

Module 1

STAGES OF PROBLEM SOLVING

1. Understand the problem
2. Define the problem
 - Given(s), goal, ownership, resources and constraints
 - Given(s) - the initial situation
 - Goal - Desired target situation
 - Ownership -who does what
 - Resources and constraints - tools, knowledge, skills, materials and rules, regulations, guidelines, boundaries, timings.
3. Define boundaries.
4. Plan solution.
5. Check solution

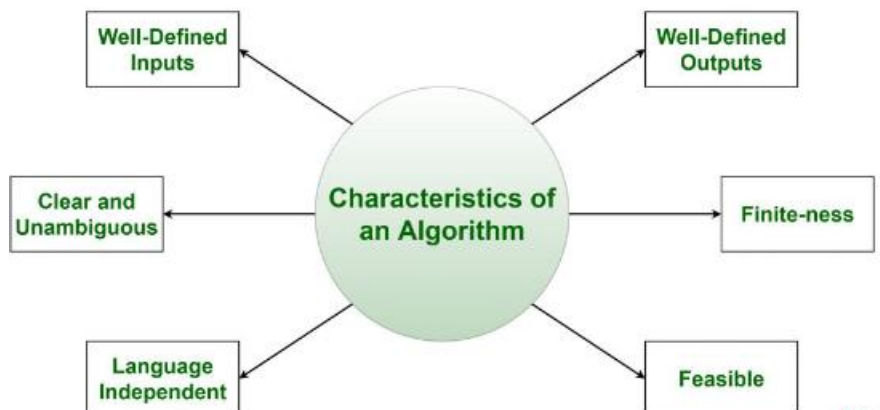
ALGORITHM

The algorithm is a step-by-step procedure which helps solve a problem. If it is written in English-like sentences then, it is called a 'PSEUDO CODE'.

An algorithm must possess the following five properties –

- Input
- Output
- Finiteness
- Definiteness
- Effectiveness

Characteristics of an Algorithm



- Clear and Unambiguous: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- Well-Defined Inputs: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- Well-Defined Outputs: The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- Finite-ness: The algorithm must be finite, i.e. it should terminate after a finite time.
- Feasible: The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- Language Independent: The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected
- Input: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.
- Output: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- Definiteness: All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- Finiteness: An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- Effectiveness: An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Algorithm to print the first 10 natural numbers is as follows –

Step 1: [initialize] Set $I = 1$, $N = 10$

Step 2: Repeat Steps 3 and 4 while $I \leq N$

Step 3: Print I

Step 4: SET $I = I + 1$

[END OF LOOP]

Step 5: End

FLOWCHART

Flowchart is a graphical representation of an algorithm. Programmers often use it as a program-planning tool to solve a problem. It makes use of symbols which are connected among them to indicate the flow of information and processing. The process of drawing a flowchart for an algorithm is known as “flowcharting”.

Rules For Creating Flowchart :




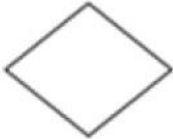
A flowchart is a graphical representation of an algorithm. It should follow some rules while creating a flowchart.


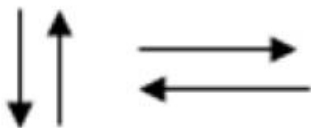

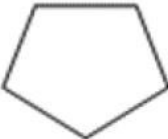


Rule 1: Flowchart opening statement must be ‘start’ keyword.

Rule 2: Flowchart ending statement must be ‘end’ keyword.

Rule 3: All symbols in the flowchart must be connected with an arrow line.

Rule 4: The decision symbol in the flowchart is associated with the arrow line.

Name	Symbol	Purpose
Terminal	 oval	start/stop/begin/end
Input/output	 Parallelogram	Input/output of data
Process	 Rectangle	Any processing to be performed can be represented
Decision box	 Diamond	Decision operation that determine which of the alternative paths to be followed

Connector	 <p>Circle</p>	Used to connect different parts of flowchart
Flow	 <p>Arrows</p>	Join 2 symbols and also represents flow of execution
Predefined process	 <p>Double sided rectangle</p>	Module (or) subroutines specified else where
Page connector	 <p>Pentagon</p>	Used to connect flowchart in 2 different pages
For loop symbol	 <p>Hexagon</p>	Shows initialization, condition and incrementation of loop variable
Document	 <p>Printout</p>	Shows the data that is ready for printout

Advantages of Flowchart:

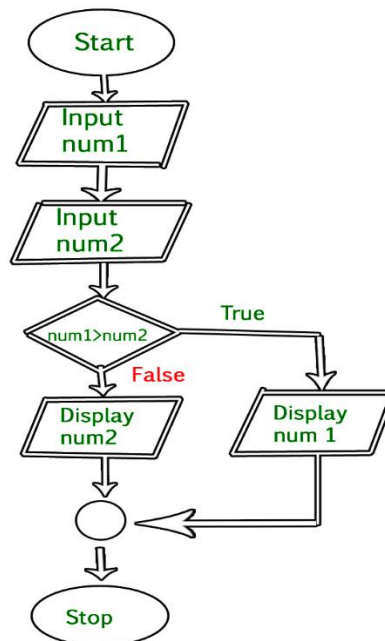
- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.

- It provides better documentation.
- Flowcharts serve as a good proper documentation.
- Easy to trace errors in the software.
- Easy to understand.
- The flowchart can be reused for inconvenience in the future.
- It helps to provide correct logic.

Disadvantages of Flowchart:

- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- Difficult to reproduce the flowcharts.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- Some developer thinks that it is waste of time.
- It makes software processes low.
- If changes are done in software, then the flowchart must be redrawn

Example : Draw a flowchart to input two numbers from the user and display the largest of two numbers



MODULAR APPROACH IN PROGRAMMING

- Definition: The modular approach involves breaking a program into smaller, manageable pieces (modules), each of which can be developed and tested independently.

- Characteristics:
 - Each module performs a specific task and can be reused in other programs.
 - Modules are loosely coupled and highly cohesive, meaning they focus on one responsibility and don't heavily depend on each other.
 - Examples in C are functions and libraries that can be independently tested and reused in various contexts.
- Advantages:
 - Easier debugging and maintenance.
 - Code reusability.
 - Improved readability and collaboration in teams.

TOP-DOWN AND BOTTOM-UP APPROACHES IN C PROGRAMMING

In software design and development, Top-Down and Bottom-Up are two different approaches for designing, developing, and debugging programs. These approaches reflect how a program or system is built, structured, and organized.

Top-Down Approach

Definition: The top-down approach involves breaking down a complex system or problem into smaller, manageable sub-problems or modules. The system is designed from a high-level overview (the "top") and moves downward by progressively refining each part into more detail until the entire system is specified.

Key Points:

- **Focus:** Starts with the overall system and breaks it into smaller components.
- **Modular Design:** The system is divided into modules or functions. Each module is further subdivided into smaller sub-modules until they become simple enough to be implemented directly.
- **Main Function First:** The design begins by defining the main function (or the highest-level module), and then sub-functions or sub-modules are designed.
- **Control Flow:** The main control and logic flow is outlined first, and details are added progressively.
- **Advantages:**
 - The overall structure of the system is clear from the beginning.
 - Easier to track the control flow and dependencies between different parts of the program.
 - Ideal for solving large, complex problems where a systematic decomposition is needed.

- Disadvantages:
 - May lead to delays in development since lower-level details are worked on later.
 - Revisions can be more time-consuming if higher-level design needs to change after lower-level modules have been implemented.

Example: In a top-down approach, you would first define the main functionality of the program and then work down to the specific details. For instance:

```
#include <stdio.h>

void getInput();           // Function prototypes
void processData();
void outputResult();

int main() {
    getInput();           // Call high-level functions in the
                           // main program
    processData();
    outputResult();
    return 0;
}

void getInput() {
    // Get user input
}

void processData() {
    // Process the input data
}

void outputResult() {
    // Display the results
}
```

In this example, the overall structure is clear from the beginning, but the details of `getInput()`, `processData()`, and `outputResult()` are filled in later.

2. Bottom-Up Approach

Definition: The bottom-up approach starts by designing and building the most basic or low-level components of the system first. These components are then integrated progressively to form more complex, higher-level modules or systems until the whole program is complete.

Key Points:

- Focus: Starts from the lowest-level details (e.g., functions or modules) and builds upwards.
- Reusable Components: The bottom-up approach encourages the creation of reusable functions or modules that can be combined to solve more complex problems.
- Function-Driven: Often focuses on writing and testing independent functions or libraries first, which can then be integrated into the larger system.
- Integration: Higher-level functionalities are built by combining and linking pre-built, tested modules.
- Advantages:
 - Easier to test and validate smaller, independent components early on.
 - Encourages code reuse and modular design.
 - Changes in lower-level components have minimal effect on the overall structure.
- Disadvantages:
 - The overall system design may be unclear initially, leading to potential integration challenges.
 - Requires a good understanding of how different components will interact to ensure seamless integration.

Example: In a bottom-up approach, you would start by defining the lower-level functions, test them, and then build up toward the main function. For instance:

```
#include <stdio.h>

// Lower-level function to add two numbers
int add(int a, int b) {
    return a + b;
}

// Lower-level function to multiply two numbers
int multiply(int a, int b) {
    return a * b;
}

// Main function
int main() {
    int sum = add(5, 3);           // Use the pre-defined
                                   add() function
    int product = multiply(5, 3); // Use the pre-defined
                                   multiply() function

    printf("Sum: %d\n", sum);
    printf("Product: %d\n", product);

    return 0;
}
```


Here, the add() and multiply() functions are developed and tested first, and then these are integrated into the main() function.

Comparison of Top-Down and Bottom-Up Approaches

S.No.	Top-Down Approach	Bottom-Up Approach
1.	In this approach, the problem is broken down into smaller parts.	In this approach, the smaller problems are solved.
2.	It is generally used by structured programming languages such as C, COBOL, FORTRAN, etc.	It is generally used with object oriented programming paradigm such as C++, Java, Python, etc.
3.	It is generally used with documentation of module and debugging code.	It is generally used in testing modules.
4.	It does not require communication between modules.	It requires relatively more communication between modules.
5.	It contains redundant information.	It does not contain redundant information.
6.	Decomposition approach is used here.	Composition approach is used here.
7.	The implementation depends on the programming language and platform.	Data encapsulation and data hiding is implemented in this approach.

CHARACTERISTICS OF A GOOD PROGRAM

The characteristics of a good program are as follows :

- **Flexibility:** It should be possible to accommodate modifications in the program without having the requirement to write the entire code again. The program should be able to serve multiple purposes. For instance, CAD (Computer Aided Design) software can efficiently be used to serve diverse fields like engineering drafting, industrial art, fabric designing, printing circuit layout and designing, architectural drawing etc. Most

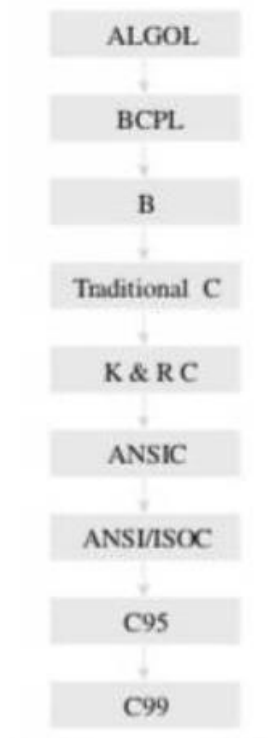
softwares are developed for a particular period and updations are required to make the software adaptable for serving new purposes as well.

- **User Friendly:** User friendly programs are those programs which can easily be understood by a beginner. The interaction between user and program should be easy to understand. The commands in the program like to input data, to print result, etc. must easily be comprehensible by the naive user.
- **Portability:** The ability of a program to run on different operating systems without having any or minimum changes is termed as the portability of the program or application. With the rise of numerous platforms having different operating systems embedded within them, and with the availability of various hardware and softwares, portability is something which is being taken care of by every application developer these days. Without portability, a program highly usable for one platform becomes obsolete for another. Program developed in high level languages are generally more portable than programs developed in assembly level languages.
- **Reliability:** Reliability of a program refers to its ability to perform its intended function accurately along with certain modifications. The ability of a program to handle exceptions of wrong input or no input is one of the fundamental behaviors which define reliability. Although, the program is not expected to produce correct result in these exceptions but it should be able enough to provide an error message. A program equipped with such a characteristic is called a reliable program.
- **Self-Documenting Code:** A self-documenting code is the source code, in which suitable names for the identifiers are being used. Proper names for the identifiers makes it easier for the user to understand the functionality of a particular variable or consonant (identifier). Hence, it is advisable that every good program must have a self-documenting code.

EVOLUTION OF C PROGRAMMING

C is derived from ALGOL, the first language to use a block structure. ALGOL's introduction in the 1960s led the way for the development of structured programming, concepts. In 1967 Martin Richards developed a language called BCPL (Basic Combined Programming Language). BCPL was a type-less (had no concept of data types) language which facilitated direct access of memory. This made it useful for system programmers. Then in 1970, Ken Thompson developed a language called B, which was used to develop the first version of UNIX. C was developed by Dennis Ritchie in 1972 that took concepts from ALGOL, BCPL, and B. In addition to the concepts of these languages, C also supports the concept of data types. Since UNIX operating system was also developed at Bell Laboratories along with C language, C and UNIX are strongly associated with each other.

C (also known as Traditional C) was documented and popularized in the book *The C Programming Language* by Brian W. Kernighan and Dennis Ritchie in 1978. In 1983, the American National Standards Institute (ANSI) started working on defining the standard for C. This standard was approved in December 1989 and came to be known as ANSI C. In 1990, the International Standards Organization (ISO) adopted the ANSI standard. This version of C came to be known as C89. In 1995, some minor changes were made to C89, the new modified version was known as C95. During the 1990s, C++ and Java programming languages became popular among users so the Standardization Committee of C felt that a few features of C++/Java, if added to C, would enhance its usefulness. So, in 1999 when some significant changes were made to C95, the modified version came to be known as C99.



FEATURE OF C

c is a PL (procedural language) not OOP

C is a robust language whose rich set of built-in functions and operators can be used to write complex programs. The C compiler combines the features of assembly languages and high-level languages, which makes it best suited for writing system software as well as business packages. Some basic characteristics of C language that define the language and have led to its popularity as a programming language are listed below.

- C is a high-level programming language, which enables the programmer to concentrate on the problem at hand and not worry about the machine code on which the program would be run.
- Small size—C has only 32 keywords. This makes it relatively easy to learn as compared to other languages.
- C makes extensive use of function calls.
- C is well suited for structured programming. In this programming approach, C enables users to think of a problem in terms of functions/modules where the collection of all the modules

makes up a complete program. This feature facilitates ease in program debugging, testing, and maintenance.

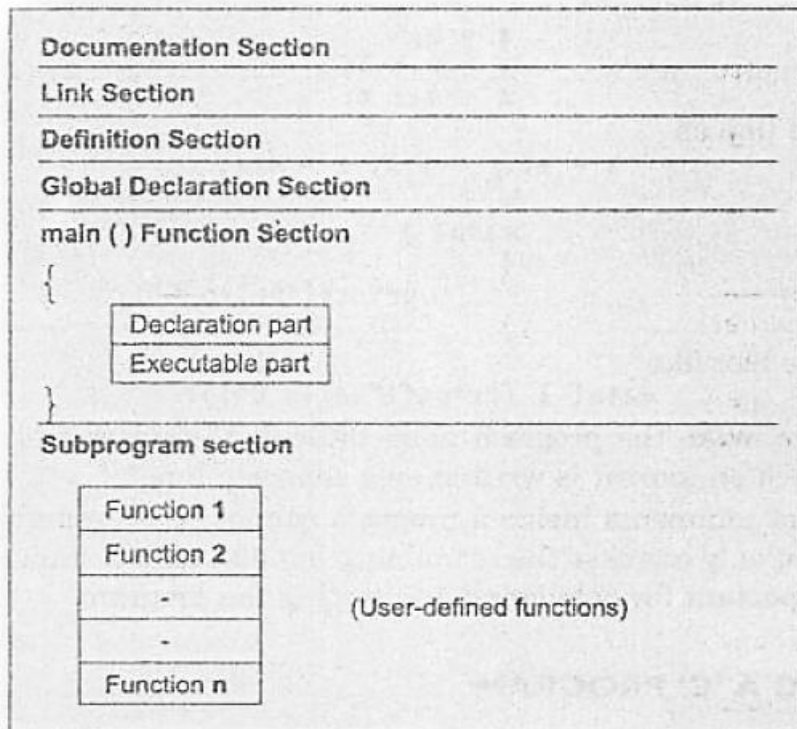
- Unlike PASCAL it supports loose typing (as a character can be treated as an integer and vice versa).
- Structured language as the code can be organized as a collection of one or more functions.
- Stable language. ANSI C was created in 1983 and since then it has not been revised.
- Quick language as a well-written C program is likely to be as quick as or quicker than a program written in any other language. Since C programs make use of operators and data types, they are fast and efficient.

For example: a program written to increment a value from 0-15000 using BASIC would take 50 seconds whereas a C program would do the same in just 1 second.

- Facilitates low-level (bitwise) programming.
- Supports pointers to refer to computer memory. arrays, structures, and functions.
- Core language. C is a core language as many other programming languages (like C++, Java, Perl, etc.) are based on C. If you know C, learning other computer languages becomes much easier.
- C is a portable language, i.e., a C program written for one computer can be run on another computer with little or no modification.
- C is an extensible language as it enables the user to add his own functions to the C library.
- C is often treated as the second-best language for any given programming task. While the best language depends on the particular task to be performed, the second best language, on the other hand, will always be C.

BASIC STRUCTURE OF C PROGRAMS

A C program can be viewed as a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. A C program may contain one or more sections as shown in Figure below.



The *documentation section* consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.

The *link section* provides instructions to the compiler to link functions from the system library. The *definition section* defines all symbolic constants.

There are some variables that are used in more than one function. Such variables are called global variables and are declared in the *global declaration section* that is outside of all the functions. This section also declares all the user-defined functions.

Every C program must have one *main() function section*. This section contains two parts, declaration part and executable part. The declaration part declares all the variables used in the executable part. There is at least one statement in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function section is the logical end of the program. All statements in the declaration and executable parts end 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 sections, except the main function section, may be absent when they are not required.

PROGRAMMING STYLE

Unlike some other programming languages (COBOL, FORTRAN, etc.,) C is a free-form language. That is, the C compiler does not care, where on the line we begin typing. While this may be a licence for bad programming, we should try to use this fact to our advantage in developing readable programs. Although several alternative styles are possible, we should select one style and use it with total consistency.

First of all, we must develop the habit of writing programs in lowercase letters. C program statements are written in lowercase letters. Uppercase letters are used only for symbolic constants.

Braces, group program statements together and mark the beginning and the end of functions. A proper indentation of braces and statements would make a program easier to read and debug. Note how the braces are aligned and the statements are indented in the program of Fig. 1.5. Since C is a **free-form language**, we can group statements together on one line. The statements

```
a = b;  
x = y + 1;  
z = a + x;
```

can be written on one line as

```
a = b; x = y + 1; z = a + x;
```

The program

```
main()  
{  
    printf("hello C");  
}
```

may be written in one line like

```
main() {printf("Hello C");}
```

However, this style make the program more difficult to understand and should not be used. In this book, each statement is written on a separate line.

The generous use of comments inside a program cannot be overemphasized. Judiciously inserted comments not only increase the readability but also help to understand the program logic. This is very important for debugging and testing the program.

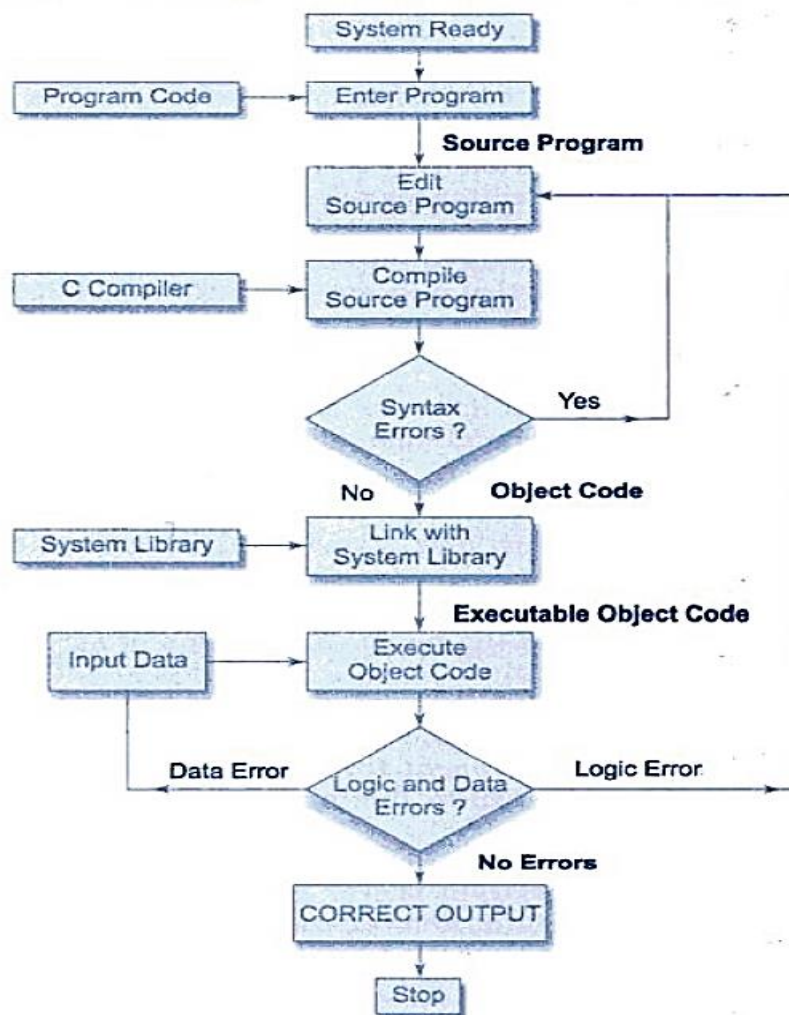
DOCUMENTATION AND MAINTENANCE

- *Documentation:*
 - Documentation involves writing descriptions and explanations for the code to clarify what it does, how it works, and how it can be used or modified.
 - It can be internal (within code, using comments) or external (separate documents or manuals).
 - Proper documentation includes:
 - Descriptions of functions and modules.
 - Explanation of parameters, return values, and algorithms used.
 - Usage examples.
- *Maintenance:*
 - Maintenance is the process of updating and improving a program after its initial deployment. This includes bug fixes, performance improvements, or adapting the program to new environments or requirements.
 - Types of maintenance:
 - Corrective: Fixing bugs or errors.
 - Adaptive: Adjusting the program to new operating environments or technologies.
 - Perfective: Improving performance or enhancing features.
 - Preventive: Updating the code to prevent future issues, like improving security.

EXECUTING A C PROGRAM

Executing a program written in C involves a series of steps. These are:

1. Creating the program;
2. Compiling the program;
3. Linking the program with functions that are needed from the C library; and
4. Executing the program.



The figure above illustrates the process of creating, compiling and executing a C program. Although these steps remain the same irrespective of the operating system, system commands for implementing the steps and conventions for naming files may differ on different systems.

An operating system is a program that controls the entire operation of a computer system. All input/output operations are channelled through the operating system. The operating system, which is an interface between the hardware and the user, handles the execution of user programs.

The following section explains the procedure to be followed in executing C program under the UNIX operating system.

Creating the program

Once we load the UNIX operating system into the memory, the computer is ready to receive program. The program must be entered into a file. The file name can consist of letters, digits and special characters, followed by a dot and a letter e. Examples of valid file names are:

hello.c

program.c

ebgl.c

The file is created with the help of a text editor, either ed or vi. The command for calling the editor and creating the file is

ed filename

If the file existed before, it is loaded. If it does not yet exist, the file has to be created so that it is ready to receive the new program. Any corrections in the program are done under the editor. (The name of your system's editor may be different. Check your system manual.) When the editing is over, the file is saved on disk. It can then be referenced any time later by its file name. The program that is entered into the file is known as the **source program**, since it represents the original form of the program.

Compiling and Linking

Let us assume that the source program has been created in a file named ebgl.c. Now the program is ready for compilation. The compilation command to achieve this task under UNIX is

cc ebgl.c

The source program instructions are now translated into a form that is suitable for execution by the computer. The translation is done after examining each instruction for its correctness. If everything is alright, the compilation proceeds silently and the translated program is stored on another file with the name ebgl.o. This program is known as **object code**.

Linking is the process of putting together other program files and functions that are required by the program. For example, if the program is using exp() function, then the object code of this function should be brought from the math library of the system and linked to the main program.

Under UNIX, the linking is automatically done (if no errors are detected) when the `cc` command is used.

If any mistakes in the syntax and semantics of the language are discovered, they are listed out and the compilation process ends right there. The errors should be corrected in the source program with the help of the editor and the compilation is done again.

The compiled and linked program is called the **executable object code** and is stored automatically in another file named `a.out`. Note that some systems use different compilation command for linking mathematical functions.

`cc filename -lm`

is the command under UNIPLUS SYSTEM V operating system.

Executing the Program

Execution is a simple task. The command

`a.out`

would load the executable object code into the computer memory and execute the instructions. During execution, the program may request for some data to be entered through the keyboard. Sometimes the program does not produce the desired results. Perhaps, something is wrong with the program logic or data. Then it would be necessary to correct the source program or the data. In case the source program is modified, the entire process of compiling, linking and executing the program should be repeated.

Creating Your Own Executable File

Note that the linker always assigns the same name `a.out`. When we compile another program, this file will be overwritten by the executable object code of the new program. If we want to prevent from happening, we should rename the file immediately by using the command.

`mv a.out name`

We may also achieve this by specifying an option in the `cc` command as follows:

`cc -o name source-file`

This will store the executable object code in the file name and prevent the old file `a.out` from being destroyed.

Multiple Source Files

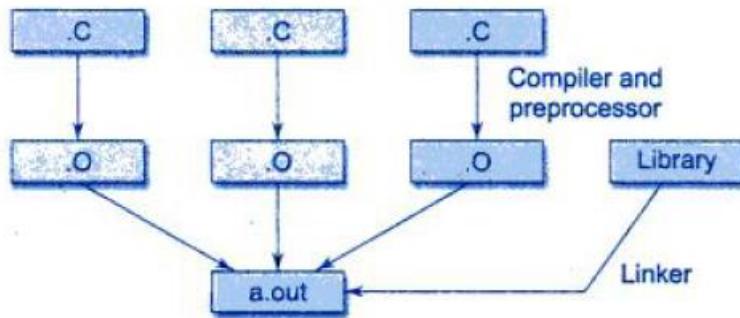
To compile and link multiple source program files, we must append all the files names to the `cc` command.

`cc filename-1.cfilename-n.c`

These files will be separately compiled into object files called

`filename -i.o`

and then linked to produce an executable program file `a.out` as shown



It is also possible to compile each file separately and link them later. For example, the commands

`cc -c mod1.c`

`cc -c mod2.c`

will compile the source files `mod1.c` and `mod2.c` into object files `mod1.o` and `mod2.o`. They can be linked together by the command

`cc mod1.o mod2.o`

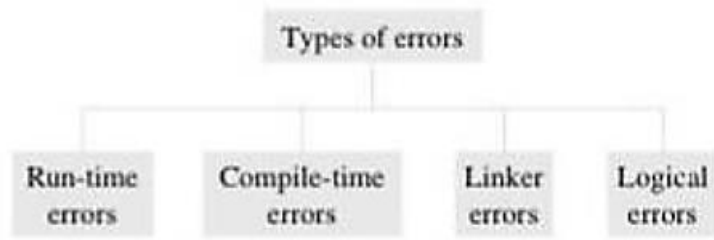
we may also combine the source files and object files as follows:

`cc mod1.c mod2.o`

Only `mod1.c` is compiled and then linked with the object file `mod2.o`. This approach is useful when one of the multiple source files need to be changed and recompiled or an already existing object files is to be used along with the program to be compiled.

TYPE OF ERRORS

While writing programs, very often we get errors in our programs. These errors if not removed will either give erroneous output or will not let the compiler to compile the program. These errors are broadly classified under four groups.



Run-time Errors :

Run-time errors occur when the program is being run executed. Such errors occur when the program performs some illegal operations like

- Dividing a number by zero
- Opening a file that already exists
- Lack of free memory space
- Finding square or logarithm of negative numbers

Run-time errors may terminate program execution, so the code must be written in such a way that it handles all sorts of unexpected errors rather terminating it unexpectedly.

Compile-time Errors :

Compile-time errors occur at the time of compilation of the program. Such errors can be further classified as follows:

Syntax Errors:

Syntax errors are generated when rules of a programming language are violated.

Semantic Errors:

Semantic errors are those errors which may comply with rules of the programming language but are not meaningful to the compiler.

Logical Errors:

Logical errors are errors in the program code that result in unexpected and undesirable output which is obviously not correct. Such errors are not detected by the compiler, and programmers must check their code line by line or use a debugger to locate and rectify the errors. Logical errors occur due to incorrect statements.

Linker Errors:

These errors occur when the linker is not able to find the function definition for a given prototype.