case labels must end with semicolon.
- The break statement transfers the control out of the switch statement.
- The break statement is optional. That is, two or more case labels may belong to the same statements.



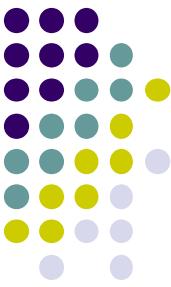
# Rules for switch statement

- The default label is optional. If present, it will be executed when the expression does not find a matching case label.
- There can be at most one default label.
  - The default may be placed anywhere but usually placed at the end.
  - It is permitted to nest switch statements.



# The ?: Operator

- The C language has an unusual operator, useful for making two-way decisions. This operator is a combination of ? and :, and takes three operands. This operator is popularly known as the conditional operator.
- The general form of use of the conditional operator is as follows:
- `conditional expression ? expression1 : expression2 !`
- The conditional expression is evaluated first. If the result is nonzero, expression) is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned.



# The ?: Operator

- For example, the segment

```
if (x < 0)
```

```
    Flag = 0;
```

```
else
```

```
    flag = 1;
```

can be written as

```
Flag = ( x < 0 ) ? 0 : 1;
```

- For example, consider the weekly salary of a sales girl who is selling some domestic products. If  $x$  is the number of products sold in a week, her weekly salary is given by



# The ?: Operator

$$\text{salary} = \begin{cases} 4x + 100 & \text{for } x < 40 \\ 300 & \text{for } x = 40 \\ 4.5x + 150 & \text{for } x > 40 \end{cases}$$

This complex equation can be written as salary •

$y = (x \neq 40) ? ((x < 40) ? (4*x+100) : (4.5*x+150)) : 300;$

The same can be evaluated using if...else statements as follows:

```
if (x <= 40)
    if (x < 40)
        salary = 4 * x+100;
    else
        salary = 300;
else
    salary = 4.5 * x+150;
```



# The ?: Operator

- When the conditional operator is used, the code becomes more concise and perhaps, more efficient. However, the readability is poor. It is better to use if statements when more than a single nesting of conditional operator is required.



# The goto statement

- C supports the **goto** statement to branch unconditionally from one point to another in the program. The **goto** requires a label in order to identify the place where the branch is to be made.
- A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred. The general forms of **goto** and label statements are shown below:

goto label;

-----

-----

-----

-----

label: ←  
statement;

label: ←  
statement;

-----

-----

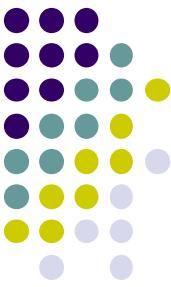
-----

-----

goto label; -----

Forward jump

Backward jump



# The goto statement

- The *label:* can be anywhere in the program either before or after the **goto** label; statement. During running of a program when a statement like  
**`goto begin;`**
- is met, the flow of control will jump to the statement immediately following the label **begin:**.
- This happens unconditionally.
- Note that a **goto** breaks the normal sequential execution of the program. If the *label:* is before the statement `goto label;` a *loop* will be formed and some statements will be executed repeatedly. Such a jump is known as a ***backward jump***.



# The goto statement

- On the other hand, if the label: is placed after the **goto** *label*; some statements will be skipped and the jump is known as a *forward jump*.
- A **goto** is often used at the end of a program to direct the control to go to the input statement, to read further data. Consider the following example:

```
main()
{
    double x, y;
    read:
    scanf("%f", &x);
    if (x < 0) goto read;
    y = sqrt(x);
    printf("%f %f\n", x, y);
    goto read;
}
```



# The goto statement

- This program is written to evaluate the square root of a series of numbers read from the terminal. The program uses two **goto** statements, one at the end, after printing the results to transfer the control back to the input statement and the other to skip any further computation when the number is negative.
- Due to the unconditional **goto** statement at the end, the control is always transferred back to the input statement. In fact, this program puts the computer in a permanent loop known as an infinite loop. The computer goes round and round until we take some special steps to terminate the loop. Such infinite loops should be avoided.

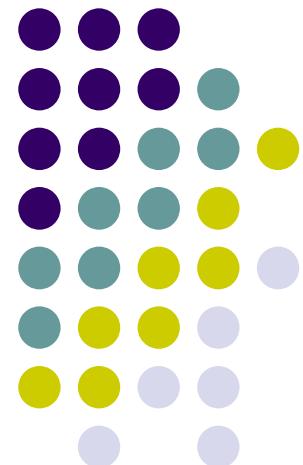


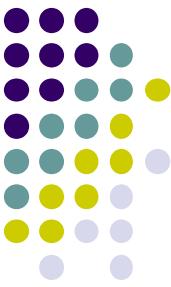
# Introduction to C Programming

---

Chapter 7

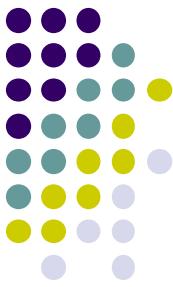
Arrays





# Introduction

- ▶ So far we have used only the fundamental data types, namely char, int, float double and variations of int and double.
- ▶ Although these types are very useful, they are constrained by the fact that a variable of these types can store only one value at any given time.
- ▶ Therefore, they can be used only to handle limited amounts of data.
- ▶ In many applications. however, we need to handle a large volume of data in terms of reading, processing and printing.



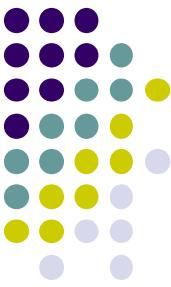
# Introduction

- To process such large amounts of data, we need a powerful data type that would facilitate efficient storing, accessing and manipulation of data items.
  - C supports a derived data type known as array that can be used for such applications.
- 
- *An array is a fixed-size sequenced collection of elements of the same data type. It is simply a grouping of like-type data.*
  - In its simplest form, an array can be used to represent a list of numbers, or a list of names.



# Introduction

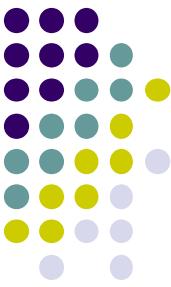
- Some examples where the concept of an array can be used:
  - List of temperatures recorded every hour in a day, or a month. or a year.
  - List of employees in an organization.
  - List of products and their cost sold by a store.
  - Test scores of a class of students.
  - List of customers and their telephone numbers.
  - Table of daily rainfall data. and so on.
- *since an array Provides a convenient structure for representing data, it is classified as one of the data structures in C.*



# Introduction

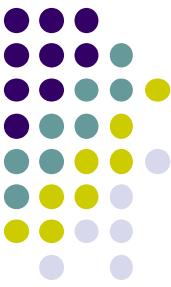
- ▶ As we mentioned earlier, an array is a sequenced collection of related data items that share a common name.
- ▶ For instance, we can use an array name **salary** to represent a set of salaries of a **group of employees** in an organization.
- ▶ We can refer to the individual salaries by writing a number called **index or subscript in brackets** after the array name.  
For example.

- ▶ represents the sa **salary [10]**



# Introduction

- While the complete set of values is referred to as an **array**, individual values are called **elements**.
- The ability to use a single name to represent a collection of items and to refer to an item by specifying the item number enables us to develop concise and efficient programs.
- *For example, we can use a loop construct, discussed earlier, with the subscript as the control variable to read the entire array, perform calculations; and print out the results. We can use arrays to represent not only simple lists of values but also tables of data in two, three or more dimensions.*

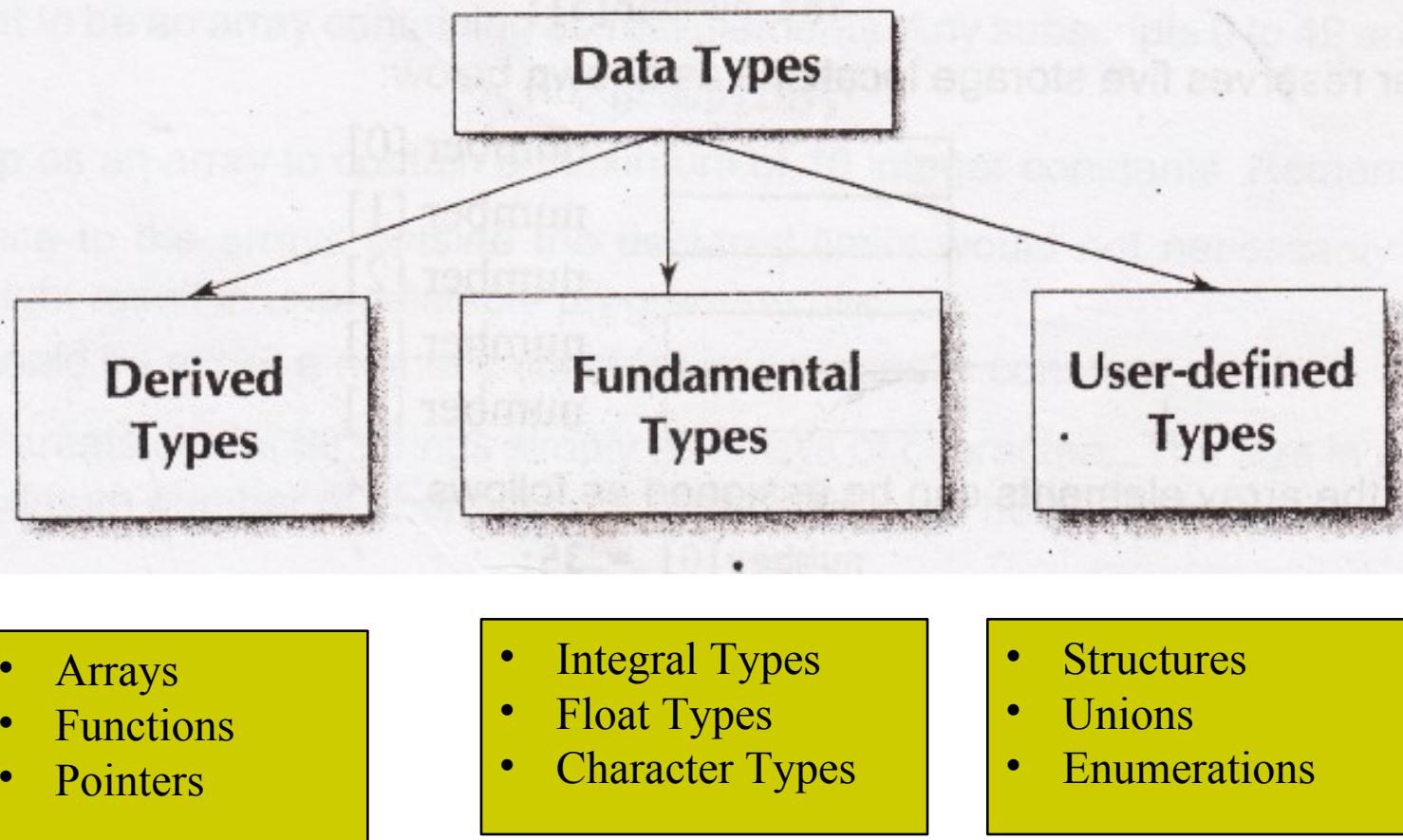


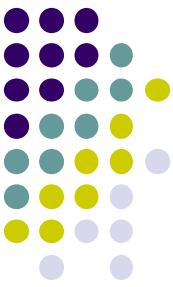
# Introduction

- In this chapter, we introduce the concept of an array and discuss how to use it to create and apply the following types of arrays.
  - One-dimensional arrays
  - Two-dimensional arrays



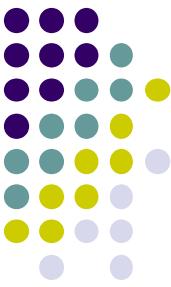
# Data Structure





# Data Structure

- Arrays and structures are referred to as **structured data types** because they can be used to represent data values that have a structure of some sort.
- *Structured data types provide an organizational scheme that shows the relationships among the individual elements and facilitate efficient data manipulations.*
- In programming parlance, such data types are known as **data structures**.



# One Dimensional Array

- A list of items can be given one variable name using only one subscript and such a variable is called a **single-subscripted variable** or a **one-dimensional array**. In mathematics, we often deal with variables that are single-subscripted. For instance, we use the equation.

$$A = \frac{\sum_{i=1}^n x_i}{n}$$

- to calculate the average of n values of x. The subscripted variable  $x_i$ , refers to the ith element of x.



# One Dimensional Array

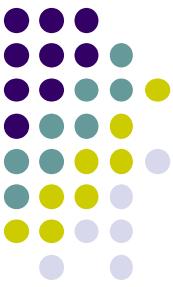
- In C, single-subscripted variable x, can be expressed as

x[1], x[2], x[3], x[n]

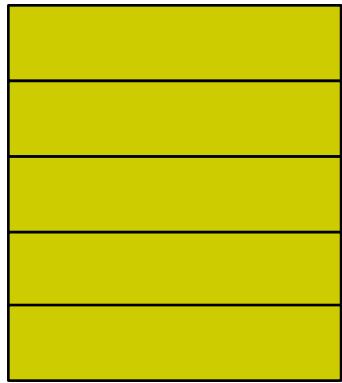
- *The subscript can begin with number 0. That is x[0] is allowed*
- For example, if we want to represent a set of five numbers, say (35, 40, 20, 57, 19), by an array variable number. then we may declare the variable number as follows

int number[5];

- and the con shown  
below:

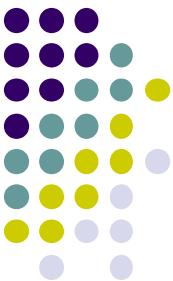


# One Dimensional Array



The values to the array elements can be assigned as follows:

```
number [0] = 35;
number [1] = 40;
number [2] = 20;
number [3] = 57;
number [4] = 19;
```



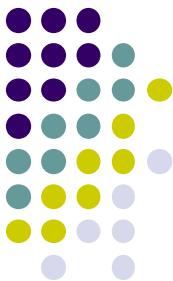
# One Dimensional Array

- This would cause the array number to store the values as shown below:

Number [0]	35
Number [1]	40
Number [2]	20
Number [3]	57
Number [4]	19

- These elements may be used in programs just like any other C variable. For example, the following are valid statements:

- $a = \text{number}[0] + 10;$
- $\text{number}[4] = \text{number}[0] + \text{number}[2];$
- $\text{number}[2] = x[5] + y[10];$
- $\text{value}[6] = \text{number}[i] * 3;$



# Declaration of One Dimensional Array

- Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in memory. The general form of array declaration is

***type variable-name[ size ];***

- The **type** specifies the type of element that will be contained in the array, such as **Int**, **float**, or **char** and the **size** indicates the **maximum number of elements** that can be stored inside the array. For example,

***float height [50] ;***

- declares the **height** to be an array containing **50** real elements.



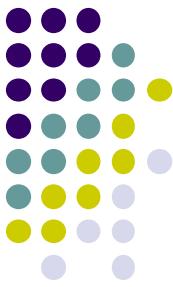
# Declaration of One Dimensional Array

- Any subscripts 0 to 49 are valid. Similarly,

```
int group[10] ;
```

- declares the **group** as an array to contain a maximum of **10 integer constants**. Remember:

- Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.
- The size should be either a **numeric constant or a symbolic constant**.



# Declaration of One Dimensional Array

- The C language treats character strings simply as **arrays of characters**.
- *The size in a character string represents the maximum number of characters that the string can hold.* For instance,

```
char name[10];
```

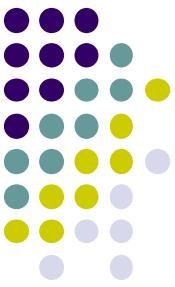
- declares the **name** as a character array (string) variable that can hold a maximum of **10 characters**.
- Suppose we read the following string constant into the string variable name. "**WELL DONE**"



# Declaration of One Dimensional Array

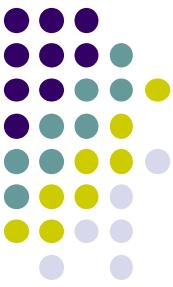
- Each character of the string is treated as an element of the array name and is stored in the memory as follows:
- When the compiler sees a character string, it terminates it with an additional **null character**.
- Thus, the element **name[10]** holds the **null character '\0'**.
- *When declaring character arrays, we must allow one extra element space for the null terminator.*

‘W’
‘E’
‘L’
‘L’
‘ ’
‘D’
‘O’
‘N’
‘E’
‘\0’



# Initialization of One Dimensional Array

- After an array is declared, its elements must be initialized. Otherwise, they will contain '**garbage**'. An array can be initialized at either of the following stages:
  - At compile time
  - At run time



# Compile Time Initialization

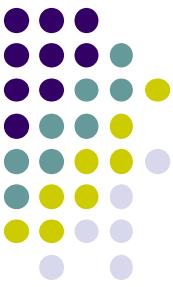
- We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is:

```
type array-name[size] = { list of values };
```

- The values in the list are separated by **commas**. For example, the statement

```
int number [3] = { 0,0,0 };
```

- will declare the variable **number** as an **array of size 3** and will assign zero to each element.



# Compile Time Initialization

- If the number of values in the list is **less than** the number of elements, then only that many elements will be initialized. The remaining elements will be set to **zero** automatically. For instance,

```
float total [5] ={0.0,15.75,-10};
```

- will initialize the first three elements to 0.0. 15.75, and -10.0 and the remaining two elements to zero.
- The size may be omitted. In such cases the compiler allocates enough space for all initialized elements.



## Compile Time Initialization

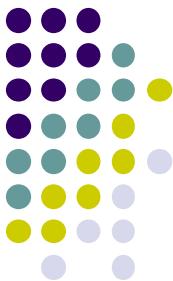
- For example, the statement

```
int counter[ ] = {1,1,1,1};
```

- will declare the **counter** array to contain four elements with initial values 1.
- This approach works fine as long as we initialize every element in the array.
- Character arrays may be initialized in a similar manner.  
Thus, the statement

```
char name[ ] = {'J', 'o', 'h', 'n', '\0'};
```

- declares the name to be an array of five characters, initialized with the string '**John**' ending with the null character.



## Compile Time Initialization

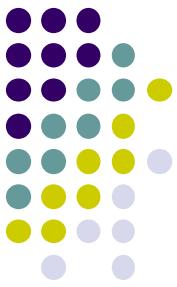
- Alternatively, we can assign the string literal directly as under:

```
char name[ ] = "John";
```

- Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases. the remaining elements are initialized to zero. if the array type is numeric and **NULL** if the type is char. For example,

```
int number [5] ={10, 20};
```

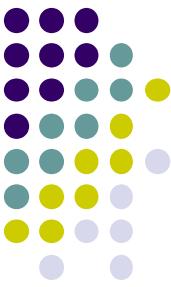
- will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0.



## Compile Time Initialization

- however, if we have more initializers than the declared size. the compiler will produce an error. That is, the statement

```
int number [3] = {10, 20, 30, 40};
```



# Run Time Initialization

- An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. For example, consider the following segment of a C program.

```
for (i = 0; i < 100; i = i+1)
{
    if (i < 50)
        sum[i] = 0.0;
    else
        sum[i] = 1.0;
}
```

- The first 50 elements of the array **sum** are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.



# Run Time Initialization

```
for (i = 0; i < 100; i = i+1)
{
    if (i < 50 )
        sum[i] = 0.0;
    else
        sum[i] =1.0;
}
```

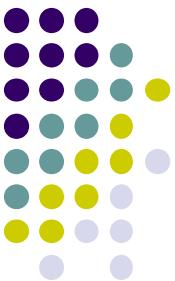
- » We can also use a read function such as **scanf** to initialize an array. For example, the statements
- »  

```
int x [3];
scanf( "%d%d%d", &x[0] , &x[1], &x[2] );
```
- » will initialize array elements with the values entered through the keyboard.



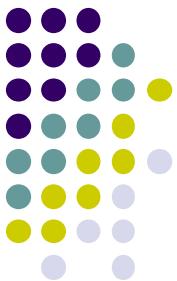
# Searching and Sorting

- Searching and sorting are the two most frequent operations performed on arrays. Computer Scientists have devised several data structures and searching and sorting techniques that facilitate rapid access to data stored in lists.
- *Sorting is the process of arranging elements in the list according to their values, in ascending or descending order.*
- A sorted list is called an **ordered list**.
- **Sorted** lists are especially important in list searching because they facilitate rapid search operations.



# Searching and Sorting

- Many sorting techniques are available. The three simple and most important among them are:
  - Bubble sort
  - Selection sort
  - Insertion sort
- Other sorting techniques include
  - Shell sort.
  - Merge sort and
  - Quick sort.
- **Searching is the process of finding the location of the specified element in a list.** The specified element is often called the search key. If the process of searching finds a match of the search key with a list element value, the search said to be successful; otherwise. it is unsuccessful.



# Searching and Sorting

- The two most commonly used search techniques are:
  - Sequential search
  - Binary search



# Two Dimensional Array

- So far we have discussed the array variables that can store a list of values. There could be situations where a table of values will have to be stored. Consider the following data table, which shows the value of sales of three items by four sales girls:

	Item 1	Item 2	Item 3
Salesgirl #1	310	275	365
Salesgirl #2	405	190	325
Salesgirl #3	210	235	240
Salesgirl #4	360	300	380



# Two Dimensional Array

- The table contains a total of 12 values, three in each line. We can think of this table as a matrix consisting of four rows and three columns. Each row represents the values of sales by a particular salesgirl and each column represents the values of sales of a particular item.
- C allows us to define such tables of items by using two-dimensional arrays. The table discussed above can be defined in C as

v[4][3]

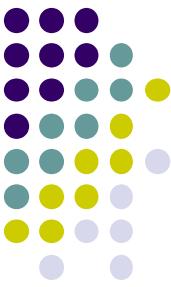


# Two Dimensional Array

- Two-dimensional arrays are declared as follows:

```
type array_name [row_size][column_size];
```

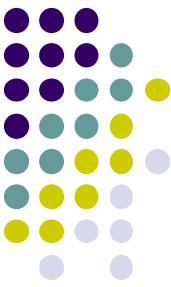
- Note that unlike most other languages, which use one pair of parentheses with commas to separate array sizes, C places each size in its own set of brackets.



# Two Dimensional Array

- Two-dimensional arrays are stored in memory, as shown in Fig. As with the single-dimensional arrays, each dimension of the array is indexed from zero to its maximum size minus one; the first index selects the row and the second index selects the column within that row.

	Item 1	Item 2	Item 3
Salesgirl #1	310	275	365
Salesgirl #2	10	190	325
Salesgirl #3	405	235	240



# Two Dimensional Array

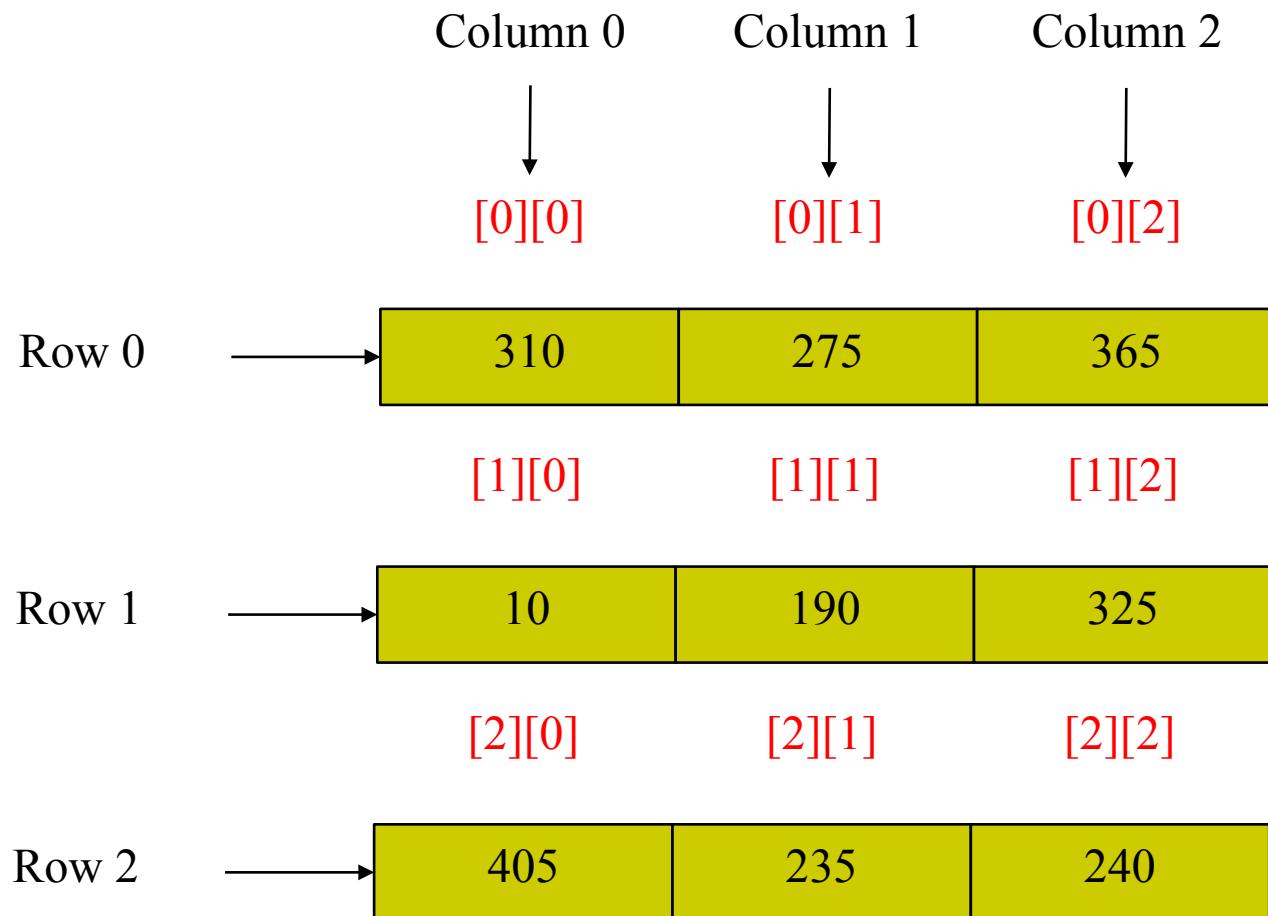
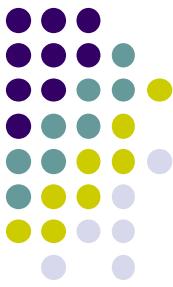


Fig: Representation of Two dimensional array



# Initializing Two Dimensional Array

- Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example,

```
int table[2] [3] = { 0,0,0,1,1,1};
```

- initializes the elements of the first row to zero and the second row to one. The initialization is done row by row. The above statement can be equivalently written as

```
int table[2] [3] = {{0,0,0}, {1,1,1}};
```

- by surrounding the elements of the each row by braces.



# Initializing Two Dimensional Array

- We can also initialize a two-dimensional array in the form of a matrix as shown below:

```
int table[2][3] = {  
    {0,0,0},  
    {1,1,1}  
};
```

- Note the syntax of the above statements. Commas are required after each brace that closes off a row, except in the case of the last row.

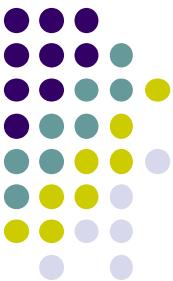


# Initializing Two Dimensional Array

- When the array is completely initialized with all values, explicitly, we need not specify the size of the first dimension. That is, the statement

```
int table [ ][3] = {  
                      { 0, 0, 0},  
                      { 1, 1, 1}  
};
```

- is permitted.



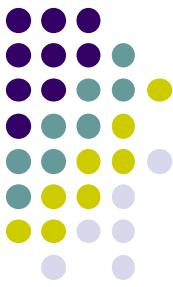
# Initializing Two Dimensional Array

- if the values are missing in an initializer, they are automatically set to zero. For instance, the statement

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

- will initialize the first two elements of the first row to one, the first element of the second row to two, and all other elements to zero.
- When all the elements are to be initialized to zero, the following short-cut method may be used.

```
int m[3] [5] = { {0}, {0}, {0}};
```



# Initializing Two Dimensional Array

- The first element of each row is explicitly initialized to zero while other elements are automatically initialized to zero. The following statement will also achieve the same result:

```
int m [3] [5] = { 0, 0};
```



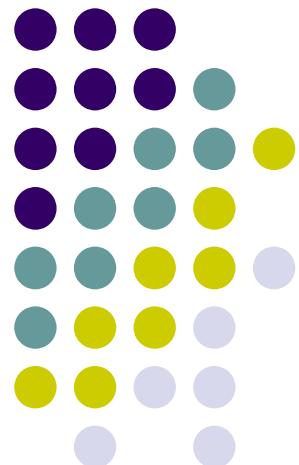
# Thank You

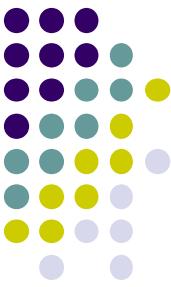
# Introduction to C Programming

---

## Chapter 8

### Characters Arrays and Strings





# Introduction

- ▶ A **string** is a sequence of characters that is treated as a single data item. Any group of characters (except double quote sign) defined between **double quotation marks** is a string constant. Example:

"Man is obviously made to think."

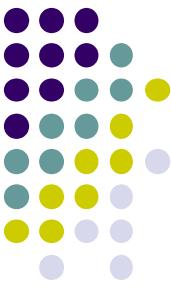
- ▶ If we want to include a double quote in the string to be printed, then we may use it with a **back slash** as shown below.

" \" Man is obviously made to think,\" said Pascal."

- ▶ For example,
- ▶ will output the string

*printf(" \" Well Done ! \"\");*

" Well Done ! "



# Introduction

- ▶ while the statement

```
printf(" Well Done !");
```

- ▶ will output the string

Well Done !

- ▶ Character strings are often used to build meaningful and readable programs. The common operations performed on character strings include:

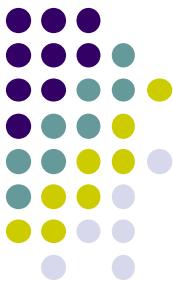
- ▶ Reading and writing strings.
- ▶ Combining strings together.
- ▶ Copying one string to another.
- ▶ Comparing strings for equality.
- ▶ Extracting a portion of a string.



# Declaring and Initializing String Variables

- C does not support strings as a data type. However, it allows us to represent strings as **character arrays**.
- In C, therefore, a **string variable** is any valid C variable name and is always declared as an array of characters. The general form of declaration of a string variable is:
- The **size** determines the **number of characters** in the **string\_name**. Some examples:

```
char string_name[ size];
```
- When the compiler assigns a string to a character array, it automatically supplies a **null character** ('\0') at the end of the string.

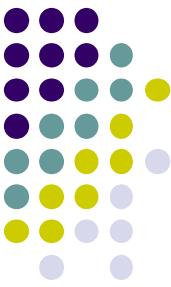


## Declaring and Initializing String Variables

- Therefore, the size should be equal to the maximum number of characters in the string plus one.
- Like numeric arrays, character arrays may be initialized when they are declared. C permits a character array to be initialized in either of the following two forms:

```
char city [9] = " NEW YORK ";
```

- The reason `char city [9]={'N','E', 'W', ','Y', 'O','R','K','\0'; }` is because the string NEW YORK contains 8 characters and one element space is provided for the null terminator.



## Declaring and Initializing String Variables

- Note that when we initialize a character array by listing its elements, we must supply explicitly the null terminator.
- C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized. For example. the statement
- defines `char string [ ] = {'G', 'O', 'O','D', '\0'};`



# Declaring and Initializing String Variables

- We can also declare the size much larger than the string size in the initializer. That is, the statement.

```
char str [10] = "GOOD";
```

- is permitted. In this case, the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

G	O	O	D	\0	\0	\0	\0	\0	\0
---	---	---	---	----	----	----	----	----	----

- However, the following declaration is illegal.

```
char str2 [3] = "GOOD";
```

- This will result in a compile time error.



# Declaring and Initializing String Variables

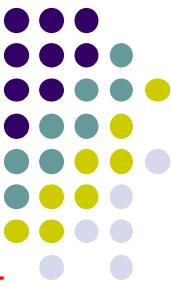
- Also note that we cannot separate the initialization from declaration. That is,

```
char str3[5] ;  
str3 = "GOOD";
```

- is not allowed. Similarly,

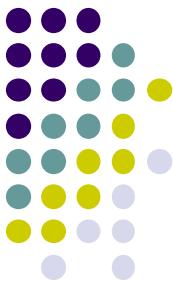
```
char sl[4] = "abc";  
char s2 [4] ;  
s2 = sl; /* Error */
```

- is not allowed. An array name cannot be used as the left operand of an assignment operator.



# Terminating Null Character

- You must be wondering, "why do we need a terminating null character?"
- As we know, a string is not a data type in C, but it is considered a data structure stored in an array. The string is a variable-length structure and is stored in a fixed-length array. The array size is not always the size of the string and most often it is much larger than the string stored in it.
- *Therefore, the last element of the array need not represent the end of the string. We need some way to determine the end of the string data and the null character serves as the "end-of-string" marker.*



# Reading String from Terminal

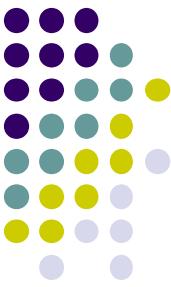
## Using scanf Function

- The familiar input function scanf can be used with `%s` format specification to read in a string of characters. Example:

```
char address [10];
scanf ("%s", address);
```

- The problem with the scanf function is that it terminates its input on the first white space it finds.
- A white space includes blanks, tabs, carriage returns, form feeds, and new lines. Therefore, if the following line of text is typed in at the terminal,

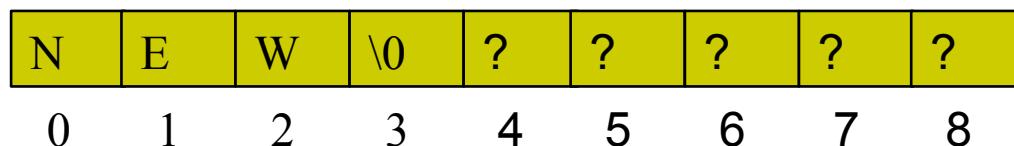
- then only the string "NEW" will be read into the array address, since the blank space after the word 'NEW' will terminate the reading of string.

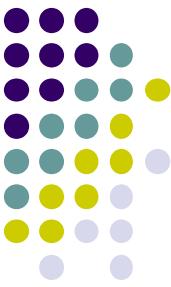


# Reading String from Terminal

## Using `scanf` Function

- The `scanf` function automatically terminates the string that is read with a **null character** and therefore the character array should be large enough to hold the input string plus the null character.
- Note that unlike previous `scanf` calls, in the case of character arrays, the ampersand (&) is not required before the variable name.
- The address array is created in the memory as shown below:





# Reading String from Terminal

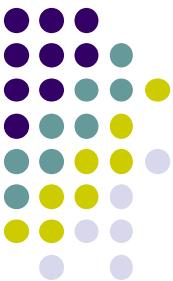
- Note that the unused locations are filled with garbage.
- If we want to read the entire line "NEW YORK", then we may use two character arrays of appropriate sizes. That is,

```
char adr1[5], adr2[5];
scanf("%s %s", adr1, adr2);
```

- with the line of text

NEW YORK

- will assign the string "NEW" to adr1 and "YORK" to adr2.

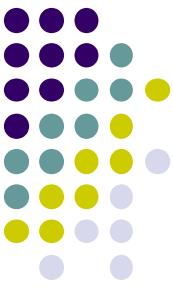


# Reading String from Terminal

- We can also specify the field width using the form `%ws` in the `scanf` statement for reading a specified number of characters from the input string. Example:

```
scanf("%ws", name);
```

- Here, two things may happen.
- *The width w is equal to or greater than the number of characters typed in. The entire string will be stored in the string variable.*
- *The width w is less than the number of characters in the string. The excess characters will be truncated and left unread.*
- Consider the following statements: `char name[10];`  
`scanf(":5s", name);`



# Reading String from Terminal

- Consider the following statements:

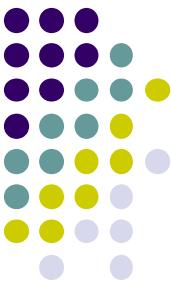
```
char name[10];  
scanf("%5s", name);
```

- The input string RAM will be stored as:

R	A	M	\0	?	?	?	?	?
0	1	2	3	4	5	6	7	8

- The input string KRISHNA will be stored as:

K	R	I	S	H	\0	?	?	?
0	1	2	3	4	5	6	7	8

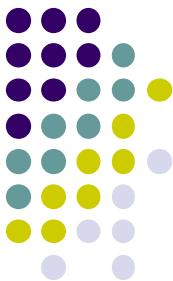


# Reading a line of Text

- We have seen just now that `scanf` with `%s` or `%ws` can read only strings without **whitespaces**. That is, they cannot be used for reading a text containing more than one word.
- However, C supports a format specification known as the **edit set conversion code** `%[. .]` that can be used to read a line containing a variety of characters, including whitespaces.
- For example, the program segment

```
char line [80];
scanf ("%[^\\n]", line);
printf ("%s", line);
```

- will read a line of input from the keyboard and display it on the screen. We would very rarely use this method, as C supports an intrinsic string function to do this job. This is discussed in the next section.

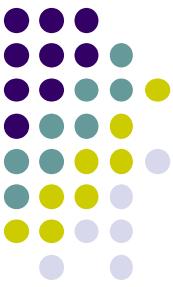


# Using getchar and gets Function

- We have discussed in Chapter 4 as to how to read a single character from the terminal, using the function `getchar`. We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array.
- The reading is terminated when the `newline character ('\n')` is entered and the null character is then inserted at the end of the string.
- The `getchar` function call takes the form:

```
char ch;  
ch = getchar();
```

- *Note that the `getchar` function has no parameters.*



# Using getchar and gets Function

- Another and more convenient method of reading a string of text containing whitespaces is to use the library function `gets` available in the `<stdio.h>` header file. This is a simple function with one string parameter and called as under:

`gets (str);`

- `str` is a string variable declared properly. It reads characters into `str` from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike `scanf`, it does not skip whitespaces. For example the code segment

```
char line [80] ;
gets (line);
printf ("%s", line);
```



# Using getchar and gets Function

- reads a line of text from the keyboard and displays it on the screen. The last two statements may be combined as follows:

```
printf ("%s", gets (line));
```

- C does not provide operators that work on strings directly. *For instance we cannot assign one string to another directly.* For example, the assignment statements.

```
string = "ABC";  
string1 = string2;
```

- are not valid. If we want to copy all the characters in **string2** into **string1**, we may do so on a character-by-character basis.



# Writings String to Screen

## Using printf Function

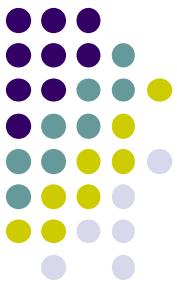
→ We have used extensively the `printf` function with `%s` format to print strings to the screen. The format `%s` can be used to display an array of characters that is terminated by the null character. For example, the statement

```
printf ("%s", name);
```

→ can be used to display the entire contents of the array `name`.

→ We can also specify the precision with which the array is displayed. For instance, the specification

→ indicates that the first four characters are to be printed in a field width of 10 columns.



# Using `printf` Function

However, if we include the minus sign in the specification (e.g., `%-10.4s`), the string will be printed left-justified.



# Using Putchar and puts functions

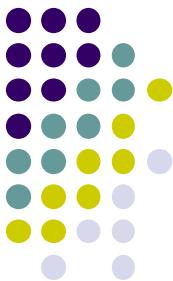
- Like `getchar`, C supports another character handling function `putchar` to output the values of character variables. It takes the following form:

```
char ch = 'A';  
putchar (ch);
```

- The function `putchar` requires one parameter. This statement is equivalent to:

```
printf ("%c", ch);
```

- We have used `putchar` function in Chapter 4 to write characters to the screen.



# Using Putchar and puts functions

- We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
char name [6] = "PARIS" ;  
for (l=0, i <5; i++)  
    putchar (name [1] );  
putchar (' \ n );
```

- Another and more convenient way of printing string values is to use the function **puts** declared in the header file **<stdio.h>**. This is a one parameter function and invoked as under:

```
puts ( str ) ;
```

- where **str** is a string variable containing a string value.

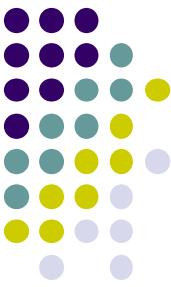


# Using Putchar and puts functions

- This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char line [80];
gets (line) ;
puts (line) ;
```

- reads a line of text from the keyboard and displays it on the screen. Note that the syntax is very simple compared to using the `scanf` and `printf` statements



## Arithmetic operations on characters

- C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into an integer value by the system. The integer value depends on the local character set of the system.
- To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then,

```
x = 'a' ;  
printf("%d\n",x);
```

- will display the number 97 on the screen.



## Arithmetic operations on characters

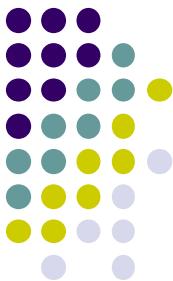
- It is also possible to perform arithmetic operations on the character constants and variables. For example,

```
x = 'z' - 1;
```

- is a valid statement. In ASCII, the value of 'z' is 122 and therefore, the statement will assign the value 121 to the variable x.
- We may also use character constants in relational expressions. For example, the expression

```
ch >= 'A' && ch <= 'Z'
```

- would test whether the character contained in the variable ch is an upper-case letter.



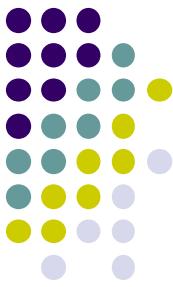
## Arithmetic operations on characters

- We can convert a character digit to its equivalent integer value using the following relationship:

$$x = \text{character} - '0' ;$$

- where **x** is defined as an integer variable and **character** contains the character digit. For example, let us assume that the **character** contains the digit '**7**', Then,

$$\begin{aligned}x &= \text{ASCII value of '7'} - \text{ASCII value of '0'} \\&= 55 - 48 \\&= 7\end{aligned}$$

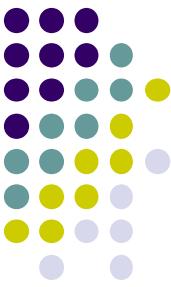


## Arithmetic operations on characters

- The C library supports a function that converts a string of digits into their integer values. The function takes the form  
 $x = \text{atoi}(\text{string});$
- **x** is in integer variable and **string** is a character array containing a string of digits. Consider the following segment of a program:

```
number = "1988";
year = atoi(number);
```

- **number** is a string variable which is assigned the string constant **"1988"**. The function atoi converts the string "1988" (contained in number) to its numeric equivalent 1988 and assigns it to the integer variable **year**. String conversion functions are stored in the header file <std.lib.h>.

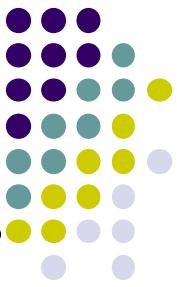


## Putting Strings Together

- Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;  
string2 = string1 + "hello";
```

- are not valid. The characters from `string1` and `string2` should be copied into the `string3` one after the other. The size of the array `string3` should be large enough to hold the total characters.



## Comparison of two strings

- Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(namel == name2)  
if (name == "ABC")
```

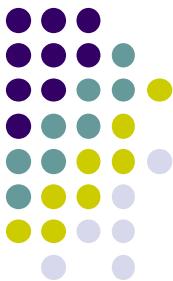
- are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first.



# String Handling Functions

- Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations discussed so far. Following are the most commonly used string-handling functions.

Function	Action
<b>strcat()</b>	Concatenates two strings
<b>strcmp()</b>	Compares two strings
<b>strcpy()</b>	Copies one string over another
<b>strlen()</b>	Finds the length of a string

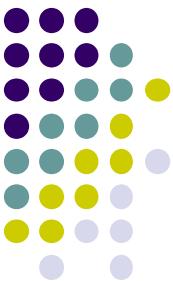


# strcat() Function

- The **strcat** function joins two strings. together. It takes the following form:

```
strcat(string1, string2);
```

- **string1** and **string2** are character arrays. When the function **strcat** is executed, **string2** is appended to **string1**.
- It does so by removing the null character at the end of **string1** and placing **string2** from there. The string at **string2** remains unchanged



# strcat() Function

- For example, consider the following three strings:

Part =1	0	1	2	3	4	5	6	7	8	9	0	1
	V	E	R	Y		\0	?	?	?	?	?	?
Part =2	0	1	2	3	4	5	6					
	G	O	O	D	\0		?					
Part =3	0	1	2	3	4	5	6					
	B	A	D	\0			?					



# Strcat

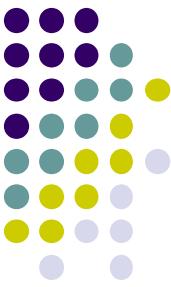
- Execution of the statement

`strcat(part1,part2)`

- Will result in

	0	1	2	3	4	5	6	7	8	9	0	1
Part =1	V	E	R	Y		G	O	O	D	\0		
Part =2	G	O	O	D	\0							

- We must make sure that the size of `string1` (to which `string2` is appended) is large enough to accommodate the final string.



# Strcat

- **strcat** function may also append a string constant to a string variable. The following is valid:

```
strcat (part1 , "GOOD" );
```

- C permits nesting of **strcat** functions. For example, the statement

```
strcat (strcat (string1, string2) , string3);
```

- is allowed and concatenates all the three strings together. The resultant string is stored in **string1**.



# strcmp() Function

- The **strcmp** function compares two strings identified by the arguments and has a value **0** if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

```
strcmp(string1 , string2);
```

- **string1** and **string2** may be string variables or string constants. Examples are:

```
strcmp(namel , name2) ;  
strcmp (nausel , "John") ;  
strcmp ("Rom" , "Ram") ;
```

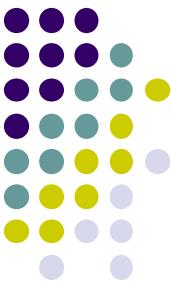


# strcmp() Function

- Our major concern is to determine whether the strings are equal; if not, which is alphabetically above. The value of the mismatch is rarely important. For example, the statement

```
strcmp("their" , "there") ;
```

- will return a value of **-9** which is the numeric difference between **ASCII "i"** and **ASCII "r"**.
- That is, **"i"** minus **"r"** in ASCII code is -9. If the value is negative, **string1** is alphabetically above **string2**.



# strcpy() Function

- The **strcpy** function works almost like a string assignment operator. It takes the form:

```
strcpy(string1, string2);
```

- and assigns the contents of **string2** to **string1**. **string2** may be a character array variable or a string constant. For example, the statement

```
strcpy(city, "DELHI");
```

- will assign the string "**DELHI**" to the string variable **city**. Similarly, the statement

```
strcpy(city1, city2);
```

- will assign the contents of the string variable **city2** to the string variable **city1**. The size of the array **city1** should be large enough to receive the contents of **city2**.



# strlen() Function

- This function counts and returns the number of characters in a string. It takes the form

```
n = strlen(string);
```

- Where **n** is an integer variable, which receives the value of the length of the string. The argument may be a **string** constant. The counting ends at the first null character.



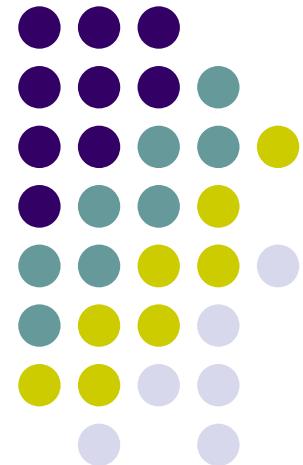
# Thank You



# Introduction to C Programming

## Chapter 9

### User Defined Functions



**Eftekhar Hossain**

*Lecturer,*

*Dept. of ETE, CUET*

# Introduction

- ▶ We have used functions in every program that we have discussed so far. However, they have been primarily limited to the three functions, namely, **main, printf, and scanf.**
- ▶ In this chapter, we shall consider in detail the following:
  - ▶ **How a function is designed?**
  - ▶ **How a function is integrated into a program?**
  - ▶ **How two or more functions are put together? and**
  - ▶ **How they communicate with one another?**
- ▶ C functions can be classified into two categories, namely,  
**library functions**  
and  
**user-defined functions**

# Introduction

- ▶ main is an example of user-defined functions.
- ▶ printf and scanf belong to the category of library functions.
- ▶ *The main distinction between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program.*
- ▶ *However, a user-defined function can later become a part of the C program library. In fact, this is one of the strengths of C language.*

## Need for User Defined Functions

- As pointed out earlier, main is a specially recognized function in C.
- Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems.
- The program may become *too large* and *complex* and as a result the task of *debugging*, *testing*, and *Maintaining* becomes difficult.
- If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit.
- These independently coded programs are *called subprograms* that are much easier to understand, debug, and test. *In C, such subprograms are referred to as 'functions'.*

# Need for User Defined Functions

- There are times when certain type of operations or calculations are **repeated at many points** throughout a program.
- For instance, we might use the **factorial of a number** at several points in the program. In such situations, we may repeat the program statements wherever they are needed.
- Another approach is to design a function that can be called and used whenever required. **This saves both time and space.**

# Need for User Defined Functions

- This "**division**" approach clearly results in a number of advantages.
  - **It facilitates top-down modular programming** as shown in Figure. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level function are addressed later.
  - *The length of a source program can be reduced by using functions at appropriate places.*
  - *It is easy to locate and isolate a faulty function for further investigations.*
  - **A function may be used by many other programs.** This means that a C programmer can build on what others have already done, instead of starting all over again from scratch.

# Need for User Defined Functions

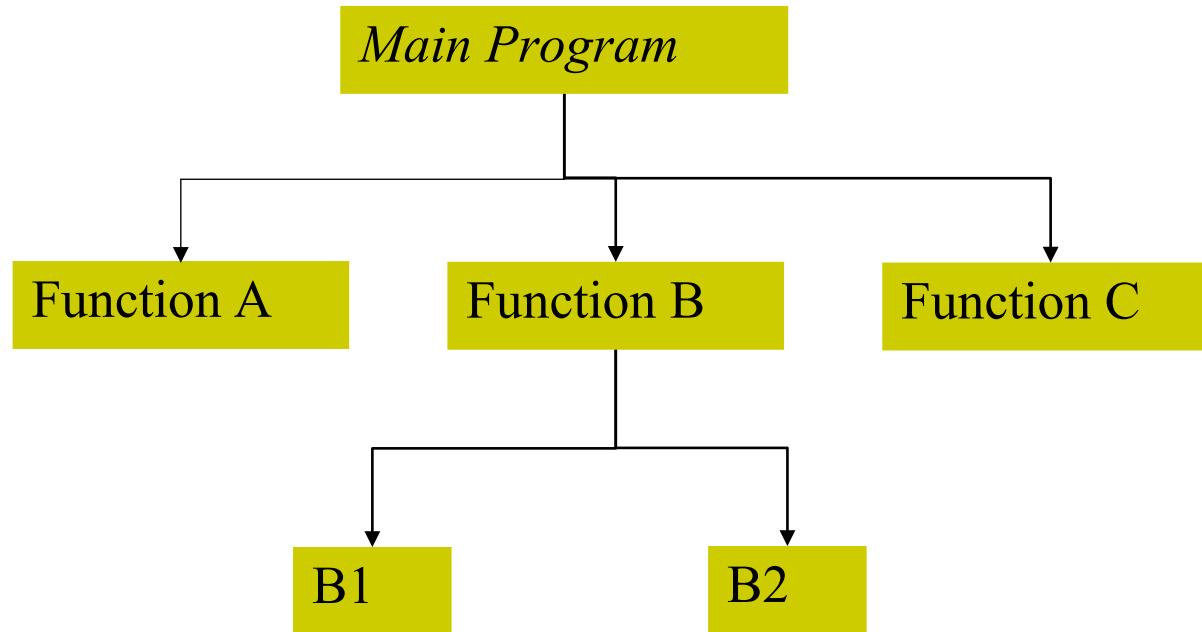


Fig: Top down modular programming using functions

## A Multi Function Program

- A function is a **self-contained block of code** that performs a particular task.
- Once a function has been designed and packed, it can be treated as a '**black box**' that takes some data from the main program and returns a value. The inner details of operation are invisible to the rest of the program. All that the program knows about a function is:

*What goes in and what comes out.*

***Every C program can be designed using a collection of these black boxes known as functions.***

# A Multi Function Program

- Consider a set of statements as shown below:

```
void printline (void)
{
    int i;
    for (i= 1; i<40 ; i++)
        print("-");
        printf("\n");
}
```

- The above set of statements defines a function called **printline**, which could print a line of 39-character length. This function can be used in a program as follows:

# A Multi Function Program

```
void printline(void); /* declaration */
main( )
{
    printline( );
    printf("This illustrates the use of C functions\n");
    printline();
}
void printline(void)
{
    int i;
    for(i=1; i<40; i++)
        printf("-");
    printf("\n");
}
```

**Output:**

---

This illustrates the use of C functions

---

## A Multi Function Program

- The above program contains two user-defined functions:
  - main() function**
  - printline() function**
- As we know, the program execution always begins with the main function. During execution of the main, the first statement encountered is  
**printline( );**
- which indicates that the function **printline** is to be executed. At this point, the program control is transferred to the function **printline**.
- After executing the **printline** function, which outputs a line of 39 character length, the control is transferred back to the **main**.

## A Multi Function Program

- Now, the execution continues at the point where the function call was executed.
- After executing the **printf** statement, the control is again transferred to the **printline** function for printing the line once more.
- The **main** function calls the user-defined **printline** function two times and the library function **printf** once.
- We may notice that the **printline** function itself calls the library function **printf** 39 times repeatedly.

*Any function can call any other function.*

*In fact, it can call itself.*

# A Multi Function Program

- *A 'called function' can also call another function.*
- *A function can be called more than once.*
- In fact, this is one of the main features of using functions.
  
- A called function can be placed either before or after the calling function. However, it is the usual practice to put all the called functions at the end. See the box "Modular Programming".

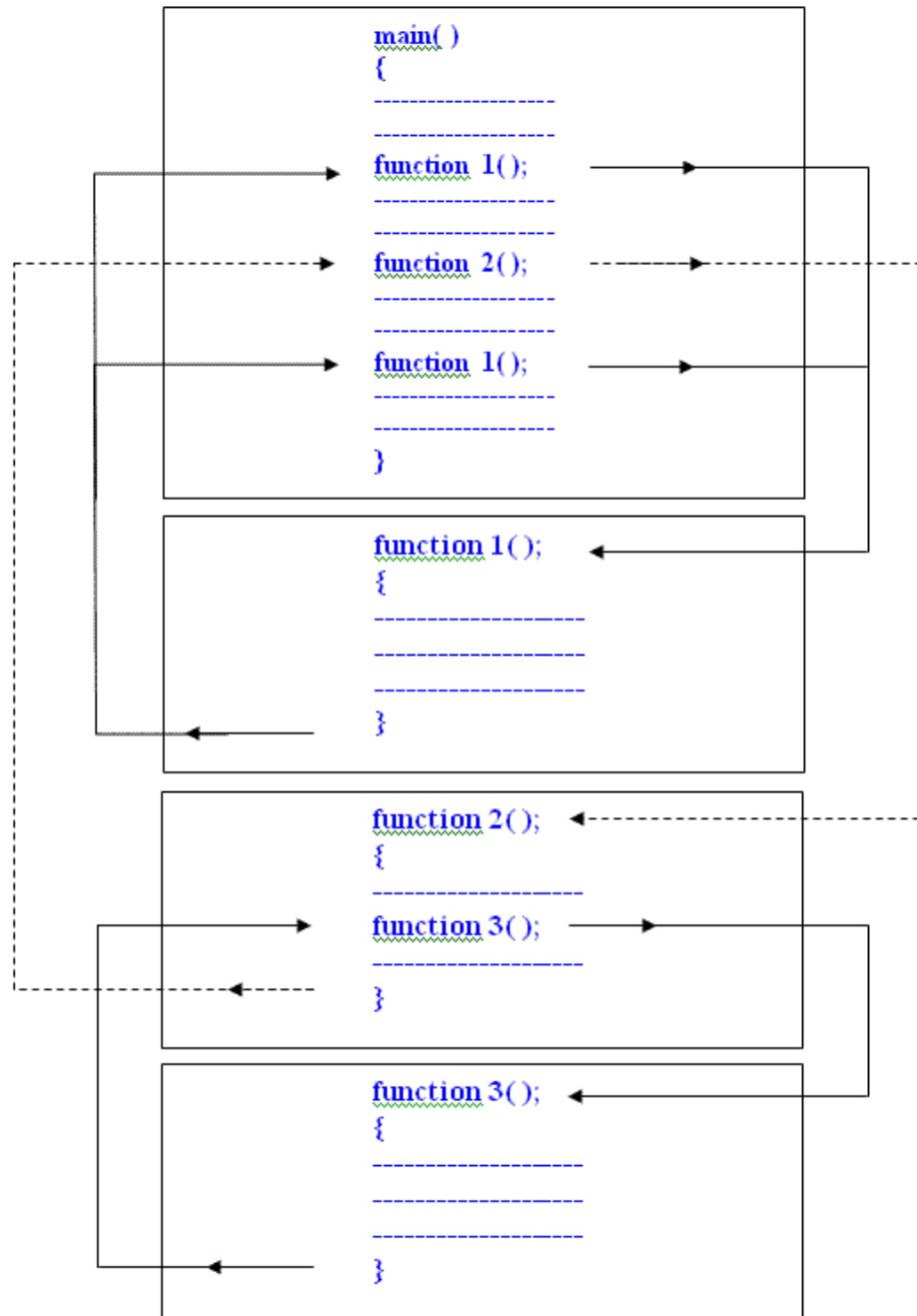


Figure : Flow of control in a multi-function program.

# Modular Programming

- ▶ Modular programming is a strategy applied to the design and development of software systems.
- ▶ It is defined as **organizing a larger program into small, independent program segments called modules that are separately named and individually callable program units.** These modules are carefully integrated to become a software system that satisfies the system requirements.
- ▶ ***It is basically a "divide-and-conquer" approach to problem solving.***

# Modular Programming

- ▶ Modules are identified and designed such that they can be organized into a top-down hierarchical structure (similar to an organization chart). In C, each module refers to a function that is responsible for a single task. Some characteristics of modular programming are:
  - ▶ Each module should do only one thing.
  - ▶ Communication between modules is allowed only by a calling module.
  - ▶ A module can be called by one and only one higher module.
  - ▶ No communication can take place directly between modules that do not have calling-called relationship.
  - ▶ All modules are designed as single-entry, single-exit systems using control structures.

## Elements of User Defined Functions

- ▶ We have discussed and used a variety of data types and variables in our programs so far. However, declaration and use of these variables were primarily done inside the main function.
- ▶ As we mentioned in Chapter 4, functions are classified as one of the derived data types in C. We can therefore define functions and use them like any other variables in C programs. It is therefore not a surprise to note that there exist some similarities between functions and variables in C.
  
- ▶ Both **function names and variable names are considered identifiers** and therefore they must adhere to the rules for identifiers
- ▶ **Like variables, functions have types**(such as int) associated with them.
- ▶ **Like variables function names and their types must be declared** and defined before they are used in a program.

# Elements of User Defined Functions

- ▶ In order to make use of a user-defined function, we need to establish **three elements** that are related to functions.

## Function definition.

- ▶ The function definition is an independent program module that is specially written to implement the requirements of the function.

## Function call.

- ▶ In order to use this function we need to invoke it at a required place in the program. This is known as the function call.

## Function declaration.

- ▶ The program (or a function) that calls the function is referred to as the calling program or calling function. The calling program should declare any function (like declaration of a variable) that is to be used later in the program. **This is known as the function declaration or function prototype.**

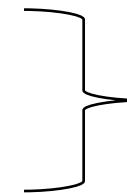
# Definitions of Functions

- A function definition, also known as function implementation shall include the following elements:

function name;

function type;

list of parameters;

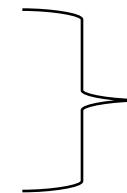


**Function header**

local variable declarations:

function statements: and

a return statement.



**Function body**

- A general format of a function definition to implement these two parts is given below:

# Definitions of Functions

- A general format of a function definition to implement these two parts is given below:

```
function_type function_name(parameter list)
{
    local variable declaration;
    executable statement1
    executable statement2;
    .....
    .....
    return statement;
}
```

- The first line `function_type function_name(parameter list)` is known as the **function header** and the statements within the opening and Closing braces constitute **the function body**, which is a compound statement.

# Function Header

- The function header consists of three parts:
  1. *the function type (also known as return type),*
  2. *the function name and*
  3. *the formal parameter list.*
- Note that a semicolon is not used at the end of the function header.

# Name and Type

- The function **type** specifies the type of value (like float or double) that the function is expected to **return** to the program **calling the function**.
- If the return type is not explicitly specified, C will assume that it is an **integer type**.
- If the function is not returning anything, then we need to specify the return type as **void**. Remember **void is one of the fundamental data types in C**.
- It is a good programming practice to code explicitly the return type, even when it is an integer. The value returned is the output produced by the function. The function name is any valid C identifier and therefore must follow the same rules of formation as other variable names in C.

# Name and Type

- The name should be appropriate to the task performed by the function. However, care must be exercised to avoid duplicating library routine names or operating system commands.

# Formal Parameter List

- The parameter list declares the variables that will receive the data sent by the calling program.
- They serve as input data to the function to carry out the specified task.
- Since they represent actual input values, they are often referred to as **formal parameters**. These parameters can also be used to send values to the calling programs.
- **The parameters are also known as arguments.** The parameter list contains declaration of variables separated by commas and surrounded by parentheses. Examples:

```
float quadratic (int a, int b, int c) {.... }  
double power (double x, int n) {.....}  
float mul (float x, float y) {.... }  
int sum (int a, int b) { ....}
```

# Formal Parameter List

- Remember, **there is no semicolon** after the closing parenthesis. Note that the declaration of parameter variables cannot be combined.
- That is, int sum (int a,b) is illegal.
- A function need not always receive values from the calling program. In such cases, functions have no formal parameters. To indicate that the parameter list is empty, we use the keyword void between the parentheses as in ,,

```
void printline (void)
{
}
```

- This function neither receives any input values nor returns back any value.

# Function Body

- The function body contains the declarations and statements necessary for performing the required task. The body enclosed in braces, contains three parts, in the order given below:
  - ⇒ *Local declarations that specify the variables needed by the function.*
  - ⇒ *Function statements that perform the task of the function.*
  - ⇒ *A return statement that returns the value evaluated by the function.*
- If a function does not return any value (like the printline function), we can omit the return statement.
- However, note that its return type should be specified as void. Again, it is nice to have a return statement even for void functions. Some examples of typical function definitions are:

# Function Body

```
float mult (float x, float y) {  
    float result; /* local variable */  
    result = x * y; /* computes the product */  
    return result; /* returns the result */  
}
```

```
void sum (int a, int b) {  
  
    printf ("sum = %s", a + b); /* no local variables */  
    return; /* optional */  
}
```

```
void display (void) {  
    /* no local variables */  
    printf ("No type, no parameters");  
    /* no return statement */  
}
```

# Function Body

- Note

When a function reaches its return statement, the control is transferred back to the calling program. In the absence of a return statement, the closing brace acts, as a **void return**

A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

# Return values and their types

- As pointed out earlier, a function may or may not send back any value to the calling function. If it does. it is done through the **return** statement.
- *While it is possible to pass to the called function any number of values, the called function can only return one value per call, at the most.*
- The **return** statement can take one of the following forms:
  - return;**
  - or return (expression);**
- The first, **the 'plain' return** does not return any value; it acts much as the closing brace of the function.
- When a return is encountered, the control is immediately passes back to the calling function. An example of the use of a simple return is as follows:

```
if (error)  
return;
```

# Return values and their types

- The second form of return with an expression returns the value of the expression. For example, the function

```
int mul (int x, int y)
{
    int p;
    p =x*y;
    return (p) ;
}
```

- returns the value of **p** which is the product of the values of x and y.
- The last two statements can be combined into one statement as follows.

```
return (x*y);
```

# Return values and their types

- A function may have more than one return statements. This situation arises when the value returned is based on certain conditions. For example:

```
if( x <=0 )  
    return (0) ;  
else  
    return (1);
```

- What type of data does a function return?
- **All functions by default return int type data** But what happens if a function must return some other type?
- We can force a function to return a particular type of data by using a type specifier in the function header as discussed earlier.

# Return values and their types

- When a value is returned, it is automatically cast to the function's type. In functions that do computations using doubles, yet return (fits, the returned value will be truncated to an integer. For instance, the function

```
int product (void)
{
    return (2.5*3.0);
}
```

- will return the value 7. only the integer part of the result.

# Function Calls

- A function can be called by simply using the function name followed by a list of actual parameters (or arguments). if any. enclosed in parentheses. Example:

```
main( )  
{  
    int y;  
    y = multi(10,5);           /*Function call */  
    printf("%d\n", y);  
  
}
```

- When the compiler encounters a function call, the control is transferred to the function **multi()**. This function is then executed line by line as described and a value is returned when a return statement is encountered. This value is assigned to y.

# Function Calls

- This is illustrated below:

```
main()
{
    int y;
    → y = multi(10,5); /* call */
}
```

```
int multi(int x, int y)
{
    int p;          /* local variable*/
    p = x*y;       /* x =10, y= 5 */
    return(p);
}
```

# Function Calls

- The function call sends two integer values 10 and 5 to the function.

```
int multi (int x, int y)
```

- which are assigned to ~~x and y respectively~~. The function computes the product x and y. assigns the result to the local variable p. and then returns the value 25 to the main where it is assigned to y again.
- There are many different ways to call a function. Listed below are some of the ways the function **multi** can be invoked.

```
multi (10, 5)
multi (m, 5)
multi (10, n)
multi (m, n)
multi (m + 5, 10)
multi (10, multi(m.n))
multi (expression1 expression2)
```

# Function Calls

- Note that the sixth call uses its own call as its one of the parameters. When we use expressions, they should be evaluated to single values that can be passed as actual parameters.
- A function which returns a value can be used in expressions like any other variable Each of the following statements is valid:

```
printf ("%d\n", mul (p,q));
y = mul (p,q) / (p+q);
if (mul (m,n)>total ) printf ("large");
```

# Function Calls

- However, a function cannot be used on the right side of an assignment statement. For instance,

```
mul (a ,b) = 15;
```

- is invalid.
- A function that does not return any value may not be used in expressions; but can be called in to perform certain tasks specified in the function. The function **printline( )** discussed in Section 9.3 belongs to this category. Such functions may be called in by simply stating their names as independent statements. Example:
- Note the presence of a semicolon at the end.

```
main( )
{
    printline( );
}
```

# Function Calls

- » A function call is a postfix expression.
- » The operator ( . ) is at a very high level of precedence. Therefore, when a **function call** is used as a part of an expression, it will be evaluated first, unless parentheses are used to change the order of precedence.
- » In a function call, the **function name is the operand** and the parentheses set ( . ) which contains the actual parameters is the operator.
- » *The actual parameters must match the function's formal parameters in type, order and number.* Multiple actual parameters must be separated by commas.'

# Function Calls

## NOTE:

- If the actual parameters are more than the formal parameters, the extra actual arguments will be discarded.
- On the other hand, if the actuals are less than the formals, the unmatched formal arguments will be initialized to some garbage.
- Any mismatch in data types may also result in some garbage values.

# Function Declaration

- Like variables, all functions in a C program must be declared. before they are invoked. A function declaration (also known as function prototype) consists of four parts.
  - Function type (return type)
  - Function name.
  - Parameter list.
  - Terminating semicolon.
- They are coded in the following format:

```
function-type function-name (parameter list) ;
```

# Function Declaration

- This is very similar to the function header line except the terminating semicolon. For example, multi function defined in the previous section will be declared as:

```
int multi (int m, int n); /* Function prototype */
```

# Points to Note

- The parameter list must be separated by commas.
- The parameter names do not need to be the same in the prototype declaration and the function definition.
- The types must match the types of parameters in the function definition, in number and order.
- Use of parameter names in the declaration is optional.
- If the function has no formal parameters, the list is written as (**void**).
- The return type is optional. when the function returns **int** type data.
- The retype must be **void** if no value is returned.
- When the declared types do not match with the types in the function definition, compiler will produce an error.

# Points to Note

- Equally acceptable forms of declaration of multi function are:

```
int multi (int, int);  
        multi (int a, int b);  
        multi (int, int);
```

- When a function does not take any parameters and does not return any value, its prototype is written as:

```
void display (void);
```

# Points to Note

- A prototype declaration may be placed in two places in a program.
  - ⇒ Above all the functions (including main).
  - ⇒ Inside a function definition.
- When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as **a global prototype**. Such declarations are available for all the functions in the program.
- When we place it in a function definition (in the local declaration section), the prototype is called a **local prototype**. Such declarations are primarily used by the functions containing them.
- The place of declaration of a function defines a region in a program in which the function may be used by other functions. **This region is known as the scope of the function.**

# Prototypes Yes or NO

- Prototype declarations are not essential.
- If a function has not been declared before it is used, C will assume that its details evadable at me time of linking.
- Since the prototype is not available. C will assume that the **return type is an integer** and that the types of parameters match the formal definitions.
- If these assumptions are wrong, the linker will fail and we will have to change the program. The moral is that we must always include prototype declarations: preferably in global declaration section

# Parameters Everywhere

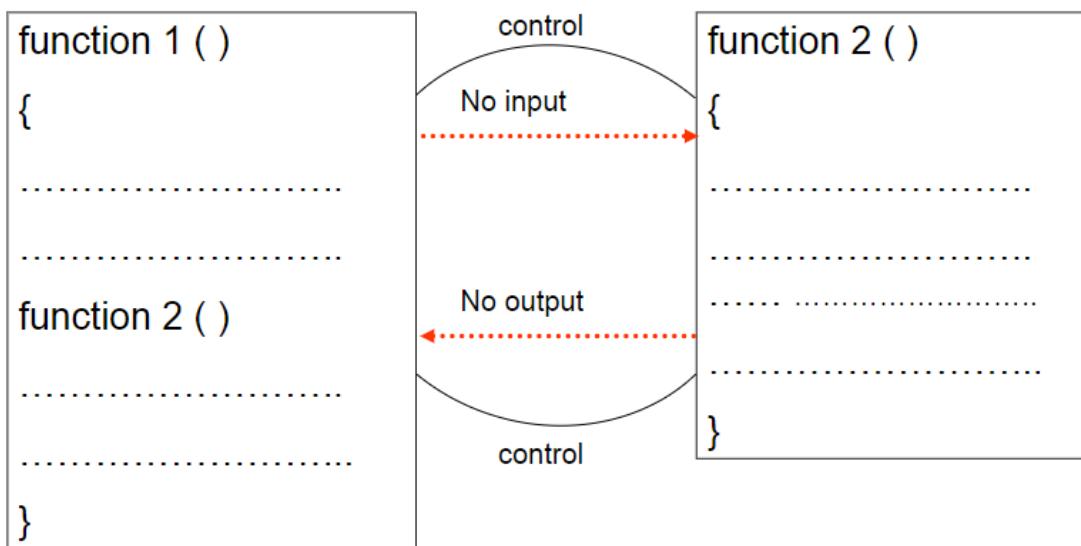
- Parameters (also known as arguments) are used in three places:
  - **in declaration (prototypes),**
  - **in function call, and**
  - **in function definition.**
- The parameters used in prototypes and function definitions are called **formal parameters** and those used in function calls are called **actual parameters**.
- *Actual parameters used in a calling statement may be simple constants, variables or expressions.*
- The formal and actual parameters must match exactly in type, order and number. Their names, however, do not need to match.

# Category of Functions

- A function, depending on whether arguments are present or not and whether a value is returned or not, may belong to one of the following categories:
- Category 1: Functions with no arguments and no return values.
- Category 2: Functions with arguments and no return values
- Category 3: Functions with arguments and one return value.
- Category 4: Functions with no arguments but return a value.
- Category 5: Functions that return multiple values.

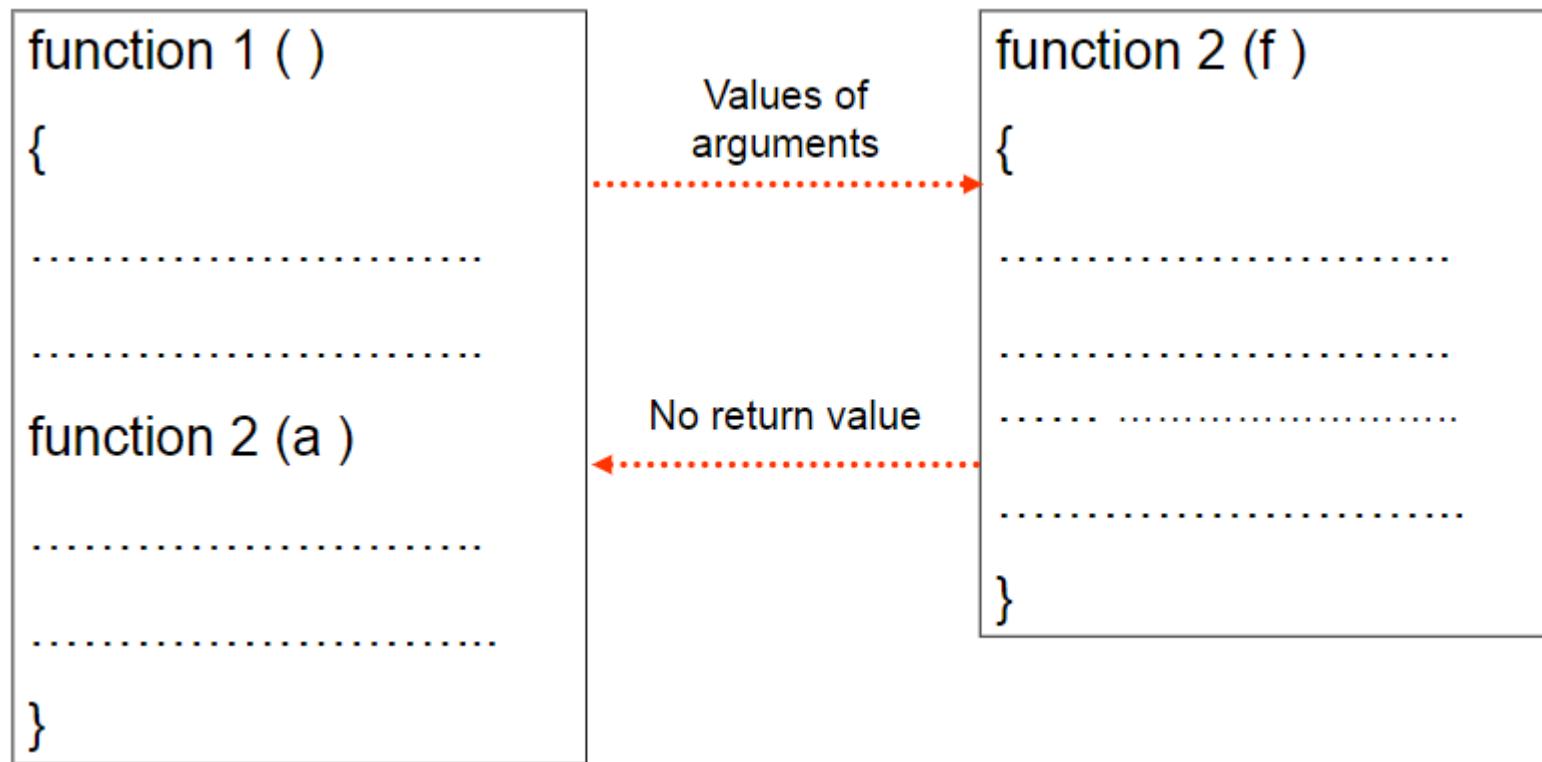
# No arguments and No Return values

- When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does receive data from the called function. In effect, there is no data transfer between the calling function and the called function. The dotted lines indicate that there is only a transfer of control but not data.



# Arguments but No Return values

- The nature of data communication between the calling function and the called function with arguments but no return value is shown in Fig. 9.5.



# Arguments but No Return values

- We shall modify the definitions of both the called functions to include arguments as follows:

```
void printline(char ch)
void value(float p, float r, int n)
```

- The arguments ch, p, r, and n are called the formal arguments. The calling function can now send values to these arguments using function calls containing appropriate arguments. For example, the function call

```
value(500,0,12,5)
```

- would send the values 500,0.12 and 5 to the function

```
void value( float p, float r, int n)
```

- and assign 500 to p, 0.12 to r and 5 to n.

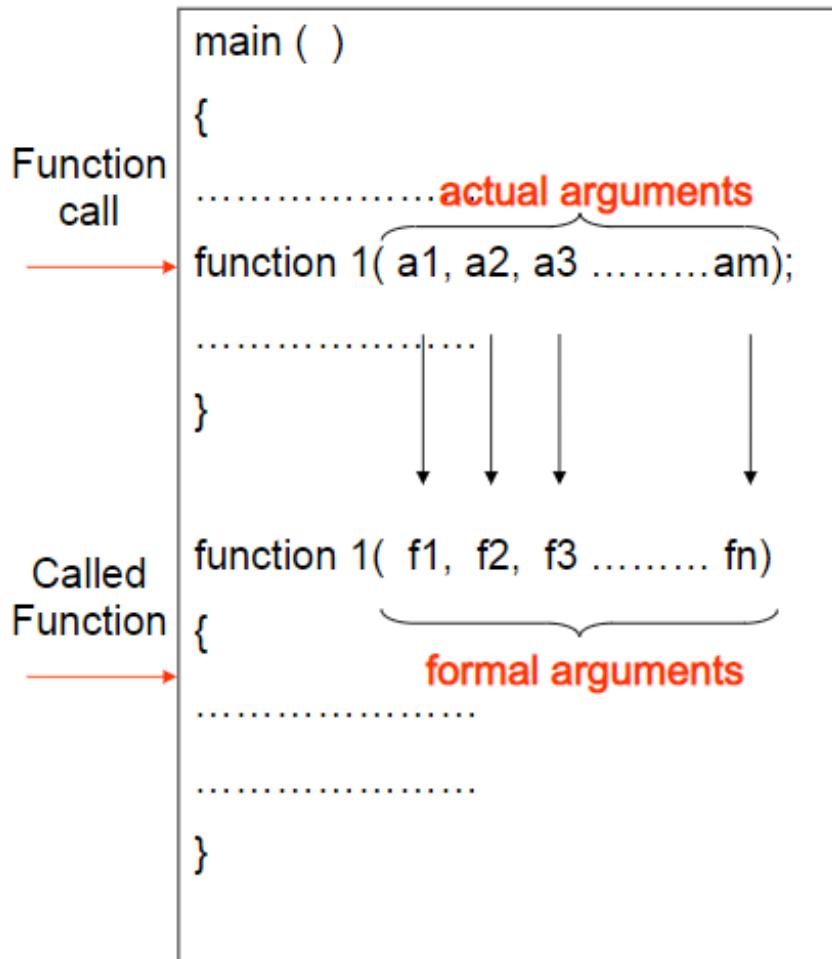
# Arguments but No Return values

- The values 500, 0,12 and 5 are the actual arguments, which become the values of the formal arguments inside the called function.
- **The actual and formal arguments should match in number, type, and order.** The values of actual arguments are assigned to the formal arguments on a one to one basis, starting with the first argument as shown in Fig. 9.6.

```
void printline(char ch)
void value(float p, float r, int n)
```

# Arguments but No Return values

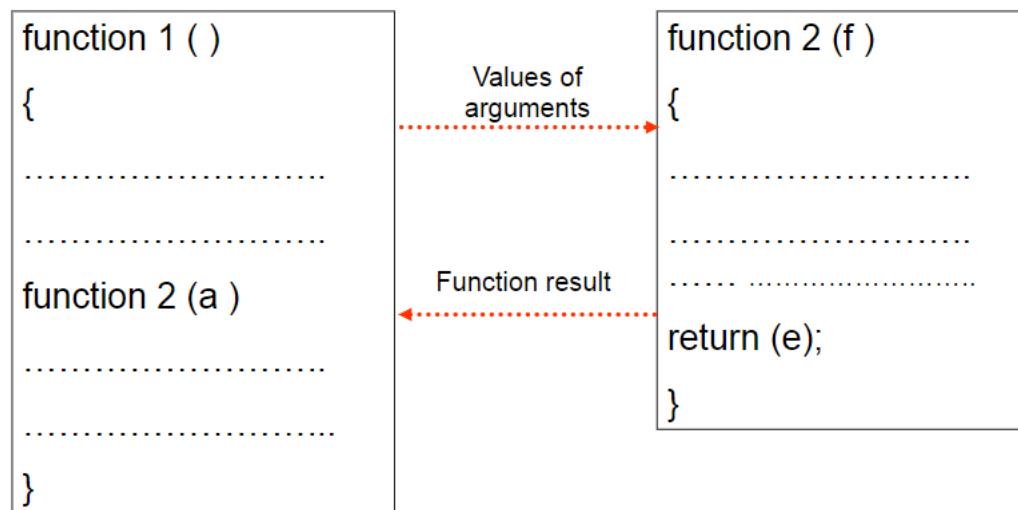
Arguments matching between the function call and the called function



1. The function call should have matching arguments.
2. If the actual arguments are more than the formal arguments, then the extra actual arguments are discarded.
3. If the actual arguments are less than the formal arguments, the unmatched formal arguments are initialized to some garbage values.
4. A mismatch in the data type will also result in passing garbage values.

# Arguments with Return values

- These shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program.
- A self-contained and independent function should behave like a 'black box' that receives a predefined form of input and outputs a desired value. Such functions will have data communication as shown in Fig. 9.8.



# Arguments with Return values

- The function value in Fig. receives data from the calling function through arguments, but does
- not send back any value. Rather, it displays the results of calculations at the terminal. However, we may not always wish to have the result of a function displayed. We may use it in the calling function for further processing.
- Moreover, to assure a high degree of portability between programs, a function should generally be coded without involving any I/O operations. For example, different programs may require different output formats for display of results.

# No Argument but Return Value

- There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function.
- A typical example is the `getchar` function declared in the header file `<stdio.h>`.
- We have used this function earlier in a number of places. The `getchar` function has no parameters but it returns an integer type data that represents a character.

# No Argument but Return Value

- We can design similar functions and use in our programs.  
Example:

```
int get_number(void);
main
{
    int m = get_number();
    printf ("%d",m);
}
int get_number(void)

{
    int number;
    scanf("%d",&number);
    return(number);
}
```

# Nesting of Functions

C permits nesting of two functions freely. There is no limit how deeply functions can be nested. Suppose a function a can call function b and function b can call function c and so on. Consider the following program:

```
#include<stdio.h>
float ratio(int,int,int);
int difference(int,int);

void main()
{
    int a,b,c;
    scanf("%d%d%d",&a,&b,&c);
    printf("%f\n",ratio(a,b,c));
}

float ratio(int x,int y,int z)
{
    if(difference(y,z))
        return(x/(y-z));
    else
        return(0.0);
}
```

```
difference(int p,int q)
{
    if(p!=q)
        return(1);
    else
        return(0);
}
```

# Nesting of Functions

the above program calculates the ratio  $a/b-c$ ;  
and prints the result. We have the following three functions:

```
main()
ratio()
difference()
```

main reads the value of a,b,c and calls the function ratio to calculate the value  $a/b-c$ ) this ratio cannot be evaluated if( $b-c$ ) is zero. Therefore ratio calls another function difference to test whether the difference( $b-c$ ) is zero or not

# Recursion

- When a called function in turn calls another function a process of 'chaining' occurs. Recursion is a special case of this process where a function calls itself. A very simple example of recursion is presented below:

```
main( )  
{  
printf("This is an example of recursion\n")  
main( );  
}
```

- When executed, this program will produce an output something like this:

```
This is an example of recursion  
This is an example of recursion  
.....
```

# Recursion

- Another useful example of recursion is the evaluation of factorials of a given number. The factorial of a number n is expressed as a series of repetitive multiplications as shown below:

$$\text{factorial of } n = n(n-1)(n-2) \dots \dots 1$$

- For example.

$$\text{factorial of } 4 = 4 \times 3 \times 2 \times 1 = 24$$

- A function to evaluate factorial of n is as follows:

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

# Recursion

- Let us see how the recursion works. Assume  $n = 3$ . Since the value of  $n$  is not 1, the statement

fact = n \* factorial (n-1);

- will be executed with  $n = 3$ . That is,

fact = 3 \* factorial (2);

- will be evaluated. The expression on the right-hand side includes a call to factorial with  $n = 2$ . This call will return the following value:

2 \* factorial(1)

- Once again, factorial is called with  $n = 1$ . This time, the function returns 1. The sequence of operations can be summarized as follows:

$$\begin{aligned} \text{fact} &= 3 * \text{factorial}(2) \\ &= 3 * 2 * \text{factorial}(1) \\ &= 3 * 2 * 1 \\ &= 6 \end{aligned}$$

# Recursion

- Recursive functions can be effectively used to solve problems where solution is expressed in terms of successively applying the same solution to subsets of the problem.
- When we write recursive functions. we must have an If statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

# Passing Arrays to a function

- » Like the values of simple variables, it is also possible to pass the values of an array to a function.
- » To pass a one-dimensional array to a called function, it is sufficient to list the name of the array, without any subscripts, and the size of the array as arguments. For example, the call

largest(a,n)

- » will pass the whole array a to the called function. The called function expecting this call must be appropriately defined. The largest function header might look like:  
  
`float largest(float array[ ], int size)`
- » The function largest is defined to take two arguments, the array name and the size of the array to specify the number of elements in the array.

# Passing Arrays to a function

- The declaration of the formal argument array is made as follows:

float array[ ];

- The pair of brackets informs the compiler that the argument array is an array of numbers. it is not necessary to specify the size of the array here. Let us consider a problem of finding the largest value in an array of elements.

# Passing Arrays to a function

- In C. the name of the array represents the address of its first element.
- By passing the array name, we are, in fact, passing the address of the array to the called function.
- The array in the called function now refers to the same array stored in the memory.
- Therefore, any changes in the array in the called function will be reflected in the original array.

# Rules to pass an Array to a function

- The function must be called by passing only the name of the array.
- In the function definition, the formal parameter must be an array type., the size of the array does not need to be specified.
- The function prototype must show that the argument is an array.

# Passing strings to a function

- The strings are treated as character arrays in C and therefore the rules for passing strings to functions are very similar to those for passing arrays to functions.
- Basic rules are:
- The string to be passed must be declared as a formal argument of the function when it is defined. Example:

```
void display(char item_name[ ])
{
}
```

- The function prototype must show that the argument is a string. For the above function definition, the prototype can be written as

```
void display(char str[ ]);
```

# Passing strings to a function

- A call to the function must have a string array name without subscripts as its actual argument. Example:

```
display (names);
```

- where names is a properly declared string array in the calling function. We must note here that, like arrays, strings in C cannot be passed by value to functions.

# Pass by value versus pass by pointer

- The technique used to pass data from one function to another is known as parameter passing. Parameter passing can be done in two ways:
  - **Pass by value (also known as call by value).**
  - **Pass by Pointers (also known as call by pointers).**
- In **pass by value**, values of actual parameters are copied to the variables in the parameter list of the called function. The called function works on the copy and not on the original values of the actual parameters. This ensures that the original data in the calling function cannot be changed accidentally.

# Pass by value versus pass by pointer

- **In pass by pointers** (also known as pass by address), the memory addresses of the variables rather than the copies of values are sent to the called function. In this case, the called function directly works on the data in the calling function and the changed values are available in the calling function for its use.
- *Pass by pointers method is often used when manipulating arrays and strings. This method is also used when we require multiple values to be returned by the called function.*

# The Scope, Visibility and Lifetime variables

- In C not only do all variables have a data type, they also have a storage class. The following variable storage classes are most relevant to functions:
  - **Automatic variables.**
  - **External variables.**
  - **Static variables**
  - **Register variables.**

# The Scope, Visibility and Lifetime variables

- The **scope of variable** determines over what region of the program a variable is actually available for use ('active').
- **Longevity** refers to the period during which a variable retains a given value during execution of a program ('alive'). So longevity has a direct effect on the utility of a given variable.
- The **visibility** refers to the accessibility of a variable from the memory.
- The variables may also be broadly categorized, depending on the place of their declaration, as **internal (local)** or **external (global)**. *Internal variables are those which are declared within a particular function, while external variables are declared outside of any function.*

# The Scope, Visibility and Lifetime variables

- It is very important to understand the concept of storage classes and their utility in order to develop efficient multifunction programs.

# Automatic Variables

- Automatic variables are declared inside a function in which they are to be utilized.
- They are created when the function is called and destroyed automatically when the function is exited, hence the named automatic.
- *Automatic variables are therefore private (or local ) to the function in which they are declared.*
- Because of this property, automatic variables are also referred to as **local or internal variables**.
- A variable declared inside a function without storage class specification is, by default, an automatic variable. For instance, the storage class of the variable number in the example below is automatic.

# Automatic Variables

```
main( )
{
    int number;
    -----
    -----
}
```

- We may also use the keyword auto to declare automatic variables explicitly.

```
main( )
{
    auto int number;
    -----
    -----
}
```

## Automatic Variables

- *One important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other function in the program.*
- This assures that we may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

## Extern Variables

- **Variables that are both alive and active throughout the entire program are known as external variables.** They are also known as **global variables**.
- Unlike local variables, global variables can be accessed by any function in the program.
- External variables are declared outside a function. For example, the external declaration of integer number and float length might appear as:

# Extern Variables

```
int number;
float length = 7.5;
main( )
{
    -----
    -----
}

function1( )
{
    -----
    -----
}

function2( )
{
    -----
    -----
}
```

# Extern Variables

- The variables number and length are available for use in all the three functions.
- In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared. Consider the following example:

# Extern Variables

- Consider the following example:

```
int count;
main( )
{
    count = 10;
    -----
    -----
}

function( )
{
    int count = 0;
    -----
    -----
    count = count+1;
}
```

## Extern Variables

- When the function references the variable count, it will be referencing only its local variable, not the global one. The value of count in main will not be affected.

# Global variable as parameters

- Since all functions in a program source file can access global variables, they can be used for passing values between the functions. However, **using global variables as parameters for passing values poses certain problems.**
- *The values of global variables which are sent to the called function may be changed inadvertently by the called function*
- *functions are supposed to be independent and isolated modules. This character is lost, if they use global variables*
- *It is not immediately apparent to the reader which values are being sent to the called function.*
- *A function that uses global variables suffers from reusability.*

## Global variable as parameters

- One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main()
```

```
{
```

```
y=5;
```

```
.....
```

```
.....
```

```
}
```

```
int y; /* global declaration */
```

```
func1()
```

```
{
```

```
    y = y+1;
```

```
}
```

## Global variable as parameters

- We have a problem here. As far as main is concerned, y is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default.

- The statement

`y = y+1;`

- in fun1 will, therefore, assign 1 to y.

# Static Variables

- As the name suggests, the value of static variables persists until the end of the program. A variable can be declared static using the keyword **static** like

```
static int x;  
static float y;
```

- A static variable may be either an internal type or an external type depending on the place of declaration.*

# Static Variables

- Internal static variables are those which are declared inside a function. The scope of internal static variables extend up to the end of the function in which they are defined.
- Therefore, internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program. Therefore, internal static variables can be used to retain values between function calls.
- For example, it can be used to count the number of calls made to a function.
- A static variable is initialized only once, when the program is compiled. It is never initialized again.

# Static Variables

- An external static variable is declared outside of all functions and is available to all the functions in that program.
- The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.
- It is also possible to control the scope of a function. For example, we would like a particular function accessible only to the functions in the file in which it is defined, and not to any function in other files. This can be accomplished by defining 'that' function with the storage class static.

# Register Variables

- We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access, keeping the frequently accessed variables (e.g., loop control variables) in the register will lead to faster execution of programs. This is done as follows:
- Although, ANSI standard specifies `register int count;` application to any particular data type, most compilers allow only int or char variables to be placed in the register.

# Register Variables

- Since only a few variables can be placed in the register, it is important to carefully select the variables for this purpose. However, C will automatically convert register variables into non-register variables once the limit is reached.

# Summary

**Table 9.1 Scope and Lifetime of Variables**

<i>Storage Class</i>	<i>Where declared</i>	<i>Visibility (Active)</i>	<i>Lifetime (Alive)</i>
None	Before all functions in a file (may be initialized)	Entire file plus other files where variable is declared with <b>extern</b>	Entire program (Global)
<b>extern</b>	Before all functions in a file (cannot be initialized) <b>extern</b> and the file where originally declared as global.	Entire file plus other files where variable is declared	Global
<b>static</b>	Before all functions in a file	Only in that file	Global
None or <b>auto</b>	Inside a function (or a block)	Only in that function or block	Until end of function or block
<b>register</b>	Inside a function or block	Only in that function or block	Until end of function or block
<b>static</b>	Inside a function	Only in that function	Global

## Scope Rules

- **Scope:** The region of a program in which a variable is available for use. The program's ability to access a variable from the memory.
- **Lifetime:** The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

## Rules of Use

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The lifetime (or longevity) of an auto variable declared in main is the entire program execution time, although its scope is only the main function.
5. The life of an auto variable declared in a function ends when the function is exited.
6. A static local variable, although its scope is limited to its function, its lifetime extends till the end on program execution.

## Rules of Use

7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is redeclared within its scope again, it loses its visibility in the scope of the redeclared variable.

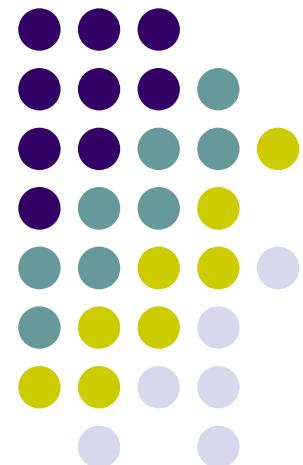
# **Thank You**

# Introduction to C Programming

---

## Chapter 9

## Pointers



**References**

[Programiz.com](https://www.programiz.com)

## Address in C

If you have a variable `var` in your program, `&var` will give you its address in the memory.

We have used address numerous times while using the `scanf()` function.

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

Here, the value entered by the user is stored in the address of `var` variable. Let's take a working example.

```
#include <stdio.h>
int main()
{
    int var = 5;
    printf("var: %d\n", var);

    // Notice the use of & before var
    printf("address of var: %p", &var);
    return 0;
}
```

## Output

```
var: 5
address of var: 2686778
```

# C Pointers

Pointers (pointer variables) are special variables that are used to store addresses rather than values.

## Pointer Syntax

Here is how we can declare pointers.

```
int* p;
```

Here, we have declared a pointer `p` of `int` type.

You can also declare pointers in these ways.

```
int *p1;  
int * p2;
```

Let's take another example of declaring pointers.

```
int* p1, p2;
```

Here, we have declared a pointer `p1` and a normal variable `p2`.

## Assigning addresses to Pointers

Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;
```

Here, 5 is assigned to the `c` variable. And, the address of `c` is assigned to the `pc` pointer.

## Get Value of Thing Pointed by Pointers

To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;  
c = 5;  
pc = &c;  
printf("%d", *pc); // Output: 5
```

Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

**Note:** In the above example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c;`

By the way, `*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

## Changing Value Pointed by Pointers

Let's take an example.

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c);    // Output: 1  
printf("%d", *pc); // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed the value of `c` to 1. Since `pc` and the address of `c` is the same, `*pc` gives us 1.

Let's take another example.

```
int* pc, c;  
c = 5;  
pc = &c;  
*pc = 1;  
printf("%d", *pc); // Output: 1  
printf("%d", c); // Output: 1
```

We have assigned the address of `c` to the `pc` pointer.

Then, we changed `*pc` to 1 using `*pc = 1;`. Since `pc` and the address of `c` is the same, `c` will be equal to 1.

Let's take one more example.

```
int* pc, c, d;  
c = 5;  
d = -15;  
  
pc = &c; printf("%d", *pc); // Output: 5  
pc = &d; printf("%d", *pc); // Output: -15
```

Initially, the address of `c` is assigned to the `pc` pointer using `pc = &c;`. Since `c` is 5, `*pc` gives us 5.

Then, the address of `d` is assigned to the `pc` pointer using `pc = &d;`. Since `d` is -15, `*pc` gives us -15.

## Example: Working of Pointers

Let's take a working example.

```
#include <stdio.h>
int main()
{
    int* pc, c;

    c = 22;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 22

    pc = &c;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 22

    c = 11;
    printf("Address of pointer pc: %p\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc); // 11

    *pc = 2;
    printf("Address of c: %p\n", &c);
    printf("Value of c: %d\n\n", c); // 2
    return 0;
}
```

### Output

```
Address of c: 2686784
Value of c: 22
```

```
Address of pointer pc: 2686784
Content of pointer pc: 22
```

```
Address of pointer pc: 2686784
Content of pointer pc: 11
```

```
Address of c: 2686784
Value of c: 2
```

## Common mistakes when working with pointers

Suppose, you want pointer `pc` to point to the address of `c`. Then,

```
int c, *pc;

// pc is address but c is not
pc = c; // Error

// &c is address but *pc is not
*pc = &c; // Error

// both &c and pc are addresses
pc = &c; // Not an error

// both c and *pc are values
*pc = c; // Not an error
```

# Relationship Between Arrays and Pointers

An array is a block of sequential data. Let's write a program to print addresses of array elements.

```
#include <stdio.h>
int main() {
    int x[4];
    int i;

    for(i = 0; i < 4; ++i) {
        printf("&x[%d] = %p\n", i, &x[i]);
    }

    printf("Address of array x: %p", x);

    return 0;
}
```

## Output

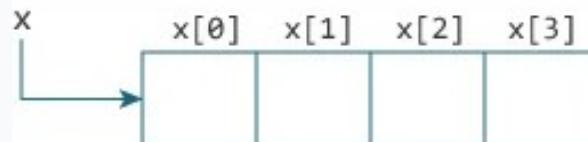
```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

## Output

```
&x[0] = 1450734448  
&x[1] = 1450734452  
&x[2] = 1450734456  
&x[3] = 1450734460  
Address of array x: 1450734448
```

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.

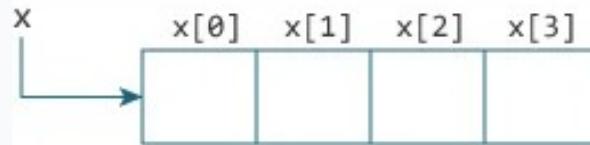


Relation between Arrays and Pointers

There is a difference of 4 bytes between two consecutive elements of array `x`. It is because the size of `int` is 4 bytes (on our compiler).

Notice that, the address of `&x[0]` and `x` is the same. It's because the variable name `x` points to the first element of the array.

### Output



Relation between Arrays and Pointers

```
&x[0] = 1450734448  
&x[1] = 1450734452  
&x[2] = 1450734456  
&x[3] = 1450734460  
Address of array x: 1450734448
```

From the above example, it is clear that `&x[0]` is equivalent to `x`. And, `x[0]` is equivalent to `*x`.

Similarly,

- `&x[1]` is equivalent to `x+1` and `x[1]` is equivalent to `*(x+1)`.
- `&x[2]` is equivalent to `x+2` and `x[2]` is equivalent to `*(x+2)`.
- ...
- Basically, `&x[i]` is equivalent to `x+i` and `x[i]` is equivalent to `*(x+i)`.

## Example 1: Pointers and Arrays

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

    int i, x[6], sum = 0;

    printf("Enter 6 numbers: ");

    for(i = 0; i < 6; ++i) {
        // Equivalent to scanf("%d", &x[i]);
        scanf("%d", x+i);

        // Equivalent to sum += x[i]
        sum += *(x+i);
    }

    printf("Sum = %d", sum);

    return 0;
}
```

```
Enter 6 numbers: 2
3
4
4
12
4
Sum = 29
```

Here, we have declared an array `x` of 6 elements. To access elements of the array, we have used pointers.

## Example 2: Arrays and Pointers

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

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

    // ptr is assigned the address of the third element
    ptr = &x[2];

    printf("*ptr = %d \n", *ptr);    // 3
    printf("*(ptr+1) = %d \n", *(ptr+1)); // 4
    printf("*(ptr-1) = %d", *(ptr-1)); // 2

    return 0;
}
```

## Example: Pass Addresses to Functions

```
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;

    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

## Example 2: Passing Pointers to Functions

```
#include <stdio.h>

void addOne(int* ptr) {
    (*ptr)++; // adding 1 to *ptr
}

int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);

    printf("%d", *p); // 11
    return 0;
}
```

# C Dynamic Memory Allocation

In this tutorial, you'll learn to dynamically allocate memory in your C program using standard library functions: `malloc()`, `calloc()`, `free()` and `realloc()`.

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

## C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a **pointer** of `void` which can be casted into pointers of any form.

### Syntax of malloc()

```
ptr = (castType*) malloc(size);
```

### Example

```
ptr = (float*) malloc(100 * sizeof(float));
```

The above statement allocates 400 bytes of memory. It's because the size of `float` is 4 bytes. And, the pointer `ptr` holds the address of the first byte in the allocated memory.

The expression results in a `NULL` pointer if the memory cannot be allocated.

## C calloc()

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

---

### Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

#### Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The above statement allocates contiguous space in memory for 25 elements of type `float`.

## C free()

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

---

### Syntax of free()

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by `ptr`.

## Example 1: malloc() and free()

```
#include <stdio.h>

int main() {
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) malloc(n * sizeof(int));

    // if memory cannot be allocated
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);

    // deallocated the memory
    free(ptr);

    return 0;
}
```

## Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

## Example 2: calloc() and free()

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &n);

    ptr = (int*) calloc(n, sizeof(int));
    if(ptr == NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements: ");
    for(i = 0; i < n; ++i) {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

### Output

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

## C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

---

### Syntax of realloc()

```
ptr = realloc(ptr, x);
```

Here, `ptr` is reallocated with a new size `x`.

### Example 3: realloc()

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

int main() {
    int *ptr, i , n1, n2;
    printf("Enter size: ");
    scanf("%d", &n1);

    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Addresses of previously allocated memory:\n");
    for(i = 0; i < n1; ++i)
        printf("%pc\n",ptr + i);

    printf("\nEnter the new size: ");
    scanf("%d", &n2);

    // rellocating the memory
    ptr = realloc(ptr, n2 * sizeof(int));

    printf("Addresses of newly allocated memory:\n");
    for(i = 0; i < n2; ++i)
        printf("%pc\n", ptr + i);

    free(ptr);

    return 0;
}
```

### Output

```
Enter size: 2
Addresses of previously allocated memory:
26855472
26855476

Enter the new size: 4
Addresses of newly allocated memory:
26855472
26855476
26855480
26855484
```

# **Thank You**