

Chapter 1

Problem Solving Through Computers & Introduction to C



Dr. Niroj Kumar Pani

nirojpani@gmail.com

Department of Computer Science Engineering & Applications

Indira Gandhi Institute of Technology

Sarang, Odisha

Chapter Outline...

- The Problem-Solving Approach
- Algorithms
- Programming
- Structured Programming
- Introduction to C
- The 1st C Program
- Writing, Building & Executing the 1st Program
- One More Program
- The Same Program Using User-Defined Function
- Structure of a C Program
- Assignments

The Problem-Solving Approach

Problem solving through computers involves the following steps:

1. Understanding / analyzing the problem

- As the first step in problem solving, we **try to understand the problem in total**, because if we don't understand the problem, we may end up developing a solution which may not solve our purpose.
- We **figure out** the **inputs** our solution should accept and the **outputs** our solution should produce.

Example:

Problem: Find the average of three numbers.

Inputs: 3 real numbers.

Output: 1 real number which is the average of the inputs.

2. Developing an algorithm

- Once we understand the problem, the next step is to **formulate a solution**; a **sequence of steps** that gives us the desired output.
- The **solution** is represented in human-readable language and is called **an algorithm (or, logic)**.

We discuss algorithms in the next section.

3. Programming / coding

- **Algorithms are for humans** (written in human-readable languages) not for computers.
- If we want to solve a problem through *computers*, **an algorithm must be translated to a program** (a language that the computers understand).
- Different programming languages can be used for this.

We discuss programming in detail later in this chapter.

4. Testing & Debugging

- The program created should be **tested** on various parameters such as
 - **Syntax:** The program should be free from programming language-related errors.
 - **Semantic / Logic:** The program should generate correct output for all possible inputs.
 - **Performance:** The program must give the desired output within the expected time.
- The errors found in the testing phases are **debugged** or rectified and the program is tested again. This process continues until all the errors are removed from the program.

[NOTE]: Software industry follows standardized testing methods like unit or component testing, integration testing, system testing, and acceptance testing while developing complex applications. This is to ensure that the software meets all the business and technical requirements and works as expected.

Algorithms

- An algorithm (logic) is independent of any programming language.
- **[Algorithm - Definition]:** An algorithm is a step-by-step, well defined, finite set of instructions, that if followed, accomplishes a particular task. An algorithm may or may not take inputs and produces a value or set of values as output.

Notable characteristics of an algorithm (observed from the above definition):

- **Step-by-Step:** The instructions are ordered.
- **Definiteness:** Each instruction is clear and unambiguous.
- **Finiteness:** The algorithm terminates after a finite number of steps.
- **Input:** Zero or more quantities as input.
- **Output:** At least one quantity as output.

Algorithm Specification (Representation)

- There are two common methods of representing an algorithm:

1. Pseudocode
2. Flowchart

- **Pseudocode**

- It is **English-like representation** of algorithms. It is part English, part structured code.
- Most common form of algorithm representation.
- **Pseudocode Example 1 (ALG1_1):** Algorithm to find the average of three numbers.

```
FindAverage (x, y, z)
```

```
/* This algorithm takes as input three numbers, computes their average and displays it. */
```

```
1. Set Sum = x+y+z /* 'Sum' is a real number that represents the sum of x, y, z */
```

```
2. Set Avg = Sum/3 /* 'Avg' is a real number that represents the average of x, y, z */
```

```
3. Write: Avg
```

- **Pseudocode Example 2 (ALG1_2):** Algorithm to compute the largest of three numbers.

```
FindLargest(x, y, z)
```

```
/* This algorithm takes as input 3 integers x, y and z. It finds the largest among them and displays the result. */
```

```
1. Set largest = x /* 'largest' represents the largest number */
```

```
2. If y > largest, then:
```

```
    Set largest = y
```

```
3. If z > largest, then:
```

```
    Set largest = z
```

```
4. Write: largest
```


- **Pseudocode Example 3 (ALG1_3):** Algorithm to compute the average of 'n' numbers.




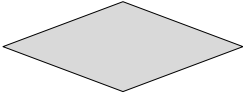
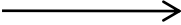

```
FindAverage (A, n)
```

```
/* This algorithm takes as input one array 'A' with 'n' elements, indexed from 0. It  
computes the average of the 'n' numbers in the array and displays it*/
```

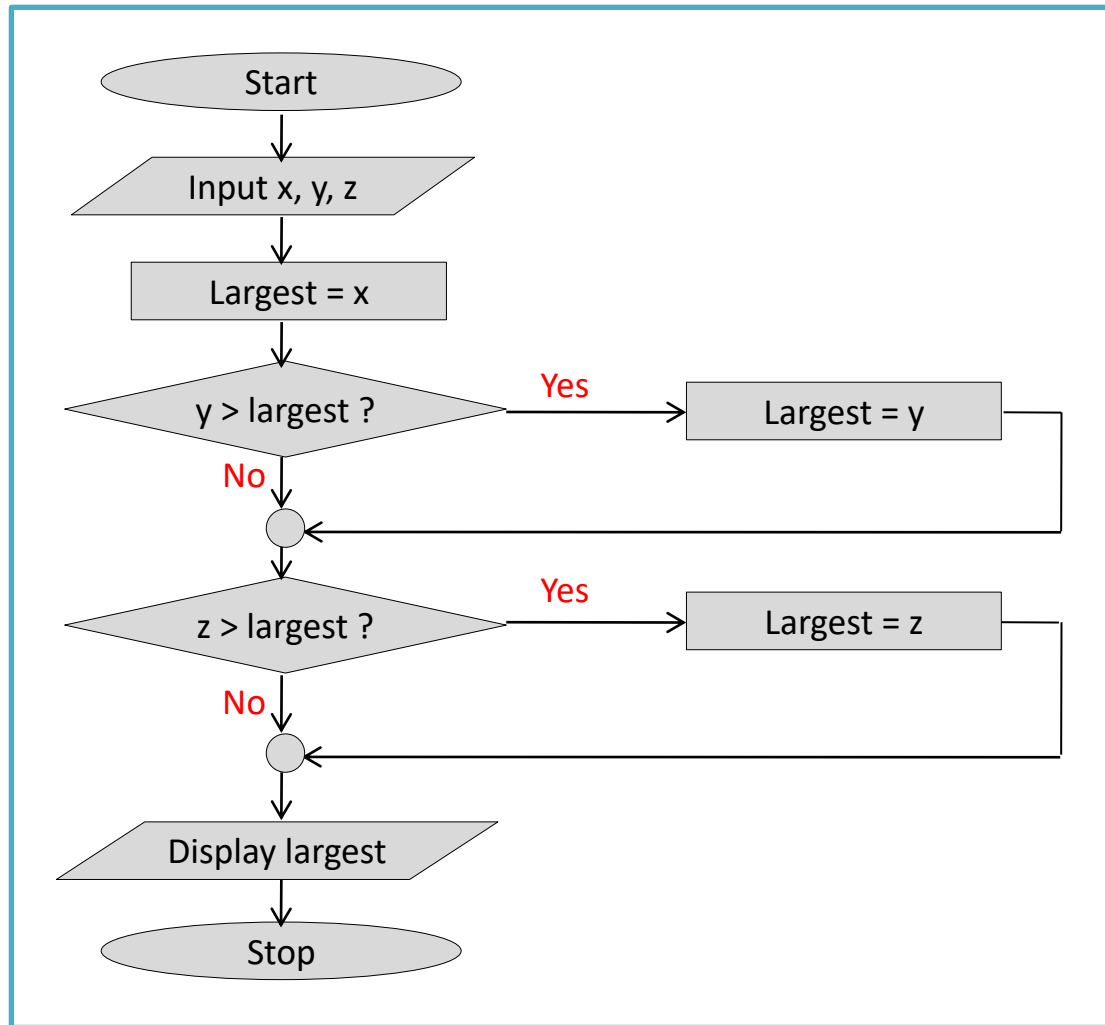
1. Set Sum = 0 /* 'Sum' is an integer. It represents the sum of all 'n' numbers*/
2. Set i = 0 /* 'i' is an integer. It is used as a counter variable */
3. While i < n, repeat steps 4 to 5
4. Set Sum = Sum + A[i]
5. Set i = i+1
6. Set Avg = Sum / n
7. Write: Avg

■ Flowcharts

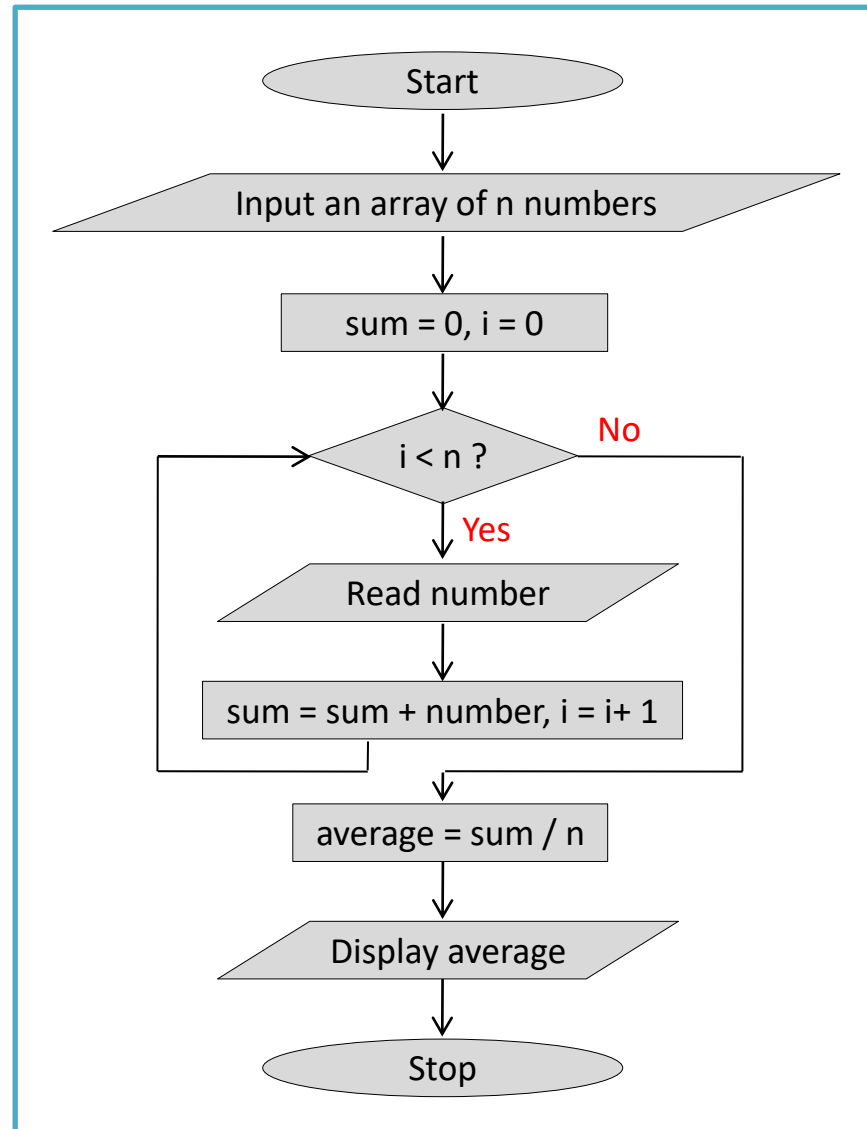
- A flow chart is a **pictorial / graphical representation** of an algorithm. It uses symbols to represent specific actions.
- **Most common symbols used in Flowcharts:**

Symbols	Use
	Start / Stop
	Input / Output
	Processing
	Condition
	Data flow
	Connector

■ **Flowchart Example 1 (FC1_1):** Compute the largest of three numbers.



- **Flowchart Example 2 (FC1_2):** Compute the average of 'n' numbers.



Control Flow

- The control flow depicts **the order in which the instructions** in an algorithm **are executed**.
- The control flow can be a
 - **Sequence:** All the instructions are executed one after another, starting from the 1st instruction to the last, e.g., [ALG1_1](#).
 - **Selection / Decision / Branch:** One set of instructions is executed if a particular condition becomes true and another set of instruction is executed if the condition becomes false, e.g., [ALG1_2](#).
 - Generally specified through an 'if' clause.
 - **Loop / Repetition / Iteration:** A set of instructions is executed repeatedly until a particular condition becomes false, e.g., [ALG1_3](#).
 - Generally specified through an 'while' clause.

Key Factors in Algorithm Development

- For a given problem, more than one algorithm is possible.

For example, in ALG1_1 the three instructions can be replaced by just one instruction: Write: $(x+y+z) / 3$.

Therefore, we must design the most suitable one.

- Developing a good algorithm depends upon many factors, such as
 - **Deployment of top-down approach:** Whether the problem should be solved in total or it should be decomposed into number of independent sub-problems, each of which is handled separately.
 - **Choosing the right design technique:** Some of the most popular algorithm design techniques are:
 - Divide-and-conquer
 - Greedy method
 - Dynamic programming
 - Backtracking
 - Branch-and-bound

- **Choice of suitable data structure:** Which data structure would be most suitable for the operations:
 - Normal variables
 - Arrays
 - Stacks
 - Queues
 - Trees
 - Graphs
- **Deciding the flow of control:** Whether the flow of control at different parts of the algorithm would be in a
 - Sequence
 - Branch / selection / decision
 - Loop / repetition / iteration

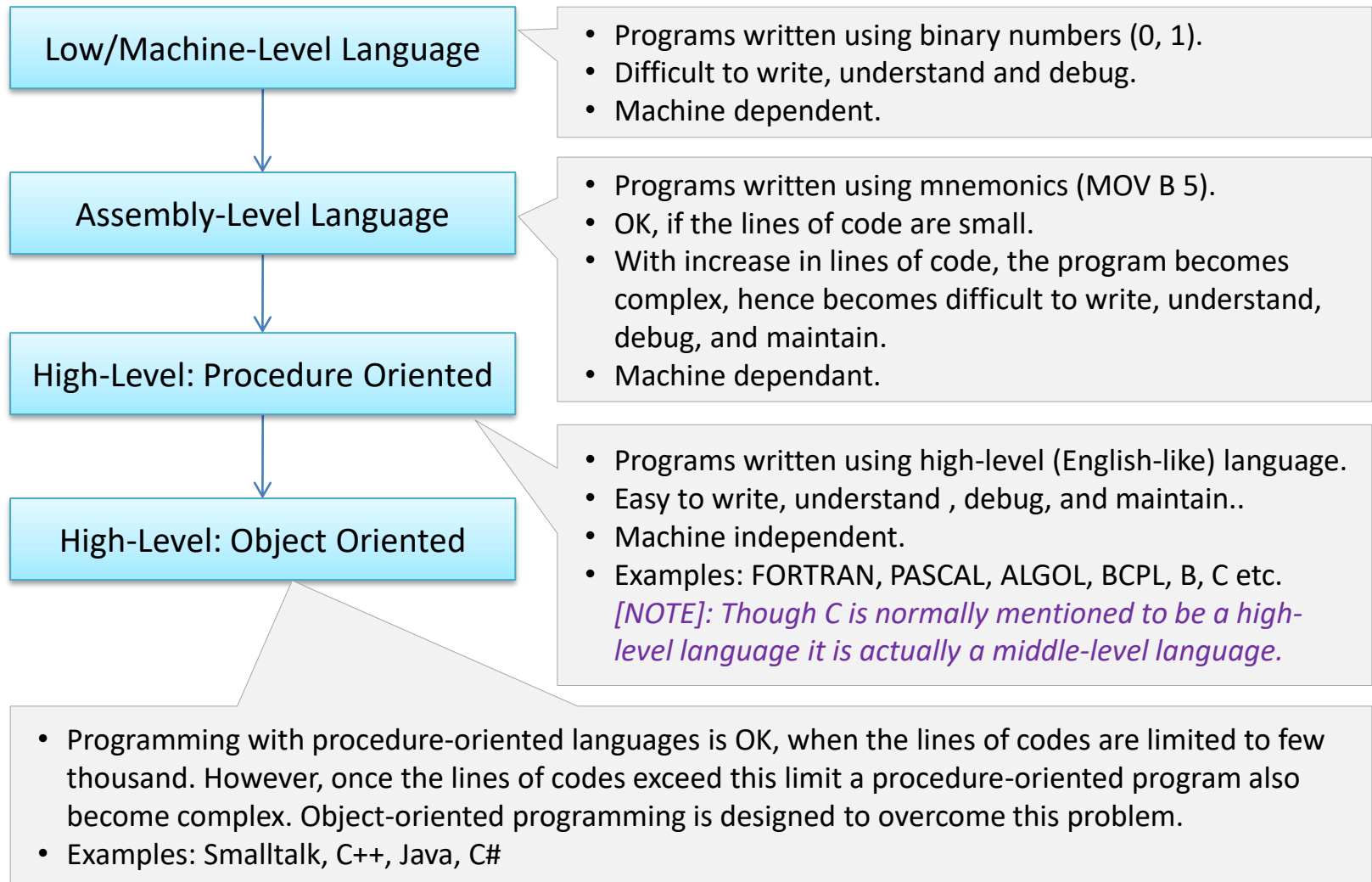
Programming

- **Programming:** It is the art / technique of writing programs (good programs).
- **A Program:** A program is nothing, but an algorithm (logic) written according to the syntax of a specific programming language.

Program = Algorithm + Syntax

[Formal Definition]: A program is a step-by-step, well defined, finite set of instructions, **written according to the syntax of a specific programming language**, that if followed, accomplishes a particular task. A program may or may not take inputs and produces a value or set of values as output.

Evolution of Programming Paradigms



Assemblers, Compilers, and Interpreters

A computer understands only 0 and 1 (machine-level language). Therefore, the codes written in assembly-level or high-level languages need to be converted (translated) to equivalent binary forms (0, 1), so that the computer can understand and execute them.

- **Assembler:** An assembler is a software that translates assembly-level language code to equivalent machine-level language code. This translation process (from assembly-level to machine-level) is called **assembling**.
- **Compiler:** A compiler is a software that translates high-level language code to equivalent machine-level language code. This translation process (from high-level to machine-level) is called **compilation**.

[NOTE]: Some old compilers produces assembly-level language as output, which in turn should be passed through an assembler to produce the desired machine-level codes (Most of the modern compilers - all C compilers - produce machine-level language as output.)

- **Interpreter:** Many high-level languages use another software, called the interpreter, to translate high-level language code to equivalent machine-level language code. The translation process (through an interpreter) is called **interpretation**.

Though, both compiler and interpreter do the same thing, there are some differences between these two (given in next slide).

Compiler	Interpreter
<ul style="list-style-type: none"> • A compiler takes as input the entire program (source code file) and scans it. If there is any error in any part of the entire program, it displays the list of errors without translating a single instruction. Once all the errors, in the entire program are corrected (debugged), the source code file as a whole is translated into machine-level code and is stored in a separate file, called the object file. • Since the compiler shows the list of errors only after scanning the entire source code, debugging is comparatively hard. • It takes large amount of time to analyze & translate the source code, but the overall execution time is comparatively less. • Programming language like C, C++ use compilers 	<ul style="list-style-type: none"> • An interpreter takes as input one instruction at a time and scans it. If there is any error in <i>that</i> instruction, it displays these list of errors without translating <i>that</i> instruction. Once all the errors in <i>that</i> instruction are corrected (debugged), it translates <i>that</i> instruction to machine-level code, and moves to the next instruction. This process continues until the program ends. In this case no object file is created. • Since the errors are displayed on the go per instruction, debugging is comparatively easy. • It takes less amount of time to analyze the source code, but the overall execution time is high (as no object file is created). • Programming language like Python, Ruby use interpreters.

Structured Programming

- Structured programming refers to the methodology of writing a program which is structured - *meaning that* - the program is made of **well-defined blocks** (commonly called, functions or methods), and **each block has the single-entry and single-exit property**.
- The single-entry and single-exit property helps in reducing the number of paths of flow of control, which makes the program simple to design and understand.
 - If there are arbitrary paths for flow of control, the program will be difficult to write, understand, debug, and maintain.
- **Examples:** C, C++, Java, C# are all examples of structured programming.

Introduction to C

History of C



■ **ALGOL (Algorithmic Language):**

- Introduced in 1960, by a group of international scientists.
- Mother of all high-level languages.
- 1st gave the concept of structured programming.

■ **BCPL (Basic Combined Programming Language):**

- Developed by Martin Richards in 1967.
- Primarily designed for writing system software.

■ **B:**

- Created by Ken Thompson in 1970.
- Used the features of BCPL.
- Used to write the early version of UNIX in the Bell laboratories.

■ **C:**

- Created by Denis Ritchie at Bell laboratories in 1972.
- He used the features of ALGOL, BCPL, and B and added the concept of data types and some more powerful features to formulate this language.

- **UNIX is almost entirely coded using C.**
- **Until 1978, mainly used in academic environment.**
- **Became very popular when Brian Kernighan and Denis Ritchie, in 1978, described this language in his book “The C Programming Language”.**
- **This original form of C is also known as “K&R C”.**

■ **ANSI C:**

- **The rapid growth of C led to the development of different versions of C, that were similar but often became incompatible. This proposed a serious problem for system development.**
- **To assure that the C language remains a standard, in 1983, ANSI (American National Standards Institute) appointed a technical committee to define a standard for C.**
- **This committee approved a version of C in December 1989, which is known as ANSI C.**

- **ANSI/ISO C:** ANSI C was approved by ISO (International Standards Organization) in 1990; hence, known as ANSI/ISO C. This version of C is also referred to as C89.
- **C99:**
 - During 1990's, C++, a language based upon C, underwent a number of changes.
 - During the same period, Sun Microsystems of USA created a new language JAVA modeled on C++.
 - In 1999, the standardization committee of C felt that some new features of C++/JAVA, if added to C, would enhance the usefulness of the language.
 - The result was C99.

[NOTE]: This paper is base on the ANSI/ISO C standard.

Key Characteristics of C

The increasing popularity of C is due to its following characteristics:

1. **C is a robust language:** It has a rich set of operators and built-in functions that can be used to write even very complex programs.
2. **C programs are efficient and fast:** C is actually a middle-level language. Hence it is well suited for writing system software along with utility software.
3. **C has only 32 key words:** A small set but highly effective.
4. **C is highly portable:** C programs written for one computer can be run on another computer with little or no modification.
5. **C is a structured:** Modular form of C makes it easier to develop, debug, test and maintain.
6. **C has the ability to extend itself:** The user can write and add his own functions to the C library or even can create his own library.

The 1st C Program

```
1  /* PR1_1.c: Program that displays the message "Welcome to Sea" */
2  #include <stdio.h>
3  #include <conio.h>
4  void main()
5  {
6      printf("Welcome to Sea. \n");
7      getch();
8  }
9  /* Program Ends */
```

Output

```
Welcome to Sea
```

Explanation

■ **Line Nos. 1 and 9:** These are comment lines.

- Comments in C begins with `/*` and ends with `*/`.
- Comments are NOT executable statements; they are ignored by the compiler.
- It is always a good programming style to include comments because, it helps the programmer and others to understand & debug the program.
- Comments can be put at any part of a program, but NOT within a word or a statement.
- Comments can't be nested.

- **Line Nos. 4, 5, and 8:** This is something which is called a function in C (functions are discussed in detail in another Chapter).

- In C, the syntax of writing function is as follows:

```
return_type function_name (type arg1, type arg2, ...)
{
    /* An opening curly brace marks the beginning of a function */
    statement(s); /* Statements within the opening & closing curly braces
                    forms the function body */
}
/* A closing curly brace marks the end of a function */
```

- In our example, the name of the function is **main**.
- **main** is a special function in C that **tells the compiler where to start the program execution** (In C, all executable statements are put in one or more functions. So, which function should execute 1st when the program runs? Program execution always starts from **main**. It call other functions as and when needed.)
- **Every C program must have exactly one main function** (if there are more than one **main**, the compiler can't understand which one marks the beginning of the program. So, it generate an error message.)

- The empty parenthesis immediately following `main` indicates that `main` takes no **arguments**.
- The word `void` before `main` indicates that `main` has no **return type** (`main` returns nothing to the compiler).
- The opening curly brace `{` marks the **beginning of** `main`.
- The closing curly brace `}` marks the **end of** `main`.
- All the statements within the braces (in our example line numbers 5 & 6) forms the **body of** `main`.
- C permits following forms of `main`:

<code>main()</code> <code>int main()</code> <code>main(void)</code> <code>int main(void)</code>	}	All are same (Takes no arguments, returns an integer value)
<code>void main()</code> <code>void main(void)</code>	}	Both are same (Takes no arguments, returns nothing)
<code>int main(int argc, char* argv[])</code> <code>void main(int argc, char* argv[])</code>	}	Used for accepting command line arguments (discussed in Chapter “Files and Command Line Arguments.”)

- **Line No. 6:** This is the 1st executable statement in our program. This statement prints the message “Welcome to Sea” [printf is discussed in detail in Chapter 4.]
 - `printf` is a **pre-defined C function** for printing / displaying the output (In C, there are two types of functions, pre-defined and user-defined. Pre-defined means already written and compiled; readily available for using).
 - `printf` prints everything within the double quotes (“ ”).
 - Note the semicolon (;) at the end of the `printf` statement. **Every C statement should end with a semicolon.**
- **Line No. 7:** This is the 2nd executable statement in our program. `getch()` is also a pre-defined function. The details about this are discussed in Chapter 4..
- **Line No. 2 and 3:** These are **pre-processor directives**.
 - In C, every line that **begins with a hash (#)** is known as a pre-processor directive.
 - These lines are processed by a program called the **pre-processor** (pre-processing happens before compiling - details are discussed soon).

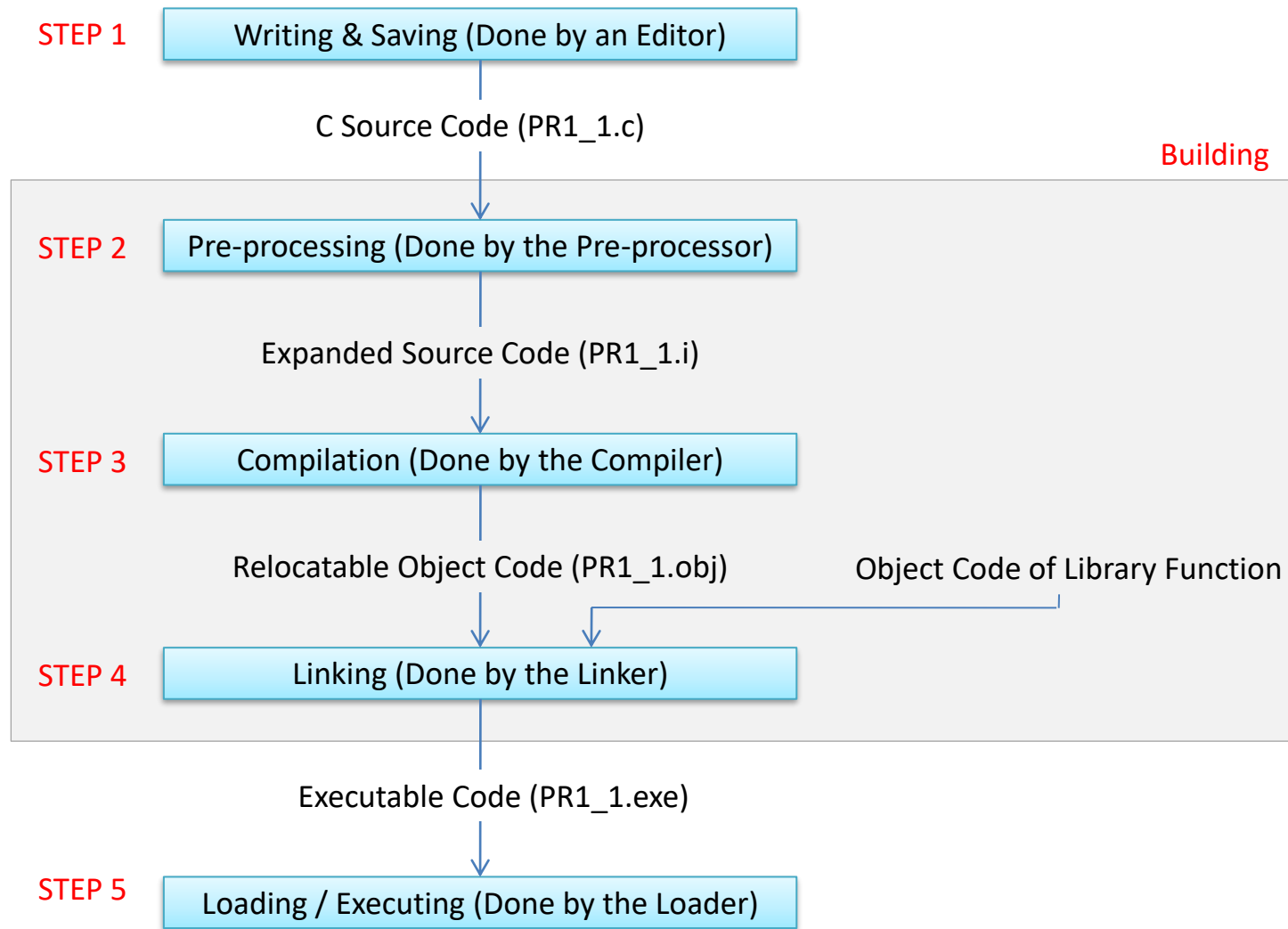
- The `#include` directive tells the pre-processor to *include* the files `stdio.h` and `conio.h` into the C program at the same place where the corresponding directive is written (When the compiler compiles the program it sees the content of the file `stdio.h` and `conio.h` in the exact place where the corresponding pre-processor directive is written).
- The files `stdio.h` (“stdio” stands for “standard input/output”) and `conio.h` (“conio” stands for “console input/output”) are header files that *contains information on the library input/output functions*. In our example `stdio.h` contains information on `printf` and `conio.h` contains information on `getch`. This information is required by the compiler, otherwise the compiler can’t understand “what are `printf` and `getch`?”.
 - **[NOTE]: C is case sensitive** (i.e., `printf` and `PRINTF` are different).
 - In C, all pre-defined things are in lower case.
 - It is a convention to write all user-defined things also in lower case, except the symbolic constants (discussed later).

Writing, Building & Executing the 1st Program

The Concept of Writing, Building & Executing

Many steps are involved between writing a C program and it gets executed. The figure in the next slide shows these steps.

[NOTE]: Most of the IDEs (Integrated Development Environments) like TC++, VC++, CodeBlocks etc. hide (does automatically) many of the steps for us. We will study *how* to perform these steps in different environments (OS) later, but before this let's 1st understand *what* is done at each step.



■ STEP 1 - Writing & Saving (Done by an Editor):

- The first step is to write & save the C program by using a suitable editor.
- The file thus obtained is called the **source code**. It has the extension `.c` (in our case `PR1_1.c`).

■ STEP 2 - Pre-processing (Done by the Pre-processor):

- In this step the pre-processor **expands the source code** based upon the pre-processor directive like `#include`, `#define`, `#ifdef` etc.
- The expanded source code is stored in an **intermediate file** with `.i` extension (in our case `PR1_1.i`).
- The expanded source code is also in C.

■ STEP 3 - Compilation (Done by the Compiler):

- The C compiler, on receiving the expanded code, does the following:
 - a) Identifies the syntax error (if any).
 - b) Displays the errors along with warnings.
 - c) If the expanded source code is error free, it translates this code to equivalent **relocatable object code** (machine-level code).

[NOTE]: Here the word “relocatable” means the program is ready to execute except for one thing - no specific memory address have yet been assigned to the code. All the addresses are relative offsets.

- The relocatable object code (machine-level code) thus obtained, is stored in a separate file, called the **object file**, with `.obj` extension (in our case `PR1_1.obj`).

- **STEP 4 - Linking (Done by the Linker):** The linker combines different object files to produce actual executable code. The executable code is stored in a separate file with `.exe` extension (in our case `PR1_1.exe`).

[NOTE]: The Concept of Linking

- All most all C programs use a set of pre-defined functions like `printf`, `scanf` etc., called the **library functions**. To increase the efficiency, these set of library functions are pre-compiled and stored as object files (binary format), called the **library files**.
- The information about these *library files* is contained in various *header files*. For example, the header file `"stdio.h"` actually contain **the information about** all the I/O library functions like `printf`, `scanf` etc.(their prototypes, and in which library file they are actually located).
- We include these header files in our source code (`PR1_1.c`) through the pre-processor directive `"#include"` (e.g., `#include <stdio.h>`).

- When the source code (PR1_1.c) is compiled, the object file (PR1_1.obj) retains this information present in the header files, along with the machine-level instructions.
- The linker on getting this information, searches the corresponding library functions (e.g., `printf`) in the mentioned library file. If the library function is found, it links / joins that library file with the object file (PR1_1.obj) to produce the desired executable file. On the other hand, if the library functions is not found in that library file or the library file itself is missing a link error is displayed.

[NOTE]: Other advantages of linking (besides improving the efficiency)

The generality of the linking scheme enables

- The uses to build their own library files.
- To combine the programs written in different languages (as the object files will be in the same machine-level language)

■ STEP 5 - Loading / Executing (Done by the Loader):

- The loader loads the executable file (PR1_1.exe) in memory for execution.
- Normally, after calling some platform (OS) specific initialization functions, `main()` is called.

Writing, Building & Executing the Program in UNIX Platform

- **STEP 1:** Open a text editor (e.g., vi, emacs etc.) and create a new file `PR1_1.c`.

This is done by writing the following command in the command prompt:

```
vi PR1_1.c
```

This is the command for calling the `vi` editor and creating the file `PR1_1.c`. If the file existed before, it will be loaded, otherwise a new file with this name will be created. Once the editing is done, save the file.

- **STEP 2:** Compile the file (In UNIX pre-processing and linking is automatically done during compilation). This is done by writing the following command in the command prompt:

```
cc PR1_1.c
```

Every UNIX system has an inbuilt C compiler, called `cc` (c c compiler). Once this step is successful, the linker will create the **default** executable file `a.out`.

- **STEP 3:** Run the default executable file `a.out`. This is done by writing the following command in the command prompt:

```
a.out
```

Writing, Building & Executing the Program in WINDOWS Platform

- In windows platform there are many IDEs (Integrated Development Environments), that includes all facilities to edit, build and run a program at one place.

Examples of such IDEs are:

- Turbo C++
- CodeBlocks

One More Program

```
/* PR1_2.c: Program that computes and displays the average of 3 numbers */
```

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

```
void main()
```

```
{
```

```
    int num1, num2, num3; /* variable declaration */
```

```
    float avg; /* variable declaration */
```

```
    printf ("Enter the 1st number: ");
```

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

```
    printf ("Enter the 2nd number: ");
```

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

```
    printf ("Enter the 3rd number: ");
```

```
    scanf ("%d", &num3);
```

```
    avg = (num1+num2+num3)/3;
```

```
    printf("\n\nThe average is: %f\n\n", avg);
```

```
}
```

```
/* Program Ends */
```

Output

```
Enter the 1st number: 3  
Enter the 2nd number: 4  
Enter the 3rd number: 6  
  
The average is: 4.000000
```

The Same Program Using User-Defined Function

```
/* PR1_3.c: Program that computes and displays the average of 3 numbers by using  
user-defined function*/
```

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

```
float average (int a, int b, int c); /* Function prototype */
```

```
void main()
```

```
{
```

```
    int num1, num2, num3;
```

```
    float avg;
```

```
    printf ("Enter the 1st number: ");
```

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

```
    printf ("Enter the 2nd number: ");
```

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

```
    printf ("Enter the 3rd number: ");
```

```
    scanf ("%d", &num3);
```

```
    avg = average (num1, num2, num3); /* Function call*/
```

```
    printf("\n\nThe average is: %f\n\n", avg);
```

```
}
```

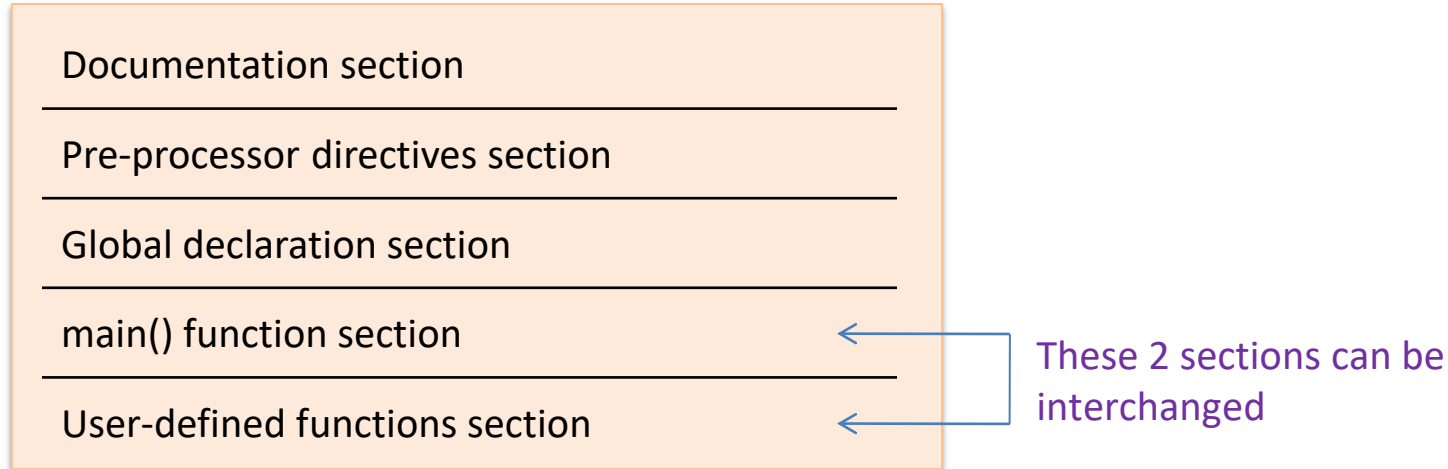
[Cont.]

```
float average (int a, int b, int c) /* Function definition */  
{  
    float avg;  
    avg = (a+b+c)/3;  
    return avg;  
}  
  
/* Program Ends */
```

Output

```
Enter the 1st number: 3  
Enter the 2nd number: 4  
Enter the 3rd number: 6  
  
The average is: 4.000000
```

Structure of a C Program



- **Documentation Section:** It contains a set of comment lines giving the name of the program, its purpose, the author and other details.
- **Pre-processor Directive Section:** It contains the pre-processor directives like `#include`, `#define`, `#ifdef` etc.
- **Global Declaration Section:** It contains declaration of global variables (the variables used by all functions) and function prototypes.

- `main()` **Function Section:** It contains the `main()` function.
- **User-Defined Function Section:** It contains a set of user-defined functions which are either called by `main()` or other user-defined functions.

Assignments

Complete the experiments given in “Lab Manual - Section 1”.

End of Chapter 1