UNIT I

Historical Development of C:
C was an offspring of the 'Basic Combined Programming Language' (BCPL) called B, developed in 1960s at Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell Laboratories in 1972. The new language was named C. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system was developed at Bell Laboratories and was coded almost entirely in C.
C was used mainly in academic environments for many years, but eventually with the release of C compiler for commercial use and the increasing popularity of UNIX, it began to gain widespread support among compiler professionals. Today, C is running under a number of operating systems including Ms-DOS. C was now standardized by American National Standard Institute. Such type of C was named ANSI C.

IMPORTANCE OF C

- It is a robust language whose rich set of built-in functions and operations can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of high-level language. Both suitable for writing system software and business packages.
- There are only 32 keywords in C and its strength lies in built-in functions.
- C is highly portable. This means that C program written for one computer can run on another with little or no modification.
- C language is well suited for structured programming, thus requiring the user to think of a problem in terms of functions or blocks. This modular structure makes program debugging, testing and maintenance easier.
- Ability to extend itself. We can continuously add our own functions to C library.

Importance of C:
- Now-a-days, the popularity of C is increasing probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.
- Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC (Beginners All Purpose Symbolic Instruction Code – a high level programming language).
- There are only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs. C is highly portable. This means that C programs written for one computer can be seen on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.
- C Language is well suited for structure programming thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would
make a complete program. This modular structure makes program debugging, testing and maintenance.

- Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own function to the C library. With the availability of a large number of functions, the programming task becomes simple.

**Basic Structure of C programs:**
Every C program consists one or more modules called function. One of the function must be called main( ). A function is a sub-routine that may include one or more statements designed to perform a specific task. A C program may contain one or more sections shown in fig:

<table>
<thead>
<tr>
<th>Documentation Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Link Section</td>
</tr>
<tr>
<td>Global Declaration Section</td>
</tr>
<tr>
<td>Main() Function Section</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>Declaration Part</td>
</tr>
<tr>
<td>Execution Part</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Subprogram Section</td>
</tr>
<tr>
<td>Function 1</td>
</tr>
<tr>
<td>Function 2</td>
</tr>
<tr>
<td>........................................</td>
</tr>
<tr>
<td>Function n</td>
</tr>
<tr>
<td>(User-defined Functions)</td>
</tr>
</tbody>
</table>

Fig : Basic Structure of a C program
Every C program must have one main( ) function section. This section consists two parts: declaration part and executable part. The declaration part declares all the variables used in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening braces and ends at the closing brace. The closing brace of the main( ) function section is the logical end of the program. All the statements in the declaration and executable parts ends with a semicolon. The subprogram section contains all the user-defined functions that are called in the main( ) function. User-defined functions are generally placed immediately after the main( ) function, although they may appear in any order. All section, except the main( ) function section may be absent when they are not required.

**INTRODUCTION**

- **Data:** Data consisting of numbers, characters and strings arranged according to specific rules called **Syntax** of specific **Programming Language**.
- **Information:** The output of processed data is called information.
- **Program**: The task of processing data is accomplished by sequence of precise instructions called **program**.
- **Syntax**: The precise instructions in program are formed using certain symbols and words according to some rigid rules known as **syntax rule (or grammar)**.

**CHARACTER SET**

The character that can be used to form words, numbers and expressions depend upon the computer on which the program is run. The character set of C are:

1. **Letters**
2. **Digits**
3. **Special characters**
4. **White spaces**

White spaces may used to separate words, but are **prohibited** between the characters of keywords and identifiers.

<table>
<thead>
<tr>
<th>C Character Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Letters</strong></td>
</tr>
<tr>
<td>uppercase A.....Z</td>
</tr>
<tr>
<td>lowercase a.....z</td>
</tr>
</tbody>
</table>

| **Digits** |
| All decimal digits 0 ..... 9 |

| **Special Characters** |
| & ampersand |
| ^ caret |
| * asterisk |
| - minus sign |
| + plus sign |
| ( or less than sign |
| ) greater than sign |
| < opening angle bracket |
| > closing angle bracket |
| ( left parenthesis |
| ) right parenthesis |
| [ left bracket |
| ] right bracket |
| { left brace |
| } right brace |
| # number sign |

| **White Spaces** |
| Blank space |
| Horizontal tab |
| Carriage return |
| New line |
| Form feed |

**C TOKENS**

The smallest individual units are called **tokens**. C has 6 types of tokens

<table>
<thead>
<tr>
<th>C TOKENS</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEYWORDS</td>
</tr>
<tr>
<td>CONSTANTS</td>
</tr>
<tr>
<td>STRINGS</td>
</tr>
<tr>
<td>OPERATORS</td>
</tr>
<tr>
<td>IDENTIFIERS</td>
</tr>
<tr>
<td>SPECIALSYMB</td>
</tr>
</tbody>
</table>

- **KEYWORDS**
  - Float
  - While
- **CONSTANTS**
  - -15.5
  - 100
- **STRINGS**
  - "ABC"
  - "12year"
- **OPERATORS**
  - + -
  - * /
- **IDENTIFIERS**
  - Main
  - Amount, number
- **SPECIALSYMB**
  - \[
  - \]
Keywords serve as basic building block of programming statements.
Keywords must be written in lower case.

<table>
<thead>
<tr>
<th>KEYWORDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>long</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>case</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>register</td>
</tr>
<tr>
<td>typedef</td>
</tr>
<tr>
<td>char</td>
</tr>
<tr>
<td>extern</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>union</td>
</tr>
<tr>
<td>continue</td>
</tr>
<tr>
<td>for</td>
</tr>
<tr>
<td>signed</td>
</tr>
<tr>
<td>void</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>default</td>
</tr>
<tr>
<td>goto</td>
</tr>
<tr>
<td>sizeof</td>
</tr>
<tr>
<td>volatile</td>
</tr>
<tr>
<td>do</td>
</tr>
<tr>
<td>if</td>
</tr>
<tr>
<td>static</td>
</tr>
<tr>
<td>while</td>
</tr>
</tbody>
</table>

IDENTIFIERS
Identifiers refer to the names of variables, functions and arrays. These are user-defined names and consist of a sequence of letters are permitted, although lowercase letters are commonly used. The underscore character ( _) is also permitted.

Rules for Identifiers
1. First character must be an alphabet
2. Must consist of only letters, digits or underscore.
3. Only first 31 characters are significant.
4. Cannot use a keyword as Identifier.
5. Must not contain white space

CONSTANTS
Constants in C are fixed values that do not change during the execution of a program.
There are mainly three types of constants namely: integer, real and character constants.

**Integer Constants:**

The integer constants

- **Whole Numbers**
  - Eg. 25, 35, -25, -46
  - Computer allocates only 2 bytes in memory.
  - 16th bit is sign bit. (if 0 then +ve value, if 1 then -ve value)

\[
\begin{align*}
1 & \quad 1 \\
1 & \quad 1 \\
2^0 & \quad 2^1 \\
2^1 & \quad 2^2 \\
2^2 & \quad 2^3 \\
2^3 & \quad 2^4 \\
2^4 & \quad 2^5 \\
2^5 & \quad 2^6 \\
2^6 & \quad 2^7 \\
2^7 & \quad 2^8 \\
2^8 & \quad 2^9 \\
2^9 & \quad 2^{10} \\
2^{10} & \quad 2^{11} \\
2^{11} & \quad 2^{12} \\
2^{12} & \quad 2^{13}
\end{align*}
\]

\[
= 1*1 + 4*1 + 8*1 + 16*1 + 32*1 + 64*1 + 128*1 + 256*1 + 512*1 + 1024*1 + 2048*1 + 4096*1 + 8192*1 + 16384*1
\]

\[
= 32767 \text{ (32767 Bits can be stored for integer constants)}
\]

- 32768 is negative
- -32767 is minimum

(i) **Decimal Integer Constant:**

- 0 to 9
- E.g. 49, 58, .62, ... (40000 cannot come bcoz it is > 32767)

(ii) **Octal Integer Constant:**

- 0 to 7
- Add "0" before the value.
- E.g.: 045, 056, 067

(iii) **Hexadecimal Integer:**

- 0 to 9 and A to F
- Add 0x before the value
- E.g: 0x42, 0x56, 0x67
REAL CONSTANTS:
The real or floating point constants are in two forms namely fractional form and the exponential form.

A real constant in fractional form must:
- Have at least one digit
- It must have a decimal point
- Could have positive or negative sign (default sign is positive)
- Must not have commas or spaces within it
- Allots 4 bytes in memory

Ex: +867.9, -26.9876, 654.0

In exponential form, the real constant is represented as two parts. The part lying before the ‘e’ is the ‘mantissa’, and the one following ‘e’ is the ‘exponent’.

The real constant in exponential form must follow the following rules:
- The mantissa part and the exponential part should be separated by the letter ‘e’
- The mantissa may have a positive or negative sign (default sign is positive)
- The exponent must have at least one digit
- The exponent must be a positive or negative integer (default sign is positive)
- The range of real constants in exponential form is 3.1e38 and 3.1e-38

Ex: +3.2e-4, 4.1e8, -0.2e+4, -3.2e-4

CHARACTER CONSTANTS
A character constant is an alphabet, a single digit or a single special symbol enclosed within inverted commas. The maximum length of a character constant can be 1 character. Allots 1 byte of memory

Ex: 'B', 'l', '#'

String Constant
A string constant is a group of alphabets, digits or special symbols enclosed within double inverted commas.

Eg: "hai, welcome", "345fh"

Variables and Expressions

A variable declaration serves three purposes:

1. It defines the name of the variable.
2. It defines the type of the variable (integer, real, character, etc.).
3. It gives the programmer a description of the variable.

```
int answer;  /* the result of our expression */
```

1. The keyword int tells C that this variable contains an integer value.
2. The variable name is answer.
3. The semicolon (;) marks the end of the statement.
4. The comment is used to define this variable for the programmer.

1. You can store values in variables.
2. Each variable is identified by a variable name.
3. Each variable has a variable type.
4. Variable names start with a letter or underscore (_), followed by any number of letters, digits, or underscores.
5. Uppercase is different from lowercase, so the names sam, Sam, and SAM specify three different variables.
6. The variables are defined at the beginning of the block

Variable Declaration
Variables should be declared either outside a function or at the start of a block of code, after the opening { and before any other statements. They must be declared before use and the type must normally be specified

int miles, yards; /* global variables */
main()
{
float kilometres; /* local variables */

Variable Types
There are a number of 'built-in' data types in C. These are listed below. Where a shorter version of the type name exists, this is given in brackets; essentially the base type int is implicit whenever short, long, or unsigned are used.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range/Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>short int</td>
<td>-128 to 127 (1 byte)</td>
</tr>
<tr>
<td>unsigned short int</td>
<td>0 to 255 (1 byte)</td>
</tr>
<tr>
<td>char</td>
<td>0 to 255 or -128 to +127 (2 bytes)</td>
</tr>
<tr>
<td>unsigned char</td>
<td>0 to 255 (1 byte)</td>
</tr>
<tr>
<td>signed char</td>
<td>-128 to 127 (1 byte)</td>
</tr>
<tr>
<td>int</td>
<td>-32,768 to +32,767 (2 bytes)</td>
</tr>
<tr>
<td>unsigned int</td>
<td>0 to +65,535 (2 bytes)</td>
</tr>
<tr>
<td>long int</td>
<td>-2,147,483,648 to +2,147,483,647 (4 bytes)</td>
</tr>
<tr>
<td>unsigned long int</td>
<td>0 to 4,294,967,295 (4 bytes)</td>
</tr>
<tr>
<td>float</td>
<td>single precision floating point (4 bytes)</td>
</tr>
<tr>
<td>double</td>
<td>double precision floating point (8 bytes)</td>
</tr>
<tr>
<td>long double</td>
<td>extended precision floating point (10 bytes)</td>
</tr>
</tbody>
</table>

The ANSI C standard states only that the number of bytes used to store a long int is equal to or greater than the number used to store an int, and that the size of an int is at least as big as the size of a short int. The C operator, sizeof, reports the number of bytes used to store a variable or a type 3, and so the above can be rewritten as:

sizeof(long int), sizeof(int), sizeof(short int)

It may be the case, for your compiler and system, that the sizes for all three are the same. The same holds for the three floating point types:

sizeof(long double), sizeof(double), sizeof(float)

If you have an ANSI C compiler, the actual ranges for your system can be found in the header files limits.h for integer types and float.h for floating point types. ANSI C has the qualifier signed which can
be applied to any of the integer types (including char). signed is the default for all the int types, but is
dependent on the compiler for char.

/* Variable Names : check Identifiers */

Variables

- It is a data name that can be used to store a data value.
- Unlike constants, a variable may take different values in memory during execution.
- Variable names follow the naming convention for identifiers.
  - Examples :: temp, speed, name2, current

Declaration of Variables

- There are two purposes:
  1. It tells the compiler what the variable name is.
  2. It specifies what type of data the variable will hold.
- General syntax:
  data-type variable-list;
- Examples:
  int velocity, distance;
  int a, b, c, d;
  float temp;
  char flag, option;

OPERATORS AND EXPRESSIONS

An operator is a symbol that tells the computer to perform certain mathematical or logical
manipulations. C supports 8 types of operators
1. Arithmetic operators
2. Comparison operators/Relational operators
3. Logical operators
4. Assignment operators
5. Increment and decrement operators
6. Conditional operators
7. Bitwise operators
8. Special operators

1. Arithmetic Operators

- multiplication
- division
- remainder after division (modulo arithmetic)
- addition
- subtraction and unary minus

The / operator is used for two different operations: integer and floating point division. If both operands
of the divide operator are of integral (char, int and its derivatives) type then integer division is
performed. If either operand is float , double or long double then real division is undertaken.

Eg: int main()
{
  float a;
  a = 1 / 3;
  return 0;
}

would print 0.000000 as integer division was performed even though a is of type float. This type is
called Real Arithmatics.

2. Relational and Logical Operators
The following relational operators produce a true or false value. True and false, when generated by such an expression, are represented by the int values 1 and 0 respectively. Thus the expression
1 < 10
has the int value 1. Note, however, that C treats any non-zero value as being true!

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>2</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than OR equal</td>
<td>2</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than OR equal</td>
<td>2</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>1</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>1</td>
</tr>
</tbody>
</table>

Assignment Operators
C has many assignment operators. Pascal and Fortran have one.

- = assign
- += assign with add
- -= assign with subtract
- *= assign with multiply
- /= assign with divide
- %= assign with remainder
- >>= assign with right shift
- <<= assign with left shift
- &= assign with bitwise AND (see 3.8)
- ^= assign with bitwise XOR
- |= assign with bitwise OR

All but the first assignment operator are shorthand methods of modifying a variable or object. For example, the following two statements are equivalent:

```c
a = a + 17;
a += 17;
```

An example of a situation where the type and value of the assignment expression is not the same as the value and type of the right hand operand, consider the following:

```c
int main()
{
    double x = 1.23, y;
    int i;
    y = i = x;
    printf("%f\n", y);
    return 0;
}
```

Here the assignment expression i = x has the int value 1, the double variable y has the value 1.0 after the assignment, not 1.23.

Increment and Decrement Operators
Increment and decrement operators give a shorthand method of adding/subtracting 1 from an object.

- ++ increment
- -- decrement
These operators can be prefix or postfix. An example of the different behaviour of the prefix and postfix forms is given below, but essentially with the prefix form the variable is changed before the value of the expression in which it appears is evaluated, and with the postfix form the variable is modified afterwards.

\[
\begin{align*}
  b &= 3; \\
  a &= b++ + 6; \quad /* a = 9, b = 4 */ \\
  b &= 3; \\
  a &= ++b + 6; \quad /* a = 10, b = 4 */
\end{align*}
\]

**Conditional Operator**

\[
\text{expr1 ? expr2 : expr3}
\]

Evaluate expr1
- if non-zero (true) evaluate expr2
- if zero (false) evaluate expr3

The value of the whole expression is the value of whichever of expr2 or expr3 was evaluated.

\[
\text{double maximum(double a, double b)}
\]

\{
  \text{return (a > b) ? a : b;}
\}

**Bitwise Operators**

Bitwise operators allow manipulation of the actual bits held in each byte of a variable. Each byte consists of a sequence of 8 bits, each of which can store the value 0 or 1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>one's complement</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift (binary multiply by 2)</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift (binary divide by 2)</td>
</tr>
</tbody>
</table>

**Example:**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>0010111</td>
<td>1101000</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>0010111</td>
<td>0101110</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>0010111</td>
<td>0001011</td>
</tr>
</tbody>
</table>

**AND**

<table>
<thead>
<tr>
<th>AND</th>
<th>XOR</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &amp; 0 = 0</td>
<td>0 ^ 0 = 0</td>
<td>0</td>
</tr>
<tr>
<td>1 &amp; 0 = 0</td>
<td>1 ^ 0 = 1</td>
<td>1</td>
</tr>
<tr>
<td>0 &amp; 1 = 0</td>
<td>0 ^ 1 = 1</td>
<td>0</td>
</tr>
<tr>
<td>1 &amp; 1 = 1</td>
<td>1 ^ 1 = 0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Special Operators**

1. **Comma operator**

   Used to link related expression together.

   **Example:**

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

   First assign value 10 to x and 5 to y, then calculate x+y and assign to variable `value`

   **In For Loops:**
For \( n=1, m=10, n<=m; n++, m++ \)

**In While Loop:**

\[
\text{while } (c = \text{getchar()}, c != '10')
\]

2. **The sizeof operator**

The unary operator `sizeof` is used to calculate the sizes of datatypes, in number of bytes. `sizeof` returns the size of the type of the variable or parenthesized. `sizeof` can be applied to all datatypes, be they primitive types such as the integer and floating-point types defined in the language, pointers to memory addresses, or the compound datatypes (unions, structs, or C++ classes) defined by the programmer.

Example:

```c
/*pointer to type int, used to reference our allocated data*/
int * pointer = malloc(sizeof(int) * 10);
m = sizeof(sum);
n = sizeof(long int);
```

**Type conversion**

1. **Implicit type conversion**

C automatically converts any intermediate values to the proper types so that the expression can be evaluated without losing any significance. This automatic conversion is called as implicit type conversion.

Rules for implicit conversion:-

1. All `short` and `char` are automatically converted to `int`; then
2. if one of the operand is `long double`, the other will be converted to `long double` and the result will be `long double`.
3. else, if one of the operand is `double`, the other will be converted to `double` and the result will be `double`.
4. else, if one of the operand is `float`, the other will be converted to `float` and the result will be `float`.

2. **Explicit type conversion**

Explicit type conversion is a type conversion which is explicitly defined within a program (instead of being done by a compiler for implicit type conversion).

```c
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result = (int)da + (int)db + (int)dc; //result == 9
/* if implicit conversion would be used (as if result = da + db + dc), result would be equal to 10 */
```

**Cast Operators**

Cast operators allow the conversion of a value of one type to another.

`(float) sum;` converts type to float

`(int) fred;` converts type to int
MANAGING INPUT OUTPUT OPERATIONS

#include<stdio.h> is the header file which help to call the functions printf(), scanf() which are helps us to perform the i/o operations in C.

Reading a Character
Reading a single character can be done by using the function getchar().
Syntax :

    variable_name = getchar();

variable_name is a valid Cname 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 the getchar().
Example:

    char name;
    name = getchar();

Some functions supported in ctype.h, therefore should include this header file in the program.

<table>
<thead>
<tr>
<th>Function</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum(c)</td>
<td>Is c an alphanumeric character?</td>
</tr>
<tr>
<td>isalpha(c)</td>
<td>Is c an alphabetic character?</td>
</tr>
<tr>
<td>isdigit(c)</td>
<td>Is c a digit?</td>
</tr>
<tr>
<td>islower(c)</td>
<td>Is c lower case letter?</td>
</tr>
<tr>
<td>isprint(c)</td>
<td>Is c a printable character?</td>
</tr>
<tr>
<td>isspace(c)</td>
<td>Is c a space character?</td>
</tr>
<tr>
<td>isupper(c)</td>
<td>Is c an upper case letter?</td>
</tr>
</tbody>
</table>
Writing a Character

`putchar()` is used to writing characters one at a time to the terminal.

Syntax:
```
putchar(variable_name);
```

Eg:
```
char answer;
answer = 'Y';
putchar(answer);
putchar(\n);
```

Formatted Input — `scanf`

`scanf` is used to interpret characters input to the computer and to store the interpretation in the specified variable(s).

`scanf("%d", &x);` read a decimal integer from the keyboard and store the value in the memory address of the variable `x`. The `&` character is vital with arguments that are not pointers.

**scanf Control String Conversion Characters**

`scanf` uses the same conversion characters as `printf`. The arguments to `scanf` must be pointers (addresses), hence the need for the `&` character above.

**Character Form of output Expected argument type**

- `c` character pointer to char
- `d` decimal integer pointer to int
- `x` hexadecimal integer pointer to int
- `o` octal integer pointer to int
- `u` unsigned integer pointer to int
- `i` integer pointer to int
- `e` floating point number pointer to float
- `f` floating point number pointer to float
- `g` floating point number pointer to float
- `s` string pointer to char
- `p` address format, depends on system pointer to void

The conversion characters `e`, `f` and `g` have exactly the same effect, they read a floating point number which can be written with or without an exponent. When reading integers using the `i` conversion character, the data entered may be preceded by `0` or `0x` to indicate that the data is in octal (base 8) or hexadecimal (base 16).

The `d`, `x`, `o`, `u` or `i` conversion characters should be preceded by `l` if the argument is a pointer to a long rather than an a pointer to an int, or by `h` of the argument is a pointer to short int. The `e`, `f`, or `g` conversion characters should be preceded by `l` if the argument is pointer to double rather than pointer to float, or `L` for pointer to long double. Maximum field widths can be specified between the `%` and the conversion character. For example, the following call to `scanf` will read no more than 50 characters into the string variable `str`.

```
scanf("%50s",str);
```

A `*` between the `%` and the conversion character indicates that the field should be skipped.

Spaces and tabs in the format string are ignored. `scanf` also skips leading white space (spaces, tabs, newlines, carriage returns, vertical tabs and formfeeds) in the input (except when reading single characters). However a subsequent space terminates the string.
char a[50];
scanf("%s", &a); /* Input “Where is it” */
puts(a); /* Outputs “Where” */
To read the other strings add more conversion characters or use gets() 
scanf("%s%s%s", &a, &b, &c);

Other characters that appear in the format string must be matched exactly in the input. For example, the following scanf statement will pick apart a date into separate day, month and year values.
scanf("%d/%d/%d", &day, &month, &year);

It is also possible to use a scan set when reading input, the scan set lists the only characters that should be read (or should not be read) from the input stream for this conversion, for example:
scanf("%[a-zA-Z]c", &c); only read alphabetic characters
scanf("%[^0-9]c", &d); do not read digits

**Character I/O — getchar & putchar**

getchar and putchar are used for the input and output of single characters respectively. getchar() returns an int which is *either* EOF (indicating end-of-file) or the next character in the standard input stream. putchar(c) puts the character c on the standard output stream.

```c
int main()
{
    int c;
    c = getchar(); /* read a character and assign to c */
    putchar(c); /* print c on the screen */
    return 0;
}
```

**Formatted output**

The *printf* Function

1. printf displays information on screen.
2. printf returns the number of characters printed.
3. printf displays the text you put inside the double quotes.
4. printf requires the backslash character - an escape sequence - to display some special characters.
5. printf can display variables by using the % conversion character.
6. printf format: a string argument followed by any additional arguments.

```c
#include <stdio.h>

main()
{
    int i = 0;
```


```c
#include <stdio.h>

void main()
{
    int a;               /* simple integer type */
    long int b;          /* long integer type */
    short int c;         /* short integer type */
    unsigned int d;      /* unsigned integer type */
    char e;              /* character type */
    float f;             /* floating point type */
    double g;            /* double precision floating point */

    a = 1023;
    b = 2222;
    c = 123;
    d = 1234;
    e = 'X';
    f = 3.14159;
    g = 3.1415926535898;

    printf("a = %d\n",a);   /* decimal output */
    printf("a = %o\n",a);   /* octal output */
    printf("a = %x\n",a);   /* hexadecimal output */
    printf("b = %ld\n",b);  /* decimal long output */
    printf("c = %d\n",c);   /* decimal short output */
    printf("d = %u\n",d);   /* unsigned output */
    printf("e = %c\n",e);   /* character output */
    printf("f = %.1f\n",f);     /* floating output */
    printf("g = %.2f\n",g);  /* double float output */

    printf("a = %d\n",a);   /* simple int output */
    printf("a = %7d\n",a);  /* use a field width of 7 */
    printf("a = %-7d\n",a); /* left justify width = 7 */

    printf("f = %.2f\n",f); /* simple float output */
}
```

An example of formatted output was provided in Fundamental Programming Structures in C. The program is duplicated here for further study.

```c
i=printf("abcde\n");

printf("total characters printed %d\n",i);
}
```

OUTPUT:

```
abcde
total characters printed 6
```

```c
#include <stdio.h>

void main()
{
    int a;               /* simple integer type */
    long int b;          /* long integer type */
    short int c;         /* short integer type */
    unsigned int d;      /* unsigned integer type */
    char e;              /* character type */
    float f;             /* floating point type */
    double g;            /* double precision floating point */

    a = 1023;
    b = 2222;
    c = 123;
    d = 1234;
    e = 'X';
    f = 3.14159;
    g = 3.1415926535898;

    printf("a = %d\n",a);   /* decimal output */
    printf("a = %o\n",a);   /* octal output */
    printf("a = %x\n",a);   /* hexadecimal output */
    printf("b = %ld\n",b);  /* decimal long output */
    printf("c = %d\n",c);   /* decimal short output */
    printf("d = %u\n",d);   /* unsigned output */
    printf("e = %c\n",e);   /* character output */
    printf("f = %.1f\n",f);     /* floating output */
    printf("g = %.2f\n",g);  /* double float output */

    printf("a = %d\n",a);   /* simple int output */
    printf("a = %7d\n",a);  /* use a field width of 7 */
    printf("a = %-7d\n",a); /* left justify width = 7 */

    printf("f = %.2f\n",f); /* simple float output */
}
```
printf("f = %12f\n",f); /* use field width of 12 */
printf("f = %12.3f\n",f); /* use 3 decimal places */
printf("f = %12.5f\n",f); /* use 5 decimal places */
printf("f = %-12.5f\n",f); /* left justify in field */
}

UNIT II

Control Statements

The statements which alter the flow of execution of the program are known as control statements. In the absence of control statements, the instruction or statements are executed in the same order in which they appear in the program. Sometimes, we may want to execute some statements several times. Sometime we want to use a condition for executing only a part of program. So, control statements enable use to specify the order in which various instruction in the program are to be executed.

There are two types of control statements:
1. Loops: for, while, do-while
2. Decisions: if, if...else, nested if....else, switch

5.1 DECISION MAKING AND BRANCHING

Conditional Branching —

5.1.1 Simple if

In all of the following examples statement can be replaced by a compound statement (or block) which consists of several statements enclosed within braces.

```c
if (expression) /* if expression is true */

statement1; /* do statement1 */
```

![Flowchart of if control statement](image)

For eg:

```c
/* ### Program to check whether the number is -ve or +ve ### */
#include<stdio.h>
main( )
```
```c
{  
  int num ;  
  clrscr( ) ;  
  printf ("Enter a number to be tested:" ) ;  
  scanf ("%d", &num) ;  
  if (num<0)  
    printf("The number is negative") ;  
  printf ("value of num is : %d\n", num) ;  
  getch( ) ;  
}  
Output : 1st run  
Enter a number to be tested : -6  
The number is negative  
Value of num is : -6  
2nd run  
Enter a number to be tested : 6  
Value of num is : 6  

5.1.2 if….else

Syntax is :
if (condition)  
statement1 ;  
else  
statement2;  
else

Flowchart:

Here if the condition is true then statement1 is executed and if it is false then statement2 is executed. After this the control transfers to the next statement which is immediately after the if...else control statement.
/* Program to check whether the number is even or odd */  
#include<stdio.h>  
main( )  
{  
```
```c
int num, remainder;
clrscr();
printf("Enter a number:");
scanf("%d", &num);
remainder = num%2; /* modular division */
if (remainder == 0) /* test for even */
    printf("Number is even\n");
else
    printf(" Number is odd\n");
getch();
}
Output:
Enter a number : 15
Number is odd.
```

**else if statement:**

Nested if ...else statement:

We can have another if...else statement in the if block or the else block. This is called nested if...else statement. For example

```c
if(condition1)
{
    if(condition2)
        statementA1;
    else
        statement A2;
    Here, we have if...else inside both if block and else block
}
else
{
    if(condition3)
        statementB1;
    else
        statementB2;
}
```

/* Program to find whether a year is leap or not */
#include<stdio.h>
main()
{
    int year,
clrscr();
    printf("Enter year: ");
    scanf("%d", &year);
    if(year%100 == 0)
    {
        if(year%400 == 0)
```

```c
```
```c
printf("Leap year\n")
else
printf("Not leap year\n")
}
getch();
}

This can also be written in place of nested if else as
if ((year%4 = = 0 && year %100!=0) | | year%400 = = 0)
printf("%d is a leap year\n", year) ;
else
printf("%d is not a leap year\n", year) ;

/* Program to find largest number from three given number */
#include<stdio.h>
main( )
{
int a, b, c, large ;
crscr( ) ;
printf("Enter three numbers : ") ;
scanf("%d%d%d", &a, &b, &c) ;
if (a>b)
{
if (a>c)
large = a ;
else
large = c ;
}else
{
if (b>c)
large = b ;
else
large = c ;
}
printf("Largest number is %d\n", large) ;
getch();
} /* End of main( ) */

Output:
Enter the numbers: 3 4 5
Largest num is 5

5.1.3 Nesting of if...else statement

if-else statement can appear nested inside another if-else statement. When nesting statements in this way, it should be noted that an else always matches with the nearest unmatched if. Thus, in the example below, statement2 will be executed only when a < b and c <d.
if (a < b)
if (c < d)
  statement1;
else
  statement2;

If it is required to match an else with an earlier if than the nearest, then braces are required. For example:

if (a < b)
{
  if (c < d)
    statement1;
}
else
  statement2;

statement2 is now executed when a > b.

5.1.4 else-if Ladder

else if statement:
This is a type of nesting in which there is an if...else statement in every else part except the last else part. This type of nesting is frequently used in programs and is also known as else if ladder.

if(condition1) if(condition1)
  statementA ; statementA ;
else elseif(condition2)
  if(condition2) statementB ;
  statementB ; elseif(condition3)
  else statementC ;
if (condition3) else
  statementC ; statement D ;
else
  statement D ;
/* Program to find out the grade of a student when the marks of 4 subjects are given. The method of assuming grade is as 
per>=80 grade = A 
per<80 and per>=60 grade = B 
per<60 and per>=50 grade = C 
per<50 and per>=40 grade = D 
p grade = F 
Here Per is percentage */

#include<stdio.h> 
main() 
{" 
float m1, m2, m3, m4, total, per ; 
char grade ;
clrscr();
printf ("Enter marks of 4 subjects : ");
scanf ("%f%f%f%f",&m1, &m2, &m3, &m4); 
total = m1+m2+m3+m4 ;
per = total /4 ; 
if(per>=80) 
grade = 'A' ;
elseif(per>=60)
grade = 'B' ;
elseif(per>=50)
grade = 'C' ;
elseif(per>=40)
grade = 'D' ;
else
grade = 'F';
printf("Percentage is %f\n Grade is %c\n", per, grade);
getch();
}

Equivalent code in simple if statement:
if(per>=80)
grade = 'A' ;
if(per<80 && per>=60)
grade = 'B' ;
if(per<60 && per>=50)
grade = 'D' ;
if(per<40)
grade = 'F' ;

In else-if ladder whenever a condition is found true other conditions are not checked, while in if statement all the conditions will always be checked wasting a lot of time and moreover the conditions here are more lengthy.

5.1.5 Conditional Selection — switch

    switch ( expression)
    {
        case value : statement; statement; ... 
        case value : statement; statement; ...
        ..
        default : statement; statement; ...
    }
switch is a mechanism for jumping into a series of statements, the exact starting point depending on the value of the expression. In the example below, for example, if the value 3 is entered, then the program will print three two one something else!

```c
int main()
{
    int i;
    printf("Enter an integer: ");
    scanf("%d", &i);
    switch(i)
    {
        case 4: printf("four ");
        case 3: printf("three ");
        case 2: printf("two ");
        case 1: printf("one ");
        default: printf("something else!");
    }
    return 0;
}
```

This may not be what was intended. This process of executing statements in subsequent case clauses is called **fall through**. To prevent fall through, break statements can be used, which cause an immediate exit from the switch statement. In the example above, replacing

```c
case 4: printf("four ");
```

with

```c
case 4: printf("four "); break;
```

and adding break statements to the statements for the other labels, will result in a program that prints only one string depending on the value of the integer input by the user. The values listed in the case
part of the switch statement must be constant integer values; integer expressions can be used as long
as the value can be determined at compile time. The default label is optional, but if it is present acts as
a catch all clause. The labels can occur in any order; there is no need to have the default label last

/* Program to perform arithmetic calculation on integers */
#include<stdio.h>
main( )
{ char op ;
  int a, b ;
  clrscr( ) ;
  printf ("Enter a number, operators and another num :") ;
  scanf ("%d%c%d", &a, &op, &b) ;
  switch (op)
  {
    case '+' : printf ("Result = %d
", a+b ) ;
      break ;
    case '-' : printf ("Result = %d
", a-b ) ;
    case '*' : printf ("Result = %d
", a*b ) ;
    case '/' : printf ("Result = %d
", a/b ) ;
    case '%' : printf ("Result = %d
", a%b ) ;
      default: printf ("Enter your valid operation") ;
  } /* end of switch */
  getch() ;
} /* end of main( ) */

5.1.6 Conditional Expressions

The statements

if (a > b)
  
    z = a;

else
  
    z = b;

compute in z the maximum of a and b. The conditional expression, written with the ternary operator "?
: ", provides an alternate way to write this and similar constructions. In the expression

expr1 ? expr2 : expr3

depends on the expression expr1 is evaluated first. If it is non-zero (true), then the expression expr2 is evaluated,
and that is the value of the conditional expression. Otherwise expr3 is evaluated, and that is the value.
Only one of expr2 and expr3 is evaluated. Thus to set z to the maximum of a and b,

z = (a > b) ? a : b; /* z = max(a, b) */
It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If \( \text{expr2} \) and \( \text{expr3} \) are of different types, the type of the result is determined by the conversion rules discussed earlier in this chapter. For example, if \( f \) is a float and \( n \) an int, then the expression \( (n > 0) ? f : n \) is of type float regardless of whether \( n \) is positive.

### 5.2.7 goto

It is possible to jump to any statement within the same function using goto. A label is used to mark the destination of the jump. goto is rarely, if ever, needed, as if, switch, while and for should provide all the branching and iteration structures needed. However, there are times when a goto simplifies the code greatly. A frequently cited example is when something goes disastrously wrong deep within nested loops:

```c
void fred(void)
{
    while (...) 
    for (...) 
    if (disaster) goto error;
    error:
    tidy up the mess
}
```

### 5.2 Decision Making And Looping

#### 5.2.1 Introduction

In many situations we have to repeat the execution of a certain number of statements for a number of times. In such case loop can be applied. The loop continues while a condition set by the programmer is true. When the condition becomes false the loop breaks and the control passes to the statement following the loop.

There are three types of loop in 'C': for, while and do-while loop.

A loop consists of two parts: -

1. Control statement
2. Body of the loop.

Again control statement can be of two types

i) Entry control
ii) Exit control

In entry control the condition of the loop is tested by the control before the loop body is executed. And on each repetition of the loop this checking is done. So it may happen that the loop body is not executed for a single time if the condition is false from the beginning.

In case of exit control statement the condition is checked at the end of the loop body. So in this type of control statement the loop body is executed unconditionally for the first time.
6.1 Loops

Loops are used when we want to execute a part of program or block of statement several times. So, a loop may be defined as a block of statements which are repeatedly executed for a certain number of times or until a particular condition is satisfied. There are three types of loop statements in C:

1. For
2. While
3. Do...while

Each loop consists of two segments, one is known as the control statement and the other is the body of the loop. The control statement in loop decides whether the body is to be executed or not. Depending on the position of control statement in the loop, loops may be classified either entry_controlled loop or exit_controlled loop. While and For are entry_controlled loops where as do...while is exit_controlled loop.

**For Loop:**

For loops is useful to execute a statement for a number of times. When the number of repetitions is known in advance, the use of this loop will be more efficient. Thus, this loop is also known as determinate or definite loop. The general syntax for (counter initialization ; test condition ; increment or decrement)

```
for (counter initialization ; test condition ; increment or decrement)
{
    * body of loop *
}
```

Algorithm:

- Program to calculate factorial
  1. Start.
2. Print "Enter a number whose factorial is to be calculated".
3. Read num.
4. Initialize fact to 1 and counter i to 1
5. For i <= num
   fact = fact * i
   i++
End of For
6. Print fact as factorial of the number num.
7. Stop

For example:

/* Calculate the factorial of a number */
/* factorial.c */
#include<stdio.h>
main(
{
 int num, i;
 long fact = 1;
clrscr();
printf ("n Enter a number whose factorial is to be calculated : ");
scanf ("%d", &num);
for (i=1; i<=num; i++)
fact *= i; /* fact = fact*i */
printf ("n The factorial is : %d", fact);
getch();
}
Output:
Enter a number whose factorial is to be calculated : 5
The factorial is : 120

While Loop:
The while statement can be written as

while(condition) while(condition)
statement ; {
 statement ; /* body of the loop */
 statement ;
 - - - - - -
}
First the condition is evaluated; if it is true then the statements in the body of loop are executed. After the execution, again the condition is checked and if it is found to be true then again the statements in the body of loop are executed. This means that these statements are executed continuously till the condition is true and when it becomes false, the loop terminates and the control comes out of the loop. Each execution of the loop body is known as iteration.

A limitation with for loop is that the loop iterates a fixed number of times and the number of iterations depends on the value set in the conditional expression. So, while working with for loop the programmer must know how many times the loop body should be executed. But it is not possible for a
programmer in some cases to know in advance how many times the loop should iterate. And in such cases we have to use while loop.

```c
/* Program to print the sum of digits of any num */
#include<stdio.h>
main( )
{
    int n, sum = 0, rem ;
    clrscr( );
    printf("Enter the number :");
    scanf(%d", &n) ;
    while (n>0)
    {
        rem = n%10 ; /* taking last digit of number */
        sum+ = rem ; /* sum = sum + rem */
        n/ = 10 ; /*n = n/10 */
        /* skipping last digit */
    }/*end of while*/
    printf("Sum of digits = %d \\
", sum);
    getch( );
}
Output:
Enter the number : 1452
Sum of digits = 12
```

5.2.4 do-while Loop

The do...while statement is also used for looping. The body of this loop may contain a single statement or a block of statements. The general syntax is:
Here firstly the segments inside the loop body are executed and then the condition is evaluated. If the condition is true, then again the loop body is executed and this process continues until the condition becomes false. Unlike while loop, here a semicolon is placed after the condition. In a 'while' loop, first the condition is evaluated and then the statements are executed whereas in do while loop, first the statements are executed and then the condition is evaluated. So, if initially the condition is false the while loop will not execute at all, whereas the do while loop will always execute at least once.

/* program to print the number from 1 to 10 using do while */
#include <stdio.h>
main()
{
    int i = 1 ;clrscr( );
do
    {printf ("%d\t", i);
i++;
    } while (i<=10) ;
printf("\n") ; getch( );
} 
Output : 1 2 3 4 5 6 7 8 9 10
5.2.5 JUMPS IN LOOP

5.2.5.1 Local Jumps — goto

/* Program to print whether the number is even or odd */
#include<stdio.h>
main()
{
    int n;
clrscr();
printf( "Enter the number : ");
scanf ("%d", &n);
if (n%2 == 0)
    goto even;
else
    goto odd;
even :
    printf ("Number is even");
goto end;
odd :
    printf ("Number is odd");
end :
printf ("\n");
getch();
}
Enter the number : 6
Number is even.

**Exit() function:**

We have already known that we can jump out of a loop using either the break statement or goto statement. In a similar way, we can jump out of a program by using the library function exit(). In case, due to some reason, we wish to break out of a program and return to the operating system. The general syntax is

```c
if (condition) exit (0) ;
```

The exit() function takes an integer value as its argument. Normally zero is used to indicate normal termination and non zero value to indicate termination due to some error or abnormal condition. The use of exit() function requires the inclusion of the header file `<stdio.h>`.

```c
/* Program to demonstrate exit() */
#include<stdio.h>
#include<stdlib.h>
main( )
{
  int choice ;
  clrscr( );
  while(1)
  {
    printf (" 1. Create database
"
    printf (" 2. Insert new record
"
    printf (" 3. Modify a record
"
    printf (" 4. Delete a record
"
    printf (" 5. Display all records
"
    printf (" 6. Exit
") ; printf ("Enter your choice :") ;
 scanf ("%d", &choice) ;
 switch (choice)
{
    case1:
      printf (":database created - - \n\n") ;
      break ;
    case2:
      printf ("Record inserted - - \n\n") ;
      break ;
    case3:
      printf ("Record modified - - \n\n") ;
      break ;
    case4:
      printf ("Record deleted - - \n\n") ;
```
break;
case5:
    printf("Record displayed - -\n\n") ;
    break ;
case6:
    exit(1)
default:
    printf("Wrong choice\n") ;
} /* end of switch */
} /* end of while */
getch();
} /* end of main */

5.2.5.2 The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

// break loop example
#include <stdio.h>
using namespace std;
int main ()
{
    int n;
    for (n=10; n>0; n--)
    {
        printf("%d",n);
        if (n==3)
        {
            printf ("countdown aborted!");
            break;
        }
    }
    return 0;
}
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

5.2.5.3 The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration.
/* Program to demonstrate continue statement */
#include<stdio.h>
main(
{
    int i, num;
    clrscr();
    printf ("Enter a number :");
    scanf (%d", &num);
    printf ("The even numbers from 2 to %d are:
", num);
    for (i=1; i<=num; i++)
    {
        if(i%2!=0)
            continue;
        printf ("%d", i);
    } /* end of for loop */
    getch();
} /* end of main() */
Output:
Enter a number : 20
The even numbers from 2 to 20 are
    2 4 6 8 10 12 14 16 18 20

UNIT VI

6.1 ARRAYS
Introduction
An **array** is a fixed-sized, homogeneous, and widely-used data structure. By homogeneous, we mean that it consists of components which are all of the same type, called *element type* or *base type*. And by fixed sized, we mean that the number of components is constant, and so does not change during the lifetime of the structure. An array is also called a *random-access data structure*, because all components can be selected at random and are equally accessible. An array can be used to structure several data objects in the programming languages. A component of an array is selected by giving its *subscript*, which is an integer indicating the position of the component in the sequence. Therefore, an array is made of the pairs (value, index): it means that with every index, a value is associated. If every index is one single value then it is called a one-dimensional array, whereas if every index is a *n*-tuple \(\{i_1, i_2, i_3, \ldots, in\}\), the array is called a *n*-dimensional array.

**Memory Representation**

An array is represented in memory by using a sequential mapping. The basic characteristic of the sequential mapping is that every element is at a fixed distance apart. Therefore, if the \(i^{\text{th}}\) element is mapped into a location having an address \(a\), then the \((i + 1)^{\text{th}}\) element is mapped into the memory location having an address \((a + 1)\), as shown in **Figure 6.1**:

![Figure 6.1: Representation of an array.](image)

The address of the first element of an array is called the base address, so the address of the the \(i^{\text{th}}\) element is \(\text{Base address} + \text{offset of the } i^{\text{th}} \text{ element from base address}\) where the offset is computed as:

\[
\text{Offset of the } i^{\text{th}} \text{ element} = \text{number of elements before the } i^{\text{th}} \times \text{size of each element}.
\]

If \(\text{LB}\) is the lower bound, then the offset computation becomes:

\[
\text{offset} = (i - \text{LB}) \times \text{size}.
\]

**Representation of Two-Dimensional Array**

A two-dimensional array can be considered as a one-dimensional array whose elements are also one-dimensional arrays. So, we can view a two dimensional array as one single column of rows and map it sequentially as shown in **Figure 6.2**. Such a representation is called a *row-major representation*. 
The address of the element of the $i^{th}$ row and the $j^{th}$ column therefore is:

$$\text{addr}(a[i, j]) = (\text{number of rows placed before } i^{th} \text{ row} \times \text{size of a row}) + (\text{number of elements placed before the } j^{th} \text{ element in the } i^{th} \text{ row} \times \text{size of element})$$

where

Number of rows placed before $i^{th}$ row = $(i - LB1)$, and LB1 is the lower bound of the first dimension.

Size of a row = number of elements in a row $\times$ a size of element.

Number of elements in a row = $(UB2 - LB2 + 1)$, where $UB2$ and $LB2$ are the upper and lower bounds of the second dimension, respectively.

Therefore:

$$\text{addr}(a[i, j]) = ((i - LB1) \times (UB2 - LB2 + 1) \times \text{size}) + ((j - LB2)\times\text{size})$$

It is also possible to view a two-dimensional array as one single row of columns and map it sequentially as shown in Figure 6.3. Such a representation is called a column-major representation.
Figure 6.3: Column major representation of a two-dimensional array.

The address of the element of the $i^{th}$ row and the $j^{th}$ column therefore is:

$$\text{addr}(a[i, j]) = ( \text{number of columns placed before } j^{th} \text{ column } \times \text{size of a column}) + (\text{number of elements placed before the } i^{th} \text{ element in the } j^{th} \text{ column } \times \text{size of each element})$$

Number of columns placed before $j^{th}$ column = $(j - LB2)$ where $LB2$ is the lower bound of the second dimension.

Size of a column = number of elements in a column * size of element Number of elements in a column = $(UB1 - LB1 + 1)$, where $UB1$ and $LB1$ are the upper and lower bounds of the first dimension, respectively.

Therefore:

$$\text{addr}(a[i, j]) = ((j - LB2) \times (UB1 - LB1 + 1) \times \text{size}) + ((i - LB1)\times\text{size})$$

APPLICATION OF ARRAYS

Whenever we require a collection of data objects of the same type and want to process them as a single unit, an array can be used, provided the number of data items is constant or fixed. Arrays have a wide range of applications ranging from business data processing to scientific calculations to industrial projects.

Implementation of a Static Contiguous List

A list is a structure in which insertions, deletions, and retrieval may occur at any position in the list. Therefore, when the list is static, it can be implemented by using an array. When a list is implemented or realized by using an array, it is a contiguous list. By contiguous, we mean that the elements are placed consecutively one after another starting from some address, called the base address. The advantage of a list implemented using an array is that it is randomly accessible. The disadvantage of such a list is that insertions and deletions require moving of the entries, and so it is costlier. A static list
can be implemented using an array by mapping the $i^{th}$ element of the list into the $i^{th}$ entry of the array, as shown in Figure 6.4.

![Diagram of a static contiguous list]

Figure 6.4: Implementation of a static contiguous list.

**Program**

A complete C program for implementing a list with operations for reading values of the elements of the list and displaying them is given here:

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

void main()
{
    void read(int *,int);
    void dis(int *,int);
    int a[5],i,sum=0;

    clrscr();
    printf("Enter the elements of array \n");
    read(a,5); /*read the array*/
    printf("The array elements are \n");
    dis(a,5);
}

void read(int c[],int i)
{
    int j;
    for(j=0;j<i;j++)
        scanf("%d",&c[j]);
    fflush(stdin);
}

void dis(int d[],int i)
```

{ 
    int j;
    for(j=0;j<i;j++)
        printf("%d ",d[j]);
    printf("\n");
}

Example
Input
Enter the elements of the first array
15
30
45
60
75
Output
The elements of the first array are
15 30 45 60 75

6.1.1 Declaring Array Variables
The following declaration states that x is an array of three integer values.
    int x[3];
Each individual part (or element) of x is accessed by adding an index value in square brackets after the array name, e.g.
    x[0] = 74;
    printf("%d \n", x[2]);
Note that the index values range from 0 to one less than the number of elements. Arrays can have any number of dimensions; to declare a 2-D array of double :
    double matrix[10][10];

6.1.2 Initialising Array Variables
    int ndigit[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
    char greeting1[] = "hello";
    char greeting2[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
The last two declarations show the two alternative ways of initialising an array of characters. They are exactly equivalent, but one is somewhat easier to type! If the number of elements of the array is not specified, then the compiler will determine this from the initialiser. The declarations of greeting1 and greeting2 above both declare an array of 6 characters. The values given in a brace delimited initialiser (first and last examples above) must be constant expressions (one whose value can be determined at the time of compilation). If the initialiser is a single expression, the restriction does not apply.
If the number of expressions in the initialiser is less than the number of elements in the array then the remaining elements will be initialised as though they were static objects; that is arrays of numeric values will be initialised with zeros whatever the storage class of the array (external, static or automatic).
When initialising multi-dimensional arrays, extra braces can be used to show the reader how the values are to be used:

```c
int x[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } );
```
Extra braces are needed if values are to be omitted from the initialiser:

```c
int x[3][2] = { { 1, 2 }, { 3 }, { 5, 6 } } ;
```
In this example we have omitted the initialiser for `x[1][1]`: without the extra braces the compiler would initialise `x[1][1]` with 5 and would leave `x[2][1]` (i.e. the last element) uninitialised.

### 6.2 Character Arrays and Strings

A string is a sequence of characters. Any sequence or set of characters defined within double quotation symbols is a constant string. In C it is required to do some meaningful operations on strings they are:

- Reading string
- Displaying strings
- Combining or concatenating strings
- Copying one string to another.
- Comparing string & checking whether they are equal
- Extraction of a portion of a string

Strings are stored in memory as ASCII codes of characters that make up the string appended with '\0' (ASCII value of null). Normally each character is stored in one byte, successive characters are stored in successive bytes.

#### 6.2.1 Declaring and Initializing Strings

Following the discussion on characters arrays, the initialization of a string must the following form which is simpler to one dimension array.

```c
char month1[] = { 'j', 'a', 'n', 'u', 'a', 'r', 'y' };
```

Then the string month is initializing to January. This is perfectly valid but C offers a special way to initialize strings. The above string can be initialized `char month1[] = "January"`; The characters of the string are enclosed within a part of double quotes. The compiler takes care of string enclosed within a pair of a double quotes. The compiler takes care of storing the ASCII codes of characters of the string in the memory and also stores the null terminator in the end.

```c
/*String.c string variable*/
#include <stdio.h>
main()
{
char month[15];
printf ("Enter the string");
gets (month);
printf ("The string entered is \%s", month);
}
```
In this example string is stored in the character variable month the string is displayed in the statement.

```c
printf("The string entered is %s", month);
```

It is one dimension array. Each character occupies a byte. A null character (\0) that has the ASCII value 0 terminates the string. The figure shows the storage of string January in the memory recall that \0 specifies a single character whose ASCII value is zero.

| J | A | N | U | R | Y | \0 |

Character string terminated by a null character \0. A string variable is any valid C variable name & is always declared as an array. The general form of declaration of a string variable is

```c
char string_name[size];
```

The size determines the number of characters in the string name.

Example:

```c
char month[10];
char address[100];
```

The size of the array should be one byte more than the actual space occupied by the string since the compiler appends a null character at the end of the string.

**Reading Strings from the terminal:**

The function scanf with %s format specification is needed to read the character string from the terminal.

Example:

```c
char address[15];
scanf("%s", address);
```

Scanf statement has a draw back it just terminates the statement as soon as it finds a blank space, suppose if we type the string new york then only the string new will be read and since there is a blank space after word "new" it will terminate the string.
Note that we can use the `scanf` without the ampersand symbol before the variable name. In many applications it is required to process text by reading an entire line of text from the terminal. The function `getchar` can be used repeatedly to read a sequence of successive single characters and store it in the array.

We **cannot** manipulate strings since C does not provide any operators for string. For instance we cannot assign one string to another directly.

For example:

```c
String="xyz";
String1=string2;
```

Are **not valid**. To copy the chars in one string to another string we may do so on a character to character basis.

**Writing strings to screen:**

The `printf` statement along with format specifier `%s` to print strings on to the screen. The format `%s` can be used to display an array of characters that is terminated by the null character for example `printf("%s",name);` can be used to display the entire contents of the array name.

**Arithmetic operations on characters:**

We can also manipulate the characters as we manipulate numbers in C language. When ever the system encounters the character data it is automatically converted into a integer value by the system. We can represent a character as a interface by using the following method.

```c
X='a';
Printf("%d\n",x);
```

Will display 97 on the screen. Arithmetic operations can also be performed on characters for example `x='z'-1;` is a valid statement. The ASCII value of ‘z’ is 122 the statement the therefore will assign 121 to variable x.

It is also possible to use ASCII value of character constants in relational expressions for example `ch>'a' && ch <= 'z'` will check whether the character stored in variable `ch` is a lower case letter. A character digit can also be converted into its equivalent integer value suppose un the expression `a=character-'1';` where `a` is defined as an integer variable & character contains value 8 then `a= ASCII value of 8 ASCII value '1'=56-49=7.`

We can also get the support of the C library function to converts a string of digits into their equivalent integer values the general format of the function is `x=atoi(string)` here `x` is an integer variable & string is a character array containing string of digits.
6.2.2 STRING HANDLING FUNCTIONS
String operations (string.h)

C language recognizes that string is a different class of array by letting us input and output the array as a unit and are terminated by null character. C library supports a large number of string handling functions that can be used to array out many of the string manipulations such as:

- Length (number of characters in the string). `strlen(string)` function
- Concatentation (adding two are more strings) `strcat(string1,string2)` function
- Comparing two strings. `strcmp(string1,string2)` function
- Substring (Extract substring from a given string) `strstr(string1,string2)` function
- Copy(copies one string over another) `strcpy(string1,string2)` function

To do all the operations described here it is essential to include string.h library header file in the program.

`strlen()` function:

This function counts and returns the number of characters in a string. The length does not include a null character.

**Syntax**

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

Where `n` is integer variable. Which receives the value of length of the string.

**Example**

```
length=strlen("Hollywood");
```

The function will assign number of characters 9 in the string to a integer variable length.

```
/*writr a c program to find the length of the string using strlen() function*/
#include <stdio.h>
#include <string.h>
void main()
{
    char name[100];
    int length;
    printf("Enter the string");
    gets(name);
    printf("The length of the string is ",strlen(name));
    printf("\n");
}```
length=strlen(name);
printf("\nNumber of characters in the string is=%d",length);
}

**strcat() function:**

When you combine two strings, you add the characters of one string to the end of another string. This process is called concatenation. The `strcat()` function joins 2 strings together. It takes the following form

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

`string1` & `string2` are character arrays. When the function `strcat` is executed, `string2` is appended to `string1`. The string at `string2` remains unchanged.

**Example**

```
strcpy(string1,"sri");
strcpy(string2,"Bhagavan");
printf("%s",strcat(string1,string2));
```

From the above program segment, the value of `string1` becomes `sribhagavan`. The string at `str2` remains unchanged as `bhagawan`.

**strcmp() function:**

In C, you cannot directly compare the value of 2 strings in a condition like `if(string1==string2)`
Most libraries however contain the `strcmp()` function, which returns a zero if 2 strings are equal, or a non-zero number if the strings are not the same. The syntax of `strcmp()` is given below:

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

`string1` & `string2` may be string variables or string constants. `string1`, & `string2` may be string variables or string constants. Some computers return a negative if the `string1` is alphabetically less than the second and a positive number if the string is greater than the second.

**Example:**

```
strcmp("Newyork","Newyork") will return zero because 2 strings are equal.
strcmp("their","there") will return a 9 which is the numeric difference between ASCII 'i' and ASCII 'r'.
strcmp("The","the") will return 32 which is the numeric difference between ASCII "T" & ASCII "t".
```

**strcmpl() function**
This function is same as `strcmp()` which compares 2 strings but not case sensitive.

Example

```
strcmpi("THE","the"); will return 0.
```

`strcpy()` function:

C does not allow you to assign the characters to a string directly as in the statement `name="Robert";` Instead use the `strcpy()` function found in most compilers the syntax of the function is illustrated below.

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

`strcpy` function assigns the contents of string2 to string1. string2 may be a character array variable or a string constant.

```
strcpy(Name,"Robert");
```

In the above example Robert is assigned to the string called name.

`strlwr()` function:

This function converts all characters in a string from uppercase to lowercase.

syntax

```
strlwr(string);
```

For example:

```
strlwr("EXFORSYS") converts to Exforsys.
```

`strrev()` function:

This function reverses the characters in a string.

Syntax

```
strrev(string);
```

For ex `strrev("program")` reverses the characters in a string into “margrop”.

`strupr()` function:

This function converts all characters in a string from lower case to uppercase.
Syntax

```c
strupr(string);
```

For example `strupr("exforsys")` will convert the string to EXFORSYS.

```c
/* Example program to use string functions */
#include <stdio.h>
#include <string.h>
void main()
{
    char s1[20],s2[20],s3[20];
    int x,l1,l2,l3;
    printf("Enter the strings");
    scanf("%s%s",s1,s2);
    x=strcmp(s1,s2);
    if(x!=0)
        {printf("nStrings are not equal\n");
         strcat(s1,s2);
        }
    else
        printf("nStrings are equal");
    strcpy(s3,s1);
    l1=strlen(s1);
    l2=strlen(s2);
    l3=strlen(s3);
    printf("\ns1=%s \t length=%d characters\n",s1,l1);
    printf("\ns2=%s \t length=%d characters\n",s2,l2);
    printf("\ns3=%s \t length=%d characters\n",s3,l3);
}
```
UNIT IV

7.1 USER DEFINED FUNCTIONS

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, performs calculations and returns the results to the calling program.

7.2 There are many advantages in using functions in a program they are:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.

2. the length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigation.

4. 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 over from scratch.

5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.

6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

We already know that C support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.
Functions are used in C for the following reasons:

1. Many programs require that a specific function is repeated many times instead of writing the function code as many timers as it is required we can write it as a single function and access the same function again and again as many times as it is required.

2. We can avoid writing redundant program code of some instructions again and again.

3. Programs with using functions are compact & easy to understand.

4. Testing and correcting errors is easy because errors are localized and corrected.

5. We can understand the flow of program, and its code easily since the readability is enhanced while using the functions.

6. A single function written in a program can also be used in other programs also.

Function definition:

[ data type] function name (argument list)
    argument declaration;
    {
        local variable declarations;
        statements;
        [return expression]
    }

Example:

    mul(a,b)
    int a,b;
    {
        int y;
        y=a+b;
        return y;
    }

When the value of y which is the addition of the values of a and b. the last two statements ie, y=a+b; can be combined as

    return(y);
    return(a+b);
7.3 Types of functions:

A function may belong to any one of the following categories:

1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

7.3.1 Functions with no arguments and no return values:

Let us consider the following program

```c
#include<stdio.h>
main()
{
    statement1();
    starline();
    statement2();
    starline();
}
/*function to print a message*/
statement1()
{
    printf("\n Sample subprogram output");
}
statement2()
{
    printf("\n Sample subprogram output two");
}
starline()
{
    int a;
    for (a=1;a<60;a++)
    {
        printf("%c",'*');
        printf("\n");
    }
}
```

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. Similarly when it does not return value the calling function does not receive any data from the called function. A function that does not return any value cannot be used in an expression it can be used only as independent statement.
7.3.2 Functions with arguments but no return values:

The nature of data communication between the calling function and the arguments to the called function and the called function does not return any values to the calling function this shown in example below:

Consider the following:
Function calls containing appropriate arguments. For example the function call

\[
\text{value (500,0.12,5)}
\]

Would send the values 500,0.12 and 5 to the function value \((p, r, n)\) and assign values 500 to \(p\), 0.12 to \(r\) and 5 to \(n\). the values 500,0.12 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

Both the arguments actual and formal should match in number type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below:

```c
main()
{
    function1(a1,a2,a3,....an);
}

function1(f1,f2,f3,....fn)
{
    function body;
}
```

here a1,a2,a3 are actual arguments and f1,f2,f3 are formal arguments.

The no of formal arguments and actual arguments must be matching to each other suppose if actual arguments are more than the formal arguments, the extra actual arguments are discarded. If the number of actual arguments are less than the formal arguments then the unmatched formal arguments are initialized to some garbage values. In both cases no error message will be generated.

The formal arguments may be valid variable names, the actual arguments may be variable names expressions or constants. The values used in actual arguments must be assigned values before the function call is made.

When a function call is made only a copy of the values actual arguments is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.
Let us consider the following program

```c
/*Program to find the largest of two numbers using function*/
#include
main()
{
    int a,b;
    printf("Enter the two numbers");
    scanf("%d%d",&a,&b);
    largest(a,b);
}
/*Function to find the largest of two numbers*/
largest(int a, int b)
{
    if(a>b)
        printf("Largest element=%d",a);
    else
        printf("Largest element=%d",b);
}
```

In the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function does not return any value.

7.3.3 Functions with arguments and return values:
The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.

To assure a high degree of portability between programs a function should generally be coded without involving any input output operations. For example different programs may require different output formats for displaying the results. Theses 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. In the above type of function the following steps are carried out:

1. The function call transfers the controls along with copies of the values of the actual arguments of the particular function where the formal arguments are creates and assigned memory space and are given the values of the actual arguments.

2. The called function is executed line by line in normal fashion until the return statement is encountered. The return value is passed back to the function call is called function.

3. The calling statement is executed normally and return value is thus assigned to the calling function.
Note that the value return by any function when no format is specified is an integer.

7.4 Return value data type of function:

A C function returns a value of type int as the default data type when no other type is specified explicitly. For example if function does all the calculations by using float values and if the return statement such as return (sum); returns only the integer part of the sum. This is since we have not specified any return type for the sum. There is the necessity in some cases it is important to receive float or character or double data type. To enable a calling function to receive a non-integer value from a called function we can do the two things:

1. The explicit type specifier corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

   type_specifier function_name(argument list)

   Argument declaration;
   {
       function statement;
   }

   The type specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data the function is actually returning. The program given below illustrates the transfer of a floating-point value between functions done in a multiple function program.

   ```c
   main()
   {
       float x,y,add();
       double sub();
       x=12.345;
       y=9.82;
       printf("%f\n" add(x,y));
       printf("%lf\n"sub(x,y);
   }
   float add(a,b)
       float a,b;
       {
           return(a+b);
       }
   ```
double sub(p,q)
    double p,q;
    {
        return(p-q);
    }

We can notice that the functions too are declared along with the variables. These declarations clarify to the compiler that the return type of the function add is float and sub is double.

7.5 Void functions:

The functions that do not return any values can be explicitly defined as void. This prevents any accidental use of these functions in expressions.

Example:

main()
{
    void starline();
    void message();
    -------
}
void printline ()
{
    statements;
}
void value()
{
    statements;
}

7.6 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:

```c
#include<stdio.h>
main()
{
    int a,b,c;
    float ratio();
    scanf("%d%d%d",&a,&b,&c);
    printf("%fn",ratio(a,b,c));
}
```
float ratio(x,y,z)
    int x,y,z;
    {
        if(difference(y,z))
            return(x/y-z);
        else
            return(0,0);
    }

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

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.

7.7 Recursion:

Recursive function is a function that calls itself. When a function calls another function and that second function calls the third function then this kind of a function is called nesting of functions. But a recursive function is the function that calls itself repeatedly.

A simple example:

```
#include<stdio.h>
main()
{
    printf("this is an example of recursive function");
    main();
}
```

when this program is executed. The line is printed repeatedly and indefinitely. We might have to abruptly terminate the execution.
```c
#include<stdio.h>
long factorial(int);  

int main()
{
    int n;
    long f;
    printf("Enter an integer to find factorial\n");
    scanf("%d", &n);

    if (n < 0)
        printf("Negative integers are not allowed.\n");
    else
    {
        f = factorial(n);
        printf("%d! = %ld\n", n, f);
    }
    return 0;  // end of main()
}
long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return(n * factorial(n-1));
}

Functions and arrays:

We can pass an entire array of values into a function just as we pass indiviual variables. In this
 task it is essential to list the name of the array along with functions arguments without any
 subscripts and the size of the array as arguments

For example: The call

    Largest(a,n);

Will pass all the elements contained in the array a of size n. the called function expecting this
call must be appropriately defined. The largest function header might look like:

    float smallest(array,size);
    float array[];
    int size;
```
The function smallest is defined to take two arguments, the name of the array and the size of the array to specify the number of elements in the array. The declaration of the formal argument array is made as follows:

```c
float array[];
```

The above declaration indicates to compiler that the arguments array is an array of numbers. It is not necessary to declare size of the array here. While dealing with array arguments we should remember one major distinction. If a function changes the value the value of an array elements then these changes will be made to the original array that passed to the function. When the entire array is passed as an argument, the contents of the array are not copied into the formal parameter array instead information about the address of the array elements are passed on to the function. Therefore any changes introduced to array elements are truly reflected in the original array in the calling function.

## 7.8 The scope and lifetime of variables in functions:

The scope and lifetime of the variables define in C is not same when compared to other languages. The scope and lifetime depends on the storage class of the variable in C language the variables can be any one of the four storage classes:

1. Automatic Variables
2. External variable
3. Static variable
4. Register variable.

The scope actually determines over which part or parts of the program the variable is available. The lifetime of the variable retains a given value. During the execution of the program. Variables can also be categorized as local or global. Local variables are the variables that are declared within that function and are accessible to all the functions in a program and they can be declared within a function or outside the function also.

**Automatic variables:**

Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits. Automatic variables are local or private to a function in which they are defined by default all variable declared without any storage specification is automatic. The values of variable remains unchanged to the changes that may happen in other functions in the same program and by doing this no error occurs.

```c
/* A program to illustrate the working of auto variables*/
#include <stdio.h>
void main()
{
        int m=1000;
```
function2();
printf("%d\n",m);
}

function1()
{
    int m=10;
    printf("%d\n",m);
}
function2()
{
    int m=100;
    function1();
    printf("%d\n",m);
}

A local variable lives throughout the whole program although it accessible only in the main. A program with two subprograms function1 and function2 with m as automatic variable and is initialized to 10,100,1000 in function 1 function2 and function3 respectively. When executes main calls function2 which in turns calls function1. When main is active m=1000. But when function2 is called, the main m is temporarily put on the shelf and the new local m=100 becomes active. Similarly when function1 is called both previous values of m are put on shelf and latest value (m=10) become active, a soon as it is done main (m=1000) takes over. The output clearly shows that value assigned to m in one function does not affect its value in the other function. The local value of m is destroyed when it leaves a function.

External variables:

Variables which are common to all functions and accessible by all functions of a program are internal variables. External variables can be declared outside a function.

Example

#include <stdio.h>
int sum;
float percentage;
main()
{
    ....
    ....
}
function2()
{
    ....
    ....
}
The variables sum and percentage are available for use in all the three functions main, function1, function2. Local variables take precedence over global variables of the same name.

For example:

```c
int i = 10;
void example(data)
    int data;
    {
        int i = data;
        printf("%d",i);   // prints i as 45 : preference goes to locally declared i
    }

main()
{
    example(45);
    printf("%d",i);   // prints i as 10 : print the value of globally declared i
}
```

In the above example both the global variable and local variable have the same name as i. The local variable i take precedence over the global variable. Also the value that is stored in integer i is lost as soon as the function exits.

A global value can be used in any function all the functions in a program can access the global variable and change its value the subsequent functions get the new value of the global variable, it will be inconvenient to use a variable as global because of this factor every function can change the value of the variable on its own and it will be difficult to get back the original value of the variable if it is required.

Global variables are usually declared in the beginning of the main program ie., before the main program however c provides a facility to declare any variable as global this is possible by using the keyword storage class extern. Although a variable has been defined after many functions the external declaration of y inside the function informs the compiler that the variable y is integer type defined somewhere else in the program. The external declaration does not allocate storage space for the variables. In case of arrays the definition should include their size as well. When a variable is defined inside a function as extern it provides type information only for that function. If it has to be used in other functions then again it has to be re-declared in that function also.
Example:

```c
#include <stdio.h>
main()
{
    int n;
    out_put();
    extern float salary[];
    ....
    ....
    out_put();
}

void out_put()
{
    extern float salary[];
    int n;
    ....
    ....
    ....
} 
float salary[size];
```

A function when its parameters and function body are specified this tells the compiler to allocate space for the function code and provides type info for the parameters. Since functions are external by default we declare them (in calling functions) without the qualifier `extern`.

**Static variables:**

The value given to a variable declared by using keyword `static` persists until the end of the program.

A static variable is initialized only once, when the program is compiled. It is never initialized again. During the first call to `stat` in the example shown below `x` is incremented to 1. Because `x` is static, this value persists and therefore the next call adds another 1 to `x` giving it a value of 2. The value of `x` becomes 3 when third call is made. If we had declared `x` as an auto then output would have been `x=1` all the three times.

```c
#include <stdio.h>
main()
{
    int j;
    for(j=1;j<3;j++)
    {
        stat();
    }
}
```
stat()
{
    static int x=0;
    x=x+1;
    printf("x=%d\n",x);
}

Register variables:

A variable is usually stored in the memory but it is also possible to store a variable in the compiler's register by defining it as register variable. The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster.

This is done as follows:

    register int count;

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

8

8.1 POINTERS

Introduction:

In C language, a pointer is a variable that points to or references a memory location in which data is stored. Each memory cell in the computer has an address which can be used to access its location. A pointer variable points to a memory location. By making use of pointer, we can access and change the contents of the memory location.

8.2 Pointer declaration:

A pointer variable contains the memory location of another variable. You begin the declaration of a pointer by specifying the type of data stored in the location identified by the pointer. The asterisk tells the compiler that you are creating a pointer variable. Finally you give the name of the pointer variable. The pointer declaration syntax is as shown below.

    type *variable name ;

Example:

    int *ptr;
    float *string;
8.3 Address operator:

The * and & Operators

& gives the address of something in memory, that is it generates a pointer to the object. * gives what is pointed at by a pointer (called dereferencing).

For example, assuming the following declaration:

```c
int i = 17;
```

Then the expression &i generates a pointer to the variable i. We can then find out what this pointer is pointing at by applying the * operator to the newly generated pointer; thus *i should have the value 17.

Once we declare a pointer variable, we point the variable to another variable. We can do this by assigning the address of the variable to the pointer as in the following example:

```c
ptr=&num;
```

The above declaration places the memory address of num variable into the pointer variable ptr. If num is stored in memory 21260 address then the pointer variable ptr will contain the memory address value 21260.

```c
#include<stdio.h>
main()
{
    int *ptr;
    int sum;
    sum=45;
    ptr=&sum;
    printf("n sum variable value is %d\n", sum);
    printf("n The 'ptr' pointer variable value is %d", ptr);
}
```

We will get the same result by assigning the address of num to a regular (non pointer) variable. The benefit is that we can also refer to the pointer variable as *ptr the asterisk tells to the computer that we are not interested in the value 21260 but in the value stored in that memory location. While the value of pointer is 21260 the value of sum is 45 however we can assign a value to the pointer * ptr as in *ptr=45.

When we place the value 45 in the memory address pointer by the variable ptr. Since the pointer contains the address 21260 the value 45 is placed in that memory location. And since this is the location of the variable num the value also becomes 45. This shows how we can change the value of pointer directly using a pointer and the indirection pointer.
/* Program to display the contents of the variable their address using pointer variable*/

#include <stdio.h>
void main()
{
    int num, *intptr;
    float x, *floptr;
    char ch, *cptr;
    num=123;
    x=12.34;
    ch='a';
    intptr=&x;
    cptr=&ch;
    floptr=&x;
    printf("Num %d stored at address %un",*intptr,intptr);
    printf("Value %f stored at address %un",*floptr,floptr);
    printf("Character %c stored at address %un",*cptr,cptr);
}

8.4 Pointer expressions & pointer arithmetic:

Like any other variable, pointer variable can be used in arithmetic expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

\[
y = *p1 * *p2;
sum = sum + *p1;
z = 5* - *p2/p1;
*p2 = *p2 + 10;
\]

C language allows us to add integers to, subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 > p2 , p1==p2 and p1!=p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/
#include <stdio.h>
main()
{
    int ptr1,ptr2;
    int a,b,x,y,z;
    a=30;b=6;
    ptr1=&a;
ptr2=&b;
  x= "ptr1+ *ptr2 -6;
  y= 6* - *ptr1/ *ptr2 +30;
printf("nAddress of a +%u",ptr1);
printf("nAddress of b %u",ptr2);
printf("na=%d, b=%d",a,b);
printf("nx=%d,y=%d",x,y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf("na=%d, b=%d",a,b);
}

8.5 Pointers and function:

The pointer are very much used in a function declaration. Sometimes only with a pointer a complex function can be easily represented and success. The usage of the pointers in a function definition may be classified into two groups.

1. Call by reference
2. Call by value.

Call by value:

We have seen that a function is invoked there will be a link established between the formal and actual parameters. A temporary storage is created where the value of actual parameters is stored. The formal parameters picks up its value from storage area the mechanism of data transfer between actual and formal parameters allows the actual parameters mechanism of data transfer is referred as call by value. The corresponding formal parameter represents a local variable in the called function. The current value of corresponding actual parameter becomes the initial value of formal parameter. The value of formal parameter may be changed in the body of the actual parameter. The value of formal parameter may be changed in the body of the subprogram by assignment or input statements. This will not change the value of actual parameters.

Call by Reference:

When we pass address to a function the parameters receiving the address should be pointers. The process of calling a function by using pointers to pass the address of the variable is known as call by reference. The function which is called by reference can change the values of the variable used in the call.

For instance consider program1
#include <stdio.h>
main()
{
  int x=50, y=70;
interchange(x,y);
printf("x=%d y=%d",x,y);
}

interchange(x1,y1) // function definition interchange()
int x1,y1;
{
    int z1;
    z1=x1;
    x1=y1;
    y1=z1;
    printf("x1=%d y1=%d",x1,y1);
}

Here the value to function interchange is passed by value.

Consider program2

main()
{
    int x=50, y=70;
    interchange(&x,&y);
    printf("x=%d y=%d",x,y);
}

interchange(x1,y1)
int *x1,*y1;
{
    int z1;
    z1=*x1;
    *x1=*y1;
    *y1=z1;
    printf("*x=%d *y=%d",x1,y1);
}

Here the function is called by reference. In other words address is passed by using symbol & and the value is accessed by using symbol *.

The main difference between them can be seen by analyzing the output of program1 and program2.

The output of program1 that is call by value is

x1=70 y1=50
x=50 y=70
But the output of **program2** that is **call by reference** is

\[ *x_1=70 \quad *y_1=50 \]
\[ x=70 \quad y=50 \]

This is because in case of **call by value** the **value is passed to function** named as interchange and there the value got interchanged and got printed as

\[ x_1=70 \quad y_1=50 \]

and again since no values are returned back and therefore original values of \( x \) and \( y \) as in main function namely

\[ x=50 \quad y=70 \] got printed.

But in case of **call by reference** **address of the variable got passed** and therefore what ever changes that happened in function interchange got reflected in the address location and therefore the got reflected in original function call in main also without explicit return value. So value got printed as \( *x=70 \quad *y=50 \) and \( x=70 \quad y=50 \)

### 8.6 Pointer to arrays:

An array name behaves in many ways as though it is a pointer to the first element of the array.

An array is actually very much like pointer. We can declare the arrays first element as \( a[0] \) or as int \( *a \) because \( a[0] \) is an address and \( *a \) is also an address the form of declaration is equivalent. The difference is pointer is a variable and can appear on the left of the assignment operator that is lvalue. The array name is constant and cannot appear as the left side of assignment operator.

```c
/* A program to display the contents of array using pointer */
main()
{
    int a[100];
    int i,j,n;
    printf("nEnter the elements of the arrayn");
    scanf("%d",&n);
    printf("Enter the array elements");
    for(I=0; I < n; I++)
        scanf("%d",&a[I]);
    printf("Array element are");
    for(ptr=a; ptr < (a+n); ptr++)
        printf("Value of a[%d]=%d stored at address %u",j++,*ptr,ptr);
}
```
Strings are characters arrays and here last element is `\0`. Arrays and pointers to char arrays can be used to perform a number of string functions.

8.7 **Pointers and structures:**

We know the name of an array stands for the address of its zeroth element the same concept applies for names of arrays of structures. Suppose `item` is an array variable of struct type. Consider the following declaration:

```c
struct products
{
    char name[30];
    int manufac;
    float net;
} item[2], *ptr;
```

This statement declares `item` as array of two elements, each type `struct products` and `ptr` as a pointer data objects of type `struct products`, the assignment `ptr = item;`

would assign the address of zeroth element to `product[0]`. Its members can be accessed by using the following notation.

```c
ptr->name;
ptr->manufac;
ptr->net;
```

The symbol `->` is called arrow pointer and is made up of minus sign and greater than sign. Note that `ptr->` is simple another way of writing `product[0]`.

When the pointer is incremented by one it is made to pint to next record ie `item[1]`. The following statement will print the values of members of all the elements of the product array.

```c
for(ptr = item; ptr < item + 2; ptr++)
    printf("%s%d%f\n", ptr->name, ptr->manufac, ptr->net);
```

We could also use the notation

```c
(*ptr).number
```

to access the member number. The parenthesis around `ptr` are necessary because the member operator `\'.' Has a higher precedence than the operator `\'\'.
8.8 **Pointers on pointer/Chain of pointers:**

While pointers provide enormous power and flexibility to the programmers, they may use cause manufactures if it not properly handled. Consider the following precautions using pointers to prevent errors. We should make sure that we know where each pointer is pointing in a program. Here are some general observations and common errors that might be useful to remember.

A pointer contains garbage until it is initialized. Since compilers cannot detect uninitialized or wrongly initialized pointers, the errors may not be known until we execute the program remember that even if we are able to locate a wrong result, it may not provide any evidence for us to suspect problems in the pointers.

The abundance of c operators is another cause of confusion that leads to errors. For example the expressions such as

*ptr++, *p[],(ptr).member

etc should be carefully used. A proper understanding of the precedence and associativity rules should be carefully used.

8.9 **Strings as pointers:**

Another way of accessing a contiguous chunk of memory, instead of with an array, is with a pointer.

Since we are talking about strings, which are made up of characters, we’ll be using pointers to characters, or rather, char *’s.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a char * to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a character pointer to keep track of a string.

```c
char label[] = "Single";
char label2[10] = "Married";
char *labelPtr;
labelPtr = label;
```

We would have something like the following in memory (e.g., supposing that the array label started at memory address 2000, etc.):

```
label @2000
S I N G L E

label2 @3000
M A R R I E D
```

Note: Since we assigned the pointer the address of an array of characters, the pointer must be a character pointer—the types must match.

Also, to assign the address of an array to a pointer, we do not use the address-of (&) operator since the name of an array (like label) behaves like the address of that array in this context.

Now, we can use labelPtr just like the array name label. So, we could access the third character in the string with:

\[ \text{printf( "Third char is: ", labelPtr[2]);} \]

It's important to remember that the only reason the pointer labelPtr allows us to access the label array is because we made labelPtr point to it. Suppose, we do the following:

\[ \text{labelPtr = label2;} \]

Now, no longer does the pointer labelPtr refer to label, but now to label2 as follows:

\[ \text{label2 } \text{ @3000} \]

\[ \text{M A R R I E D} \backslash 0 \]

\[ \text{labelPtr } \text{ @4000} \]

\[ 3000 \]

So, now when we subscript using labelPtr, we are referring to characters in label2. The following:

\[ \text{printf( "Third char is: ", labelPtr[2]);} \]

prints out \( r \), the third character in the label2 array.

**Passing strings:**

Just as we can pass other kinds of arrays to functions, we can do so with strings.

Below is the definition of a function that prints a label and a call to that function:

\[ \text{void PrintLabel(char the_label[])} \]
\[ \{ \]
\[ \text{printf( "Label: ", the_label);} \]
\[ \} \]

\[ \text{...} \]

\[ \text{int main()} \]
\[ \{ \]
\[ \text{char label[]} = "Single"; \]
\[ \text{...} \]
PrintLabel(label);
...

Since label is a character array, and the function PrintLabel() expects a character array, the above makes sense.

However, if we have a pointer to the character array label, as in:
char *labelPtr = label;

then we can also pass the pointer to the function, as in:
PrintLabel(labelPtr);

The results are the same. Why??

Answer: When we declare an array as the parameter to a function, we really just get a pointer. Plus, arrays are always automatically passed by reference (e.g., a pointer is passed).

So, PrintLabel() could have been written in two ways:

```c
void PrintLabel(char the_label[])
{
    printf("Label: ", the_label);
}
```

OR

```c
void PrintLabel(char *the_label)
{
    printf("Label: ", the_label);
}
```

There is no difference because in both cases the parameter is really a pointer.

Note: In C++, there is a difference in the use of brackets ([]) when declaring a global, static or local array variable versus using this array notation for the parameter of a function.

With a parameter to a function, you always get a pointer even if you use array notation. This is true for all types of arrays.

**Dynamically-allocated string:**

Since sometimes you do not know how big a string is until run-time, you may have to resort to dynamic allocation.

The following is an example of dynamically-allocating space for a string at run-time:

```c
void SomeFunc(int length)
{
...
```
char *str;

// Don't forget extra char for nul character.
str = new char[length + 1];

...

// Done with str.
delete [] str;
...

Basically, we've just asked new (the allocation operator) to give us back enough space for an array of the desired size. Operator new requires the type of elements (here, char) and the number of elements needed (given as the array size between [ and ]).

Note that unlike static allocation, e.g.:
   char name[20];
the size can be variable (when using new for allocation).

We keep track of the dynamically-allocated array with a pointer (e.g., the return value of the call to new is stored in str). We then can use that pointer as we used pointers to statically-allocated arrays above (i.e., we access individual characters with str[i], pass the string to a function, etc.).

Finally, note that when we are done using the string, we must deallocate it.
8.2 FILES IN C

C supports a number of functions that have the ability to perform basic file operations, which include:

1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

Real life situations involve large volume of data and in such cases, the console oriented I/O operations pose two major problems:
It becomes cumbersome and time consuming to handle large volumes of data through terminals. The entire data is lost when either the program is terminated or computer is turned off therefore it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

### File operation functions in C:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen()</td>
<td>Creates a new file for use</td>
</tr>
<tr>
<td></td>
<td>Opens an existing file for use</td>
</tr>
<tr>
<td>fclose()</td>
<td>Closes a file which has been opened for use</td>
</tr>
<tr>
<td>getc()</td>
<td>Reads a character from a file</td>
</tr>
<tr>
<td>putc()</td>
<td>Writes a character to a file</td>
</tr>
<tr>
<td>fprintf()</td>
<td>Writes a set of data values to a file</td>
</tr>
<tr>
<td>fscanf()</td>
<td>Reads a set of data values from a file</td>
</tr>
<tr>
<td>getw()</td>
<td>Reads an integer from a file</td>
</tr>
<tr>
<td>putw()</td>
<td>Writes an integer to the file</td>
</tr>
<tr>
<td>fseek()</td>
<td>Sets the position to a desired point in the file</td>
</tr>
<tr>
<td>ftell()</td>
<td>Gives the current position in the file</td>
</tr>
<tr>
<td>rewind()</td>
<td>Sets the position to the beginning of the file</td>
</tr>
</tbody>
</table>
Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the file name, data structure, purpose.

The general format of the function used for opening a file is

```c
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable `fp` as a pointer to the data type `FILE`. As stated earlier, `FILE` is a structure that is defined in the I/O Library. The second statement opens the file named `filename` and assigns an identifier to the `FILE` type pointer `fp`. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program.

The second statement also specifies the purpose of opening the file. The `mode` does this job.

- `r` open the file for read only.
- `w` open the file for writing only.
- `a` open the file for appending data to it.

Consider the following statements:

```c
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

In these statements the `p1` and `p2` are created and assigned to open the files `data` and `results` respectively. The file `data` is opened for reading and `result` is opened for writing. In case the `results` file already exists, its contents are deleted and the files are opened as a new file. If `data` file does not exist error will occur.

Closing a file:

The input output library supports the function to close a file; it is in the following format.

```c
fclose(file_pointer);
```

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.

Observe the following program.

```c
....
FILE *p1, *p2;
p1=fopen("Input","w");
p2=fopen("Output","r");
```
The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

**The getc and putc functions**

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```c
#include<stdio.h>
main()
{
  file *f1;
  printf("Data input output");
  f1=fopen("Input","w"); /*Open the file Input*/
  while((c=getchar())!=EOF) /*get a character from keyboard*/
    putc(c,f1); /*write a character to input*/
  fclose(f1); /*close the file input*/
  printf("nData outputn");
  f1=fopen("INPUT","r"); /*Reopen the file input*/
  while((c=getc(f1))!=EOF)
    printf("%c",c);
  fclose(f1);
}
```

**The getw and putw functions:**

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);
```
/*Example program for using getw and putw functions*/
#include< stdio.h >
main()
{
    FILE *f1,*f2,*f3;
    int number;  
    printf("Contents of the data file");
    f1=fopen("DATA","W");
    for(i=1;i< 30;i++)
    {
        scanf("%d",&number);
        if(number== -1)
            break;
        putw(number,f1);
    }
    fclose(f1);
    f1=fopen("DATA","r");
    f2=fopen("ODD","w");
    f3=fopen("EVEN","w");
    while((number=getw(f1))!=EOF)/* Read from data file*/
    {
        if(number%2==0) /*Write to even file*/
            putw(number,f3);
        else
            putw(number,f2); /*write to odd file*/
    }
    fclose(f1);
    fclose(f2);
    fclose(f3);
    f2=fopen("ODD","r");
    f3=fopen("EVEN","r");
    printf("Contents of the odd file");
    while(number=getw(f2))!=EOF)
    {
        printf("%d",number);
    }
    printf("Contents of the even file");
    while(number=getw(f3))!=EOF)
    {
        printf("%d",number);
    }
    fclose(f2);
    fclose(f3);
}

The fprintf & fscanf functions:
The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used. The general form of fprintf is

fprintf(fp,"control string", list);

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

fprintf(f1,%s%d%f",name,age,7.5);

Here name is an array variable of type char and age is an int variable
The general format of fscanf is

fscanf(fp,"controlstring",list);

This statement would cause the reading of items in the control string.

Example:

fscanf(f2,"5s%d",item,&quantity");

Like scanf, fscanf also returns the number of items that are successfully read.

/*Program to handle mixed data types*/
#include<stdio.h>
main()
{
FILE *fp;
int num,qty,l;
float price,value;
char item[10],filename[10];
printf("Input filename");
scanf("%s",filename);
fp=fopen(filename,"w");
printf("Input inventory datann");
printf("Item namenumber pricenquantityn");
for I=1;I<=3;I++)
{
 fscanf(stdin,"%s%d%f%d",item,&number,&price,&quality);
fprintf(fp,"%s%d%f%d",item,number,price,quantity);
}
fclose (fp);
fprintf(stdout,"nn");
fp=fopen(filename,"r");
printf("Item name number price quantity value");
for(I=1;I< =3;I++)
{
    fscanf(fp,"%s%d%f%d",item,&number,&prince,&quality);
    value=price*quantity;
    fprintf(stdout,"%s%d%f%d%dn",item,number,price,quantity,value);
} 
fclose(fp);

Random access to files:

Sometimes it is required to access only a particular part of the and not the complete file. This can be accomplished by using the following function:

1 > fseek
    fseek function:

The general format of fseek function is as follows:

    fseek(file pointer,offset, position);

This function is used to move the file position to a desired location within the file. Fileptr is a pointer to the file concerned. Offset is a number or variable of type long, and position in an integer number. Offset specifies the number of positions (bytes) to be moved from the location specified by the position. The position can take the 3 values.

Value	Meaning
0	Beginning of the file
1	Current position
2	End of the file.
The C Preprocessor
The C preprocessor modifies a source code file before handing it over to the compiler. You're most likely used to using the preprocessor to include files directly into other files, or #define constants, but the preprocessor can also be used to create "inlined" code using macros expanded at compile time and to prevent code from being compiled twice.

There are essentially three uses of the preprocessor--directives, constants, and macros. Directives are commands that tell the preprocessor to skip part of a file, include another file, or define a constant or macro. Directives always begin with a sharp sign (#) and for readability should be placed flush to the left of the page. All other uses of the preprocessor involve processing #define’d constants or macros. Typically, constants and macros are written in ALL CAPS to indicate they are special (as we will see).

Header Files
The #include directive tells the preprocessor to grab the text of a file and place it directly into the current file. Typically, such statements are placed at the top of a program--hence the name "header file" for files thus included.

Constants
If we write
    #define [identifier name] [value]
whenever [identifier name] shows up in the file, it will be replaced by [value].

If you are defining a constant in terms of a mathematical expression, it is wise to surround the entire value in parentheses:
    #define PI_PLUS_ONE (3.14 + 1)
By doing so, you avoid the possibility that an order of operations issue will destroy the meaning of your constant:
    x = PI_PLUS_ONE * 5;
Without parentheses, the above would be converted to
    x = 3.14 + 1 * 5;
which would result in 1 * 5 being evaluated before the addition, not after. Oops!

It is also possible to write simply
    #define [identifier name]
which defines [identifier name] without giving it a value. This can be useful in conjunction with another set of directives that allow conditional compilation.
Conditional Compilation

There are a whole set of options that can be used to determine whether the preprocessor will remove lines of code before handing the file to the compiler. They include #if, #elif, #else, #ifdef, and #ifndef. An #if or #if/#elif/#else block or a #ifdef or #ifndef block must be terminated with a closing #endif.

The #if directive takes a numerical argument that evaluates to true if it's non-zero. If its argument is false, then code until the closing #else, #elif, of #endif will be excluded.

Commenting out Code

Conditional compilation is a particularly useful way to comment out a block of code that contains multi-line comments (which cannot be nested).

```c
#if 0
    /* comment ...
       */
    // code
    /* comment */
#endif
```

Include Guards

Another common problem is that a header file is required in multiple other header files that are later included into a source code file, with the result often being that variables, structs, classes or functions appear to be defined multiple times (once for each time the header file is included). This can result in a lot of compile-time headaches. Fortunately, the preprocessor provides an easy technique for ensuring that any given file is included once and only once.

By using the #ifndef directive, you can include a block of text only if a particular expression is undefined; then, within the header file, you can define the expression. This ensures that the code in the #ifndef is included only the first time the file is loaded.

```c
#ifndef _FILE_NAME_H_
#define _FILE_NAME_H_
/* code */
#endif // #ifndef _FILE_NAME_H_
```

Notice that it's not necessary to actually give a value to the expression _FILE_NAME_H_. It's sufficient to include the line "#define _FILE_NAME_H_" to make it "defined". (Note that there is an n in #ifndef--it stands for "if not defined").

A similar tactic can be used for defining specific constants, such as NULL:

```c
#ifndef NULL
#define NULL (void *)0
#endif
```
Notice that it’s useful to comment which conditional statement a particular #endif terminates. This is particularly true because preprocessor directives are rarely indented, so it can be hard to follow the flow of execution.

On many compilers, the #pragma once directive can be used instead of include guards.

### Macros

The other major use of the preprocessor is to define macros. The advantage of a macro is that it can be type-neutral (this can also be a disadvantage, of course), and it’s inlined directly into the code, so there isn’t any function call overhead. (Note that in C++, it’s possible to get around both of these issues with templated functions and the inline keyword.)

A macro definition is usually of the following form:

```
#define MACRO_NAME(arg1, arg2, ...) [code to expand to]
```

For instance, a simple increment macro might look like this:

```
#define INCREMENT(x) x++
```

They look a lot like function calls, but they’re not so simple. There are actually a couple of tricky points when it comes to working with macros. First, remember that the exact text of the macro argument is "pasted in" to the macro. For instance, if you wrote something like this:

```
#define MULT(x, y) x * y
```

and then wrote

```
int z = MULT(3 + 2, 4 + 2);
```

what value do you expect z to end up with? The obvious answer, 30, is wrong! That’s because what happens when the macro MULT expands is that it looks like this:

```
int z = 3 + 2 * 4 + 2;    // 2 * 4 will be evaluated first!
```

So z would end up with the value 13! This is almost certainly not what you want to happen. The way to avoid it is to force the arguments themselves to be evaluated before the rest of the macro body. You can do this by surrounding them by parentheses in the macro definition:

```
#define MULT(x, y) (x) * (y)
```

```
int z = 3 + 2 * 4 + 2;    // now MULT(3 + 2, 4 + 2) will expand to (3 + 2) * (4 + 2)
```

But this isn’t the only gotcha! It is also generally a good idea to surround the macro’s code in parentheses if you expect it to return a value. Otherwise, you can get similar problems as when you define a constant. For instance, the following macro, which adds 5 to a given argument, has problems when embedded within a larger statement:

```
#define ADD_FIVE(a) (a) + 5
```

```
int x = ADD_FIVE(3) * 3;
```

// this expands to (3) + 5 * 3, so 5 * 3 is evaluated first
// Now x is 18, not 24!
To fix this, you generally want to surround the whole macro body with parentheses to prevent the surrounding context from affecting the macro body.

\[
\text{#define ADD_FIVE(a) \((a) + 5\)}
\]

int \text{x} = \text{ADD_FIVE}(3) * 3;

On the other hand, if you have a multiline macro that you are using for its side effects, rather than to compute a value, you probably want to wrap it within curly braces so you don't have problems when using it following an if statement.

// We use a trick involving exclusive-or to swap two variables
#define SWAP(a, b) \(a ^= b; b ^= a; a ^= b;\)

int \text{x} = 10;
int \text{y} = 5;

// works OK
SWAP(x, y);

// What happens now?
if(x < 0)
SWAP(x, y);

When SWAP is expanded in the second example, only the first statement, \(a ^= b\), is governed by the conditional; the other two statements will always execute. What we really meant was that all of the statements should be grouped together, which we can enforce using curly braces:

#define SWAP(a, b) \{
\text{a} ^= \text{b}; \text{b} ^= \text{a}; \text{a} ^= \text{b};\}

Now, there is still a bit more to our story! What if you write code like so:

#define SWAP(a, b) \{
\text{a} ^= \text{b}; \text{b} ^= \text{a}; \text{a} ^= \text{b};\}

int \text{x} = 10;
int \text{y} = 5;
int \text{z} = 4;

// What happens now?
if(x < 0)
SWAP(x, y);
else
SWAP(x, z);

Then it will not compile because semicolon after the closing curly brace will break the flow between if and else. The solution? Use a do-while loop:

#define SWAP(a, b) \do{ \text{a} ^= \text{b}; \text{b} ^= \text{a}; \text{a} ^= \text{b}; } \while(0)

int \text{x} = 10;
int \text{y} = 5;
int \text{z} = 4;
// What happens now?
if (x < 0)
    SWAP(x, y);
else
    SWAP(x, z);

Now the semi-colon doesn’t break anything because it is part of the expression. (By the way, note that we didn’t surround the arguments in parentheses because we don’t expect anyone to pass an expression into swap!)

**Multiline macros**

Until now, we’ve seen only short, one line macros (possibly taking advantage of the semicolon to put multiple statements on one line.) It turns out that by using a the "\" to indicate a line continuation, we can write our macros across multiple lines to make them a bit more readable.

For instance, we could rewrite swap as

```c
#define SWAP(a, b)  
  { 
    a ^= b;  
    b ^= a;  
    a ^= b;  
  }
```

Notice that you **do not** need a slash at the end of the last line! The slash tells the preprocessor that the macro continues to the next line, not that the line is a continuation from a previous line.

Aside from readability, writing multi-line macros may make it more obvious that you need to use curly braces to surround the body because it’s more clear that multiple effects are happening at once.

**Advanced Macro Tricks**

In addition to simple substitution, the preprocessor can also perform a bit of extra work on macro arguments, such as turning them into strings or pasting them together.

**Pasting Tokens**

Each argument passed to a macro is a token, and sometimes it might be expedient to paste arguments together to form a new token. This could come in handy if you have a complicated structure and you’d like to debug your program by printing out different fields. Instead of writing out the whole structure each time, you might use a macro to pass in the field of the structure to print.

To paste tokens in a macro, use `##` between the two things to paste together.

For instance

```c
#define BUILD_FIELD(field) my_struct.inner_struct.union_a.##field
```
Now, when used with a particular field name, it will expand to something like

my_struct.inner_struct.union_a.field1
The tokens are literally pasted together.

String-izing Tokens
Another potentially useful macro option is to turn a token into a string containing the literal text of the token. This might be useful for printing out the token. The syntax is simple—simply prefix the token with a pound sign (#).

#define PRINT_TOKEN(token) printf(#token " is %d", token)
For instance, PRINT_TOKEN(foo) would expand to

printf("<foo>" " is %d" <foo>)
(Note that in C, string literals next to each other are concatenated, so something like "token" " is " " this " will effectively become "token is this". This can be useful for formatting printf statements.)

For instance, you might use it to print the value of an expression as well as the expression itself (for debugging purposes).

PRINT_TOKEN(x + y);

8.3 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif, and defined)

Six directives are available to control conditional compilation. They delimit blocks of program text that are compiled only if a specified condition is true. These directives can be nested. The program text within the blocks is arbitrary and may consist of preprocessor directives, C statements, and so on. The beginning of the block of program text is marked by one of three directives:

- #if
- #ifdef
- #ifndef

Optionally, an alternative block of text can be set aside with one of two directives:

- #else
- #elif

The end of the block or alternative block is marked by the #endif directive.

If the condition checked by #if, #ifdef, or #ifndef is true (nonzero), then all lines between the matching #else (or #elif) and an #endif directive, if present, are ignored.
If the condition is false (0), then the lines between the #if, #ifdef, or #ifndef and an #else, #elif, or #endif directive are ignored.

### 8.3.1 The #if Directive

The #if directive has the following syntax:

```
#if constant-expression newline
```

This directive checks whether the `constant-expression` is true (nonzero). The operand must be a constant integer expression that does not contain any increment (++) or decrement (--), `sizeof`, pointer (*), address (&), and cast operators.

Identifiers in the constant expression either are or are not macro names. There are no keywords, enumeration constants, and so on. The constant expression can also include the defined preprocessing operator (see Section 8.2.7).

The constant expression in an #if directive is subject to text replacement and can contain references to identifiers defined in previous `#define` directives. The replacement occurs before the expression is evaluated. Each preprocessing token that remains after all macro replacements have occurred is in the lexical form of a token.

If an identifier used in the expression is not currently defined, the compiler treats the identifier as though it were the constant zero.

### 8.3.2 The #ifdef Directive

The #ifdef directive has the following syntax:

```
#ifdef identifier newline
```

This directive checks whether the identifier is currently defined. Identifiers can be defined by a `#define` directive or on the command line. If such identifiers have not been subsequently undefined, they are considered currently defined.

### 8.3.3 The #ifndef Directive

The #ifndef directive has the following syntax:

```
#ifndef identifier newline
```

This directive checks to see if the identifier is not currently defined.

### 8.3.4 The #else Directive

The #else directive has the following syntax:
#else newline

This directive delimits alternative source text to be compiled if the condition tested for in the corresponding #if, #ifdef, or ifndef directive is false. An #else directive is optional.

### 8.3.5 The #elif Directive

The #elif directive has the following syntax:

```c
#elif constant-expression newline
```

The #elif directive performs a task similar to the combined use of the else-if statements in C. This directive delimits alternative source lines to be compiled if the constant expression in the corresponding #if, #ifdef, #ifndef, or another #elif directive is false and if the additional constant expression presented in the #elif line is true. An #elif directive is optional.

### 8.3.6 The #endif Directive

The #endif directive has the following syntax:

```c
#endif newline
```

This directive ends the scope of the #if, #ifdef, #ifndef, #else, or #elif directive.

The number of necessary #endif directives changes according to whether the elif or else directive is used. Consider the following equivalent examples:

```c
#if true                             #if true
 .                                  .
 .                                  .
 .                                  .
 #elif true                           .
 .                                  #else
 .                                  #if false
 .                                  .
 #endif                               .
 .                                  #endif
 #endif
```

### 8.3.7 The defined Operator

Another way to verify that a macro is defined is to use the defined unary operator. The defined operator has one of the following forms:

```c
defined name
defined (name)
```

An expression of this form evaluates to 1 if name is defined and to 0 if it is not.
The defined operator is especially useful for checking many macros with just a single use of the #if directive. In this way, you can check for macro definitions in one concise line without having to use many #ifdef or #ifndef directives.

For example, consider the following macro checks:

```c
#ifdef macro1
    printf( "Hello!\n" );
#endif
```

```c
#ifndef macro2
    printf( "Hello!\n" );
#endif
```

```c
#ifdef macro3
    printf( "Hello!\n" );
#endif
```

Another use of the defined operator is in a single #if directive to perform similar macro checks:

```c
#if defined (macro1) || !defined (macro2) || defined (macro3)
    printf( "Hello!\n" );
#endif
```

Note that defined operators can be combined in any logical expression using the C logical operators. However, defined can only be used in the evaluated expression of an #if or #elif preprocessor directive.

### C Preprocessor directives:

- Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.
- Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol.
- Below is the list of preprocessor directives that C language offers.

<table>
<thead>
<tr>
<th>S.no</th>
<th>Preprocessor</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Macro</td>
<td><code>#define</code></td>
<td>This macro defines constant value and can be any of the basic data types.</td>
</tr>
<tr>
<td>2</td>
<td>Header file inclusion</td>
<td><code>#include &lt;file_name&gt;</code></td>
<td>The source code of the file &quot;file_name&quot; is included in the main program at the specified place</td>
</tr>
<tr>
<td>3</td>
<td>Conditional compilation</td>
<td><code>#ifdef, #endif, #if, #else, #ifndef</code></td>
<td>Set of commands are included or excluded in source program before compilation with respect to the condition</td>
</tr>
<tr>
<td>4</td>
<td>Other directives</td>
<td><code>#undef, #pragma</code></td>
<td><code>#undef</code> is used to undefine a defined macro variable.</td>
</tr>
</tbody>
</table>
A program in C language involves into different processes. Below diagram will help you to understand all the processes that a C program comes across.

Example program for \#define, \#include preprocessors in C:

- \#define - This macro defines constant value and can be any of the basic data types.
- \#include <file_name> - The source code of the file “file_name” is included in the main C program where “\#include <file_name>” is mentioned.
```c
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'

void main()
{
    printf("value of height : %d \n", height);
    printf("value of number : %f \n", number);
    printf("value of letter : %c \n", letter);
    printf("value of letter_sequence : %s \n", letter_sequence);
    printf("value of backslash_char : %c \n", backslash_char);
}
```

**Output:**

- value of height: 100
- value of number: 3.140000
- value of letter: A
- value of letter_sequence: ABC
- value of backslash_char: ?

**Example program for conditional compilation directives:**

a) **Example program for #ifdef, #else and #endif in C:**

- "#ifdef" directive checks whether particular macro is defined or not. If it is defined, "if" clause statements are included in source file.
- Otherwise, "else" clause statements are included in source file for compilation and execution.
b) Example program for \#ifndef and \#endif in C:

- \#ifndef exactly acts as reverse as \#ifdef directive. If particular macro is not defined, "if" clause statements are included in source file.
- Otherwise, else clause statements are included in source file for compilation and execution.
```c
#include <stdio.h>
#define RAJU 100
int main()
{
    ifndef SELVA
    {
        printf("SELVA is not defined. So, now we are going to "
           "define here\n");
        #define SELVA 300
    }
    else
    printf("SELVA is already defined in the program");
    #endif
    return 0;
}
```

Output:

```
SELVA is not defined. So, now we are going to define here
```

**Example program for undefined in C:**

This directive undefines existing macro in the program.

```c
#include <stdio.h>
#define height 100
void main()
{
    printf("First defined value for height : %d\n",height);
    #undef height // undefining variable
    #define height 600 // redefining the same for new value
    printf("value of height after undef \& redefine:%d",height);
}
```

Output:

```
First defined value for height : 100
value of height after undef \& redefine : 600
```
Example program for pragma in C:

Pragma is used to call a function before and after main function in a C program.

```c
#include <stdio.h>

void function1( );
void function2( );

#pragma startup function1
#pragma exit function2

int main( )
{
    printf ( "\n Now we are in main function" );
    return 0;
}

void function1( )
{
    printf("\nFunction1 is called before main function call");
}

void function2( )
{
    printf ( "\nFunction2 is called just before end of " \n "main function" );
}
```

Output:

Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function
More on pragma directive in C:

<table>
<thead>
<tr>
<th>S.no</th>
<th>Pragma command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>#Pragma startup</td>
<td>This directive executes function named</td>
</tr>
<tr>
<td></td>
<td>&lt;function_name_1&gt;</td>
<td>&quot;function_name_1&quot; before</td>
</tr>
<tr>
<td>2</td>
<td>#Pragma exit</td>
<td>This directive executes function named</td>
</tr>
<tr>
<td></td>
<td>&lt;function_name_2&gt;</td>
<td>&quot;function_name_2&quot; just before termination of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>program.</td>
</tr>
<tr>
<td>3</td>
<td>#pragma warn - nvl</td>
<td>If function doesn’t return a value, then warnings</td>
</tr>
<tr>
<td></td>
<td></td>
<td>are suppressed by this directive while compiling.</td>
</tr>
<tr>
<td>4</td>
<td>#pragma warn - par</td>
<td>If function doesn’t use passed function parameter,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>then warnings are suppressed</td>
</tr>
<tr>
<td>5</td>
<td>#pragma warn - rch</td>
<td>If a non reachable code is written inside a program,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>such warnings are suppressed by this directive.</td>
</tr>
</tbody>
</table>