

Structured programming (top-down approach to program development

What is structured programming?

Is a paradigm that emphasizes breaking down a program into small, clear, and manageable sections (modules or blocks), instead of writing code as one long continuous sequence of instructions. Control structures are used to organize programs.

It is an approach to writing programs that are easier to read, test, debug and modify. The approach helps in the development of large programs through stepwise refinement and modularity. Programs that are designed this way can be developed faster. When modules are used to develop large programs, several programmers can work on different modules, thereby reducing program development time.

Key features of structured programming

Modularity: programs are divided into small reusable functions or modules

Control structures (sequence, selection, iteration)

Top-down design: start with the main problem, then break it into smaller sub-problems

Single entry and exit points: each block of code has only one way in and one way out, making it easy to understand and debug.

Readability: code is easier to follow because of structured flow

In summary, structured programming serves to increase programmer productivity, program readability, program testing, program debugging and serviceability.

Advantages of structured programming

- Programs are easier to read, understand and maintain
- Supports easy debugging and testing
- Promotes code re-usability
- Reduces errors and improves readability

Steps in program development

1. Design program objectives (requirement analysis)

It involves forming a clear idea in terms of the information you want to include in the program, computations needed and the output {identification of inputs, processes and outputs or reports}. At this stage, the programmer should think in general terms, not in terms of some specific computer language.

2. Program design

The programmer employs the use of design tools like pseudocodes, flowcharts, data flow diagrams, ERDs to develop an algorithm. The programmer decides how the program will go about its implementation, what should the user interface be like {**input design**}, how should the program be organized, how to represent the data and what methods to use during processing. {Design of **input screens**, **file design** (variables and data types), **process design** and **output design** (reports the program should generate)}

3. Develop the program (coding)

Involves translating the design into a programming language. Write the source code statements in a chosen language e.g. C. the design specification is used in writing the source code. Follow coding standards and best practices

Developing a program in a structured language such as C requires at least four steps:

Editing (or writing) the source code

Compiling the source code

Linking the source code

Executing

Editing

Source code statements are written using words and symbols that are understandable to human beings. The program is typed directly into a program editor window on the screen and saving the resulting text as a separate file. C program is stored in a file with the extension .C for C programming language.

Compiling

Is the process of translating the source code into object code by the compiler. The source file must be translated into binary numbers understandable to the computer's central processing unit. This process produces an intermediate object file with the extension .obj. errors detected during this phase are normally displayed on the screen. The source code can only compile when all errors (especially syntax errors) have been corrected. A syntax error occurs when the compiler cannot recognize a statement because it violates the rules of the language. The compiler issues an error message to help in locating and fixing the incorrect statements. The `stderr` stream (normally connected to the screen) is used for displaying `error messages`. Syntax errors are also called compile errors or compile-time errors.

Linking

The object code file can be made into a fully executable program by carrying out a **Linking** process, which joins {tying together} the object code to all the other files that are needed for the execution of the program {adding library files to the program}. These routines {library files} are written by the manufacturer of the compiler to perform a variety of tasks, from input/output to complicated mathematical functions. {In the case of C the standard input and output functions are contained in a library (stdio.h)} After the linking process, an executable file with an *.EXE* extension is generated. This file is stored on a storage media.

Object code-----tied together with----library files → executable file

Executable files

The text editor produces `.c` source files, which go to the compiler. The compiler produces `.obj` object files, which go to the linker that finally produces `.exe` executable file. `.exe` file can then be run like any other application simply by typing the file name at the DOS prompt or running using windows menu.

Loading

Before the program can be executed, it must first be `placed` in `memory`. This is done by the `loader`, which takes the `executable image` from disk and transfers it to memory. Additional components

from shared libraries that support the program are also loaded.

4. Testing the program

This involves checking whether the program does what it is supposed to do. Programs may have **bugs** {syntax, logical or run time errors}. Debugging involves the finding and fixing of program errors. Types of testing that should be done include: Unit testing, integration testing, system testing and user acceptance testing. {read about beta and alpha testing}

Logical errors: Result from a wrong algorithm or program logic {incorrect use of control structures, incorrect calculation or omission of a procedure}. They are the hardest of all error types to detect. The program compiles and executes, but the output is not what you wanted or expected. {Programmer should dry run the program to detect them}

Syntax errors: Occur when the programmer does not follow the rules of the programming language. They prevent the program from running. They are also called compilation or compiler errors. {Some syntax errors include; missing semicolon at the end of a statement, use of undeclared variable, illegal declaration, misspelling keywords}

Run time errors: Occurs while the program is running e.g., when the program tries to do an operation that is impossible to carry out like dividing a value by 0.

5. Deployment

Release the software to users (installation, setup, training)

Can also involve rolling out to production, packaging and documentation

6. Program maintenance

Sometimes it's necessary to make changes to the program after some time. { corrective maintenance involves correcting errors that were not detected during testing stage, adaptive maintenance, that involves making changes to a program to make it adapt to new changes, perfective maintenance – improving the program in terms of interface, performance or efficiency}. A good program documentation simplifies the maintenance exercise greatly.

Introduction to C language

C is a structured programming language. It is called a compiled language. This means that once you write your C program, you must run it through a C compiler to turn the program into an executable code that the computer can run.

C was developed in 1970 by Dennis Ritchie at Bell Laboratories and was used to develop **UNIX** operating system. It is sometimes called a **middle level language** because it **interfaces** assembly language and high level languages like JAVA {can be used to write low level programs as well as high level programs}. Before C was invented, assembly languages were used to write computer programs. The shortcomings of the assembly language were one reason behind inventing C. it successfully combines the **features of a high level language** and the **power and efficiency of assembly language**.

C was initially designed for creating system software like operating systems, compilers and editors. Today C can be used to create any computer program {system software as well as application software}

C supports **functions** that enable easy maintainability of code, by breaking large file into smaller **modules**. Use of **comments** in C provides easy readability.

C programs are built from variable and type declarations, functions, statements and expressions.

Why C is still popular?

1. **C supports structured programming design features.** It allows programmers to break down their programs into functions. It also supports the use of comments, making programs readable and easily maintainable
2. **Simplicity and Efficiency.** Final code tends to be more compact and **runs quickly**. C is a concise language that allows you to say what you mean in a few words. {provides a small set of key words – 32 and constructs}
3. **Portability:** c programs can run on different machines with little or no modification
4. **Foundation of many other languages** – languages such as C++, java,C#, python are descendants of C
5. **Power and flexibility.** It's a powerful language used to write system software {operating systems like unix and windows, language compilers, network drivers} and

application software. It can also be used to solve problems in **any field** {physics and engineering, business, etc}

6. **Programmer oriented.** C is oriented towards the programmer's needs. It gives access to the hardware. It also has a rich selection of operators that allows the programmer to expand programming capability.
7. **Rich in library functions:** C has powerful standard libraries that provide functions for input/output, math, string handling, etc
8. **Memory management:** provides direct access to memory using pointers

Components of a C program

A typical C program is made of the following components:

- Keywords
- Preprocessor directives
- Functions
- Declaration statements
- Comments
- Expressions
- Input and output statements

Keywords

These are reserved words that have a special meaning in a language. The compiler recognizes a keyword as part of the language's built – in syntax and therefore it cannot be used for any other purpose such as a variable or a function name. C keywords must be used in lowercase. {C is case sensitive}. C language keywords include; auto, void, sizeof, signed, union, const, enum, break, default, long, unsigned, continue, extern, static, case, do, float, register, struct, volatile, else, double, for, return, switch, while, int, if, goto, short, typedef, char { they are 32 in number}

Preprocessor directives

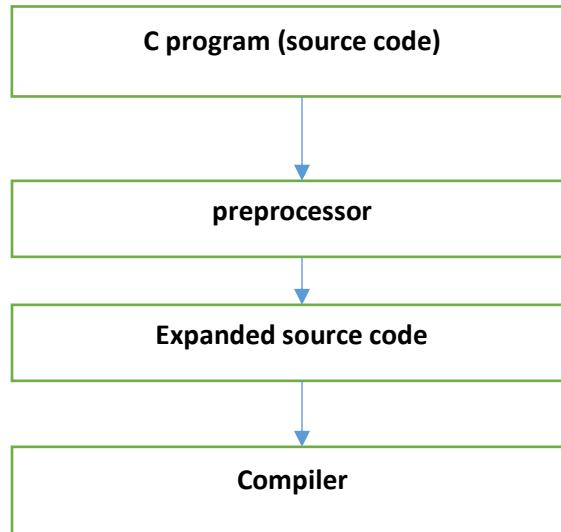
A preprocessor is a program that processes the source code before it is passed to the compiler.

The preprocessor works on the source code and creates an expanded source code.

The preprocessor offers several features called preprocessor directives or commands e.g. (#include).

Each preprocessor directive begins with # symbol.

Preprocessor directives can be placed anywhere in a program but most often placed at the beginning of the program, before function definitions.



Various preprocessor directives include:

1. File inclusion (#include)
2. Macro substitution (#define)
3. Conditional compilation (#ifdef, #else, #endif)
4. Miscellaneous directives (#pragma, #error)

File inclusion

Used to include one file into another file or it simply causes the entire content of one file to be inserted into the source code of another.

The preprocessor directive `#include` is an instruction to read in the contents of another file and include it within your program. (`#include filename`)

Header files contain details of functions and types used within the library. They must be included before the program can make use of the library functions. Header file is given an extension .h.

Library header file names are enclosed in angle brackets < >

User defined files can also be included. If the file to include is in the same folder as the parent file, then quotation marks are used e.g. `#include "file1"`

e.g.

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

```
#include<stdlib.h>
#include<string.h>
```

{c is a lightweight language. It relies on existing functions in the library header files for most activities e.g inputting or outputting data- printf and scanf functions are contained in the <stdio.h> header file }

Macro substitution (#define)

{is a feature of the C preprocessor that allows you to define short names or symbols that get replaced with specified text or code before compilation}

Allow for code substitution before compilation. Types of macros include: object-like macros and function-like macros. Object-like macros resemble data objects when they are used. They represent a symbolic constant or a simple value e.g. #define PI 3.14159 (#define identifier replacement text or value). When this line occurs in a file, all subsequent occurrences of identifier will be replaced by replacement text. Object-like macros are commonly used to give symbolic names to numeric constants. function-like macros resemble function calls. They are defined to accept arguments, similar to a function e.g. #define SQUARE (x). Symbolic constants enable the programmer to create a name for a constant and use that name throughout program execution. If this needs to be modified, it has to be done in the #define directive statement. By convention macros are written in uppercase.

Conditional compilation (#ifdef, #else, #endif, #elif)

{used to include or exclude parts of code during compilation, depending on certain conditions. This allows you to control which code is compiled, based on constants, macros, or system environments}

A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler. Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator.

A conditional in the C preprocessor resembles in some ways an if statement in C, but it is important to understand the difference between them. The condition in an if statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessing conditional directive

is tested when your program is compiled. Its purpose is to allow different code to be **included** in the program depending on the situation at the time of compilation.

Comments

Comments are non –executable program statements that are meant to enhance **program readability** and allow easier program maintenance. Any text between `/* */` is treated as a comment in C and therefore ignored by the compiler during compilation process. Comments can be in a single line or multiple lines. Sometimes comments can be used to deactivate parts of a source code – useful when testing a program.

```
/* multi line comment */  
  
// single line comment
```

Declaration statements

In C, all variables must be declared before they are used. A variable is an identifier of a memory location that holds data, which can change during program execution. Variable declarations ensure that appropriate memory space is reserved for the variables, depending on the data types of the variables.

Assignment and expression statements

An assignment statement uses the assignment operator “`=`” to give a variable on the operator’s left side the value to the operator’s right or the result of the expression on the right.

Functions

{**a function is a block of code that performs a specific task, has a name, and can be re-used throughout the program – give examples**}

All C programs consist of one or more functions, each of which contains one or more statements. In C, a function is a named **subroutine** that can be called by **other parts** of the program. Functions are the building blocks of C. in C program, there is always a main function through which other

functions can be called during program execution. When a source code is executed, the start point is the main function. From the main function the flow goes as per the programmer's choice. {main function is compulsory for any C program}

Input/output statements

Used to either accept user input from devices like the keyboard or output information through devices like monitors. The `printf()` function and `scanf()` function are **library functions** used to input or display information. The two functions are contained in the `<stdio.h>` library file.

Example 1

```
1. #include <stdio.h>
2. /* program to print hello world*/
3. Int main ( )
4. {
5. Printf("hello world");
6. Return 0;
7. }
```

Explanation

- 1: Header file containing standard functions
2. multi-line comment
3. Main function. {functions should always have a return type }
4. Open curly brace. Marks the start of executable statements
5. Function for displaying information enclosed in quotation marks
6. Shows that the program has executed successfully.
7. Ending curly brace. Marks the end of executable statements

Example 2

```
1. #include<stdio.h>
2. /* program reads and prints the same thing */
3. Int main ( )
4. {
5. Int number;
6. Printf("enter number:");
7. Scanf("%d",&number);
```

```
8. printf("number is %d\n", number);
9. Return 0;
10. }
```

Explanation

5. Declaration statement

7. Input statement using scanf () function

%d – format specifier

8. \n – used to insert a new line

Guidelines to good programming

- **Problem understanding & planning**
- Ensure that your program logic design is clear and correct. Avoid starting to code before the logic is clearly set out. Good logic will reduce coding time and result in programs that are easy to understand, error free and easily maintainable.

Proof read your program to check for any errors or omissions
- Dry run your design with some test data before running the code, then compare the two.
- Ensure you save a program any time you make changes
- **Code structure and readability.** Use meaningful names for variables to avoid ambiguity and enhance documentation for convenient program debugging, testing and maintenance. Indent code properly and consistently, keep functions short and focused on one task.
- **Comments and documentation.** Write comments to explain complex logic. At the start of each program or function, describe it's purpose, maintain external documentation – user guides, technical notes
- Always remember that C is case sensitive.
- **Maintainability:** write code that others (and your future self) can understand. Use consistent coding standards – naming, indentation, formatting
- **Continuous improvement:** review code regularly. Refactor messy or redundant code. Keep learning new techniques and best practices.

Variables

A variable is an identifier of a memory location whose value can change during program execution. In C language, a variable must be declared before it can be used. Variables can be declared at the start of any block of code.

A variable declaration begins with the **data type**, followed by **the name of one or more variables**.

Syntax: **datatype variable;** or **datatype variable1, variable2,.....variableN;**

e.g. int high, low, results[20];

Declarations can be spread out, allowing space for an explanatory comment.

Variables can also be **initialized** when they are declared. This is done by adding an equal's sign and the required value after the declaration. E.g.

```
int high =250; /* maximum temperature */  
int low = -40 ; /* minimum temperature*/  
int results[20]; /*series of temperature readings*/
```

Multiple declaration initialization

You can provide one value for variables initialized in one statement.

e.g. int x,y,z=0; is same as

```
int x=0;  
int y= 0;  
int z=0;
```

Rules of identifiers (Variable names)

1. Every variable in C must start with a letter or underscore and the rest of the name can consist of letters, numbers and underscore characters.
2. No spaces allowed. Must not contain a white space.
3. Cannot use a keyword e.g. main, while, switch
4. Length limit. Variable name should not be more than 31 characters.

NB: C is **case sensitive**. Upper case and lower case characters are being recognized as being different. E.g. **name** and **NAME** are different variable names.

NB: do not use uppercase when typing variable names. Its good programming practice to use lower case when typing variable names.

Examples of Valid and Invalid identifiers

Valid	invalid
Sum	7of9
C4_5	x-name
First_number	first number
_split_name	Axyz&

Basic data types

Variable data type indicates how much **memory** to set aside for the variable.

C supports **five** basic data types

Type	Meaning	keyword
Integer	Signed whole number	Int, short int and long int
Character	Character data	char
Float	Floating point numbers	float
Double	Double precision floating-point numbers	Double, long double
Void	Valueless	void

The basic data types above specify amount of space (bytes) to set aside, what can be stored in the space and what operations (addition, multiplication, etc) can be performed on those variables.

The “**int**” specifier

It is a type of specifier used to declare integer variables. E.g. to declare count as integer you would write:

```
int count;
```

Integer variables can only hold **signed whole numbers** (numbers with no fractional part).

Type	bytes	bits	minimum value	maximum value
Short int	2	16	-32768	32767

Int	4	32	-2147483648	2147483647
Long int	4	32	-2147483648	2147483647

The ‘**char**’ specifier

A variable of type **char** is 1 byte long and is mostly used to hold a single character. E.g. to declare ‘ch’ to be a character type you would write:

```
Char ch;
```

The ‘**float**’ specifier

It is a type specifier that is used to declare floating-point variables. These are numbers that have a whole number part and a fractional part e.g. 234.675. to declare ‘f’ to be of type float, you would write:

```
Float f;
```

Floating point variables typically occupy **4 bytes** or 32 Bits

The ‘**double**’ specifier

It is a type specifier that is used to declare double-precision floating point variables. These are variables that store float point numbers with a precision twice the size of a normal float value. To declare ‘d’ to be of type double you would write:

```
double d;
```

the variable of type “**double**” occupies **8 bytes** or 64 Bits

the “**long double**” specifier used as double specifier but occupies **10 bytes** or 80 Bits

Using printf() to output values

Printf() function can be used to display values of characters, integers and floating-point values. To do so, however, requires that you know more about the printf() function.

Consider the following statement;

```
Printf("this prints the number %d",99);
```

Upon execution of the above statement, ‘**this prints the number 9**’ will be displayed on the screen. This call to the printf() function contains two arguments. The first one is the quoted string and the other is the constant 99. Note that the arguments are separated from each other by a comma.

In general, when there is more than one argument to a function, the arguments are separated from each other by commas. The first argument is a quoted string that may contain either normal characters or **format specifiers** that begin with a percent (%) sign.

Normal characters are simply displayed as is on the screen in the order in which they are encountered in the string (reading left to right). A **format specifier**, on the other hand informs printf() that a **different type** item is being displayed. {In the above example, the ‘%d’, means that an integer is to be output in decimal format. The value to be displayed is to be found in the second argument. This value is then output at the position at which the format specifier is found on the string}.

The table below shows the various format specifiers that can be used when displaying data

Code	format
%c	character
%d	Signed decimal integers
%i	Signed decimal integers
%e	Scientific notation
%f	Floating point numbers
%lf	Double floating-point numbers
%s	String of characters
%u	Unsigned decimal integers
%x	Unsigned hexadecimal

Example 1

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

```

Void main ( )
{
Int num;
Num=100;
Printf("the value is %d",num);
}

```

The program above declares a variable called num. second, it assigns this variable the value 100. Finally, it uses printf() function to display ‘**the value is 100**’ on the screen.

Example 2

```

#include <stdio.h>

Int main ( )
{
Char ch;
Float f;
Double d;
Ch='X';
F = 100.123;
D= 123.009;
Printf("ch is %c\n", ch);
Printf("f is %f\n",f);
Printf("d is %f",d);
Return 0;
}

```

Width and precision-with printf function

You can use **width** and **precision** to determine format of output. **Width** specifies number of character spaces in printing the field. Precision (for floating point numbers) specifies how many **characters after the decimal point** should be displayed.

Example: printf("%5d%8.3f\n",753,4.1678);

Produces 753 4.168

Inputting values from the keyboard using scanf()

Scnf() function is used to read a value from a keyboard and store it in a named memory location.

General form:

Scnf("%d",&variablename);

The first argument to scanf() is a **format specifier** that determines how the second argument will be treated. In this case %d specifies that the second argument will be receiving an **integer** value entered in decimal format.

Consider:

Int num;

Scnf ("%d",&num);

Address operator (&): put before a variable e.g. &num. tells the computer to store the value read at the location or address of the variable.

Example 3

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

```
Int main ()
```

```
{
```

```
Int num;
```

```
Float f;
```

```
Printf("enter an integer:");
```

```
Scnf("%d",&num);
```

```
Printf("enter a floating point number:");
```

```
Scnf("%f",&f);
```

```
Printf("%d",num);
```

```
Printf("%f",f);
```

```
Return 0;
```

```
}
```

Example 4

```
#include <stdio.h>

Int main()
{
    Int len, width, area;
    Printf("enter the length:");
    Scanf("%d",&len);
    Printf("enter the width:");
    Scanf("%d",&width);
    Area=len*width;
    Printf("the area is %d",area);
    Return 0;
}
```

The above program computes the area of a rectangle, given its dimensions. It prompts the user for the length and width of the rectangle and then displays the area.