NEP NEW SYLLABUS

A Book Of

Advanced C Programming

For F.Y.B.Sc. (Computer Application) : Semester – II

2024 Pattern as per NEP-2020

As Per New Syllabus, Effective from 2024-2025

Mrs. Deepali N. Bhoskar
M. Phil, M.Sc. (Computer Application)
Free Lancer, Systronics Academy , Pune

Mr. Kamil Ajmal Khan
Assistant Professor
B.C.A./M.C.A. Department
Abeda Inamadar Senior College, Camp, Pune

Price ₹ 185.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N8326

ADVANCED C PROGRAMMING **ISBN 978-93-6109-402-6**

First Edition : January 2025
© : Authors

The text of this publication, or any part thereof, should not be reproduced or transmitted in any form or stored in any computer storage system or device for distribution including photocopy, recording, taping or information retrieval system or reproduced on any disc, tape, perforated media or other information storage device etc., without the written permission of Authors with whom the rights are reserved. Breach of this condition is liable for legal action.

Every effort has been made to avoid errors or omissions in this publication. In spite of this, errors may have crept in. Any mistake, error or discrepancy so noted and shall be brought to our notice shall be taken care of in the next edition. It is notified that neither the publisher nor the authors or seller shall be responsible for any damage or loss of action to any one, of any kind, in any manner, therefrom. The reader must cross check all the facts and contents with original Government notification or publications.

Published By :
NIRALI PRAKASHAN
Abhyudaya Pragati, 1312, Shivaji Nagar,
Off J.M. Road, Pune – 411005
Tel - (020) 25512336/37/39
Email : niralipune@pragationline.com

Polyplate

Printed By :
YOGIRAJ PRINTERS AND BINDERS
Survey No. 10/1A, Ghule Industrial Estate
Nanded Gaon Road
Nanded, Pune - 411041

DISTRIBUTION CENTRES

PUNE

Nirali Prakashan
(For orders outside Pune)
S. No. 28/27, Dhayari Narhe Road, Near Asian College
Pune 411041, Maharashtra
Tel : (020) 24690204; Mobile : 9657703143
Email : bookorder@pragationline.com

Nirali Prakashan
(For orders within Pune)
119, Budhwar Peth, Jogeshwari Mandir Lane
Pune 411002, Maharashtra
Mobile : 9657703145, 9890997937
Email : niralilocal@pragationline.com

MUMBAI

Nirali Prakashan
Rasdhara Co-op. Hsg. Society Ltd., 'D' Wing Ground Floor, 385 S.V.P. Road
Girgaum, Mumbai 400004, Maharashtra
Mobile : 7045821020, Tel : (022) 2385 6339 / 2386 9976
Email : niralimumbai@pragationline.com

DISTRIBUTION BRANCHES

DELHI

Nirali Prakashan
Room No. 2 Ground Floor
4575/15 Omkar Tower, Agarwal Road
Darya Ganj, New Delhi 110002
Mobile : 9555778814/9818561840
Email : delhi@niralibooks.com

BENGALURU

Nirali Prakashan
Maitri Ground Floor, Jaya Apartments,
No. 99, 6th Cross, 6th Main,
Malleswaram, Bengaluru 560003
Karnataka; Mob : 9686821074
Email : bengaluru@niralibooks.com

NAGPUR

Nirali Prakashan
Above Maratha Mandir, Shop No. 3,
First Floor, Rani Jhanshi Square,
Sitabuldi Nagpur 440012 (MAH)
Tel : (0712) 254 7129
Email : nagpur@niralibooks.com

KOLHAPUR

Nirali Prakashan
438/2, Bhosale Plaza, Ground Floor
Khasbag, Opp. Balgopal Talim
Kolhapur 416 012, Maharashtra
Mob : 9850046155
Email : kolhapur@niralibooks.com

JALGAON

Nirali Prakashan
34, V. V. Golani Market, Navi Peth,
Jalgaon 425001, Maharashtra
Tel : (0257) 222 0395
Mob : 94234 91860
Email : jalgaon@niralibooks.com

SOLAPUR

Nirali Prakashan
R-158/2, Avanti Nagar, Near Golden
Gate, Pune Naka Chowk
Solapur 413001, Maharashtra
Mobile 9890918687
Email : solapur@niralibooks.com

marketing@pragationline.com | www.niralibooks.com

Also find us on www.facebook.com/niralibooks

Preface ...

We take this opportunity to present this book entitled as **"Advanced C Programming"** to the students of Second Semester – B.sc(CA). The object of this book is to present the subject matter in a most concise and simple manner. The book is written strictly according to the New Syllabus (2024 pattern as per NEP-2000).

The book has its own unique features. It brings out the subject in a very simple and lucid manner for easy and comprehensive understanding of the basic concepts, its intricacies, procedures and practices. This book will help the readers to have a broader view on Advanced C Programming. The language used in this book is easy and will help students to improve their vocabulary of technical terms and understand the matter in a better and happier way.

We sincerely thank Shri. Dineshbhai Furia and Shri. Jignesh Furia of Nirali Prakashan, for the confidence reposed in us and giving us this opportunity to reach out to the students of management studies.

We have given our best inputs for this book. Any suggestions towards the improvement of this book and sincere comments are most welcome on niralipune@pragationline.com.

Authors

■■■

Syllabus ...

1. **Preprocessor** **[06 Hrs]**
   - Concept, Format of Preprocessor Directives, File inclusion Directives (#include), Macro Substitution Directives (#define), Nested Macros, Parameterized Macros.
   - Macros Versus Functions, #error / #pragma directives, Conditional Compilation (#if/#ifdef/#else/#elif/#endif), Predefined Macros (_DATE_ / _TIME_ / _FILE_ / _LINE_/ _STDC_).
2. **Pointers** **[07 Hrs]**
   - Concept – Reference and Dereference, Declaration, Definition, Initialization and Use, Types of Pointers.
   - Pointer Arithmetic, Multiple Indirection.
   - Parameter Passing – Call By Value and Call By Reference.
   - Arrays and Pointers - Pointer to Array, Array of pointers.
   - Functions and Pointers - Passing Pointer to Function, Returning Pointer from Function, Function Pointer, Pointers and const.
   - Dynamic Memory Management, Allocation, Resizing, Releasing, Memory Leak / Dangling Pointers.
3. **Strings** **[05 Hrs]**
   - Concept, Declaration, Definition, Initialization, Format Specifiers, String Literals/ Constants and Variables – Reading and Writing from and to Console, Importance of Terminating NULL Character, Strings and Pointers.
   - Array of Strings and Array of Character Pointers, User Defined Functions, Predefined Functions in string.h - strlen, strcpy, strcat, strcmp, strcmpi, strrev, strlwr, strupr, strset, strchr, strrchr, strstr, strncpy, strncat, strncmp, strncmpi, strnset, strtok, Command Line Arguments – argc and argv.
4. **Structures** **[06 Hrs]**
   - Concept, Declaration, Definition, Initialization, Accessing Structure Members (.operator).
   - Array of Structures, Pointers to Structures, Declaring Pointer to Structure.
   - Accessing Structure Members via Pointer to Structure, Structures and Functions.
   - Passing each Member of Structure as a Separate Argument, Passing Structure by Value / Address.
   - Nested Structures, typedef and Structures, Concept of Union.
5. **File Handling** **[06 Hrs]**
   - Concept of Streams, Need, Types of Files, Operations on Text and Binary Files, Random Access File, Library Functions for File Handling – fopen, fclose, fgetc, fseek, fgets, fputc etc.

■■■

Contents ...

1. **Preprocessor** 1.1 – 1.18

2. **Pointers** 2.1 – 2.30

3. **Strings** 3.1 – 3.36

4. **Structures** 4.1 – 4.28

5. **File handling** 5.1 – 5.34

■■■

1... Preprocessor

Objectives ...

- To understand concepts of preprocessor.
- To study format of preprocessor directives.
- To learn Nested macro and Parameterized macro.
- To study predefined macros.

1.1 INTRODUCTION

- The C language preprocessor is a collection of special statements, called directives that are executed at the beginning of the complication process.
- Preprocessor directives are processed by a program called a preprocessor.
- Each of the preprocessor directives begins with a hash symbol (#) and do not require semicolon at the end.

1.2 CONCEPT

- C preprocessor is a unique feature of C programming language.
- C preprocessor has many tools that are not available in other high-level languages. These tools can be used by programmer to make his program easy to write, read, and modify, more efficient and portable.
- **Preprocessor** is a program that processes source code before it passes through the complier.
- It operates under the control of preprocessor directives or preprocessor command line.
- Preprocessor directives are placed in source program before main ().
- The source code is first processed by the preprocessor, which checks for any preprocessor directives. After handling these directives, the modified code is passed to the compiler for compilation.

1.3 FORMAT OF PREPROCESSOR DIRECTIVES

- Preprocessor directives follow special format that are different from normal C program formats.
- **Rules of Preprocessor Format:**
  1. All preprocessor directives start with # symbol in first column.
  2. Preprocessor do not require semicolon at the end e.g., #include, #define.
  3. Preprocessor directives are placed in the source program before calling the main () function.
- Following is list of commonly used preprocessor directives and their functions.

Table 1.1: Preprocessor Directives and Functions

Sr. No.	Directives	Functions
1.	**#define**	Defines a macro substitution.
2.	**#include**	Specifies files to be included.
3.	**#undef**	Undefines a macro.
4.	**#ifdef**	Test for macro definition.
5.	**#endif**	Specifies the end of #if.
6.	**#ifndef**	Tests whether a macro is not defined.
7.	**#if**	Test a compile time condition.
8.	**#else**	Specifies alternatives when #if test fails.

- The above listed directives can be divided into three categories:
  1. File Inclusion directives (#include).
  2. Macro substitution directives (#define).
  3. Complier control directives.

1.4 FILE INCLUSION DIRECTIVES (#include)

- As a part of program, an external file containing functions or macro definitions can be included.
- #include preprocessor directive is used to include an external file or macro.

Format of file inclusion directive:

```
#include "filename"
      or
#include<filename>
```

Where, Filename is the name of file containing required definitions or functions.

- #include preprocessor inserts entire contents of filename into the source code of program.
- There is difference between "filename" and <filename> inclusion.
  - When filename is included in double quotation mark, the search for file is first made in current directory and then in standard directories.

- When filename is without double quotes, file is searched only in standard directories.

- Nesting of included files is allowed i.e., an included file can include another file but a file cannot include itself.
- If included file is not found, an error is reported and compilation is terminated.

Example:

- Let us consider following 'C' programs:
  1. SYMBOLS.c contains symbols definitions.
  2. SHAPES.c contains shapes related functions.
  3. TEST.c contains test functions.
- We can add these definitions or functions to use in program by including them with #include preprocessor directive.

```
#include <stdio.h>
#include "SYMBOLS.c"
#include "SHAPES.c"
#include "TEST.c"
#define x 100
main ()
{
    ..............
    ..............
}
```

1.5 MACRO SUBSTITUTION DIRECTIVES (#define)

- An identifier in a program is replaced by a predefined string composed of one or more tokens, is called as Macro Substitution.
- Macro substitution process is accomplished with the help of **#define** directive statement.
- Macro statement is usually known as Macro definition or a Macro.

Syntax of Macro declaration:

```
#define identifier string
```

- If a macro is included in a program at the beginning, preprocessor replaces each occurrence of identifier in the source code by the string.
- #define keyword is written in the first column followed by identifier and a string with at least one blank space between them.
  [Note: Definition is never terminated by a semicolon.]
- String may be any text whereas identifier must be a valid 'C' name.
  e.g. `#define COUNT 100`

Types of Macro substitution:

- Macro substitution in programming, particularly in languages like C and C++, involves replacing code snippets with predefined macros. There are two main types of macro substitution:
  1. **Object-like Macros:**
     These are simple substitutions where a macro name is replaced by a value or a code snippet.
     Example:
     ```
     #define PI 3.14
     ```
     In this case, any occurrence of PI in the code is replaced with 3.14.
  2. **Function-like Macros:**
     These resemble functions and can take arguments, allowing for more dynamic code substitution.
     Example:
     ```
     #define SQUARE(x) ((x) * (x))
     ```
     Here, SQUARE(5) would be replaced with ((5) * (5)).
- Following are common forms of Macro substitution:
  1. Nested Macro Substitution.
  2. Parameterized Macro (Argumented Macro Substitution).
  3. Macros Vs Functions.
- Let us see details of each form in next sections.

Simple Macro Substitution:

- **To define constants simple string replacement is used.**

Example:
```
#define        M                500
#define      TRUE                1
#define    SUBJECTS              4
#define       PI               3.142
#define     STATE        "MAHARASHTRA"
```

- It is a convention to write all macros **(Identifiers)** in capital to identify them as symbolic constants.
  ```
  #define X    5
  ```
  For above macro definition, for occurrences of X will be replaced with 5, starting from line of definition to the end of program.
- A macro name inside a string does not get replaced.
  Example:
  ```
  Sum = X * value;
  printf("X = % d\n", X);
  ```
  These two lines would be changed during preprocessing as shown below:
  ```
  Sum = 5 * value;
  printf("X = % d\n", 5);
  ```
  Notice that the string "X = %d\n" is left unchanged.

- A macro definition can include more than a simple constant value. It can include expressions. Following are valid definitions.

```
#define        AREA        5*12.46
#define        SIZE        Sizeof (float)*4
#define        TWO-PI      2.0*3.1415926
```

- It is good practice to use parentheses for expressions used in macro definitions.

```
#define        M           (45 - 22)
#define        N           (78 + 32)
```

Consider evaluation of expression,

$$\text{ratio} = \frac{M}{N}$$

parenthesis used result is different and if not used results would be different.

- Semicolons are not used to terminate #define statements as preprocessor replaces macro definition whenever defined macro name occurs.

Example:

```
#define        STMT        if(a > b)
#define        AND
#define        DISPLAY     printf("a is greater \n");
```

If a statement is written as,

```
STMT AND DISPLAY
```

The preprocessor would replace the statement to if (a > b) printf("a is greater \n");

Program 1.1: To define a macro using #define directive.

```
#include <stdio.h>
#define AREA length*breadth
/* macro AREA definition */
void main( )
{
   int length, breadth;
   clrscr( );
   printf("Enter the Length and Breadth\n\n");
   scanf("%d %d",&length, &breadth);
   /* echo the data */
   printf("\n Length = %d Breadth = %d\n", length, breadth);
   printf("InArea = %d square units", AREA);
   getch( ); /* freeze the screen */
}
```

Output:

```
Enter the Length and Breadth
10 5

Length = 10 Breadth = 5
InArea = 50 square units
```

1.6 NESTED MACROS

- Macro definitions may be nested. Nested macro means, we can use one macro in definition of another macro.

  Example:

```
#define          A                  5
#define          B                A + 2
#define      SQUARE (X)         ((X) * (X))
#define       CUBE (X)       (SQUARE(X) * (X))
#define       TOTAL (X)     (CUBE (X) * CUBE (X))
```

- The preprocessor expands each #define macro until no more macros appear in the text.

  Example:

```
(CUBE (X) * CUBE (X))
```

  Last definition of TOTAL (X)

  Last definition is first expanded into

```
(SQUARE (X) * (X))      *      (SQUARE (X) * (X))
   CUBE (X)                       CUBE (X)
```

  Since SQUARE(X) is a macro, it is further expanded into

```
(((X) * (X)) * (X))      *      (((X) * (X)) * (X))
   SQUARE (X)                      SQUARE (X)
```

  The final evolution of TOTAL(X) is X^6.

- Macro calls can be nested in similar way as function calls.

  Example:

  1. Given definitions of P and Q, we can define following macro to give max of 2.

```
#define MAX (P,Q)     ((P) > (Q))?  (P): (Q)
```

2. Similarly given definition of MAX (P, Q), we can use following nested call to give max of 3 values X, Y, Z.
   `MAX (X, MAX (Y, Z))`

Program 1.2: For nesting macro.

```
#include<stdio.h>
#define SQU(x)((x)*x)
#define CUBE(x)(SQU(x)*x)
int main()
{
   int x;
   int y;
   x = SQU(3); // Argumented Macro
   y = CUBE(3); // Nested Macro
   printf("\n Square of 3: %d",x);
   printf("\n Cube of 3: %d",y);
   return(0);
}
```

Output:

```
Square of 3: 9
Cube of 3: 27
```

Undefining a Macro: Using #undef statement, a defined macro can be undefined.

Syntax: `#undef identifier`

Example: `#undef MAX (P, Q)`

- This is useful when we want to restrict the definition only to particular part of program.

1.7 PARAMETERIZED MACROS

- A parameterized macro is a macro that is able to insert given parameter into its expansion.
- Each time the macro name is encountered the parameters associated with it and are replaced by actual parameters found in the program.
- It works like function with parameters so parameterized macros are also called as Macro Function.
- Parameters are not case sensitive; therefore, we can pass any numeric variable to an argumented macro that expects a numeric argument.
- Argumented macros execute much faster as compared to similar functions.

Syntax of Parameterized Macros:

```
#define identifier (f1, f2… fn) string
```

The identifier f1, f2, ..., fn are formal macro parameters that are similar to formal parameters in function definition.

- Occurrence of macro with parameter is known as macro call (similar to function call).
- Whenever a macro is called the preprocessor substitutes the string, replacing the formal parameter with actual parameters.

Example:

```
#define   CUBE(m)   (m*m*m)
```

Where CUBE is macro name, m is a formal parameter and (m*m*m) is definition of macro with parameter CUBE(m).

```
Volume = CUBE(side);
```

If above statement appears in program, then preprocessor would expand this statement to,

```
Volume = (side*side*side);
```

(side*side*side) is expansion of CUBE(side) macro with actual parameter i.e. side.

Program 1.3: For macro with arguments.

```
#include <stdio.h>
/* macro LARGEST definition */
#define LARGEST(a, b, c) (a>b?(a>c? a:c):(b>c?b:c))
void main( )
{
    float x=65.3, y=35.2, z=87.9, big;
    int i=41, j=91, k=11, max;
    clrscr();
    big = LARGEST(x,y,z);
    printf("Largest of %8.2f %8.2f %8.2f is %8.2f\n\n", x, y, z, big);
    max = LARGEST(i,j,k);
    printf("Largest of %8d %8d %8d is %8d\n\n", i, j, k, max);
}
```

Output:

```
Largest of    65.30    35.20    87.90 is    87.90

Largest of      41       91       11 is      91
```

Program 1.4: For macros with parameters.

```
#include <stdio.h>
#define MAX(x,y) (x>y ? x:y)
int main(void)
{
    clrscr();
    printf("\n Max between 40 and 20 is %d\n", MAX(40, 20));
    return 0;
}
```

Output:

```
Max between 40 and 20 is 40
```

1.8 MACROS VERSUS FUNCTIONS

Table 1.2

Sr. No.	Macros	Functions
1.	Macros are preprocessed i.e., all macros would be processed before program complies.	Functions are not preprocessed but compiled.
2.	No type checking which can lead to errors sometimes.	Type checking is done in the function.
3.	Using macro increases the code length.	Using function keeps the code length unaffected.
4.	Speed of execution is faster.	Execution speed is slower.
5.	Before compilation, macro name is replaced by macro value.	During function call, transfer of control takes place.
6.	Macros are useful when small code is repeated many times.	Functions are useful when large code is to be written.
7.	Macros do not check compile time errors.	Function checks compile time errors.

1.9 #ERROR / #PRAGMA DIRECTIVES

1.9.1 #pragma Directive

- #pragma is an implementation oriented directive that allows us to specify various instructions to be given to the compiler.

  Syntax: `#pragma name`

  Where, name is the name of pragma instruction to be given.

  Example: `#pragma startup`

  Before the execution of main( ), the function specified in pragma is needed to run.

  ```
  #pragma exit
  ```

  Before the end of program, the function specified in pragma is needed to run.

Program 1.5: For *pragma* directives.

```
#include <stdio.h>
int display();
#pragma startup display
#pragma exit display
int main()
{  printf("I am in main function");
   return 0;
}
```