

Chapter  
1

# INTRODUCTION

*Get on the  
Fast Track!*



TM

**SYS-ED/  
COMPUTER  
EDUCATION  
TECHNIQUES, INC.**

**Objectives**

You will learn:

- C History and background of the C language.
- C Steps which comprise the basic load module creation steps - edit, compile, link, workbench, etc.
- C Concepts underlying preprocessors.
- C How to implement examples for a C program and structure.
- C Concepts of a function.
- C Rules of comments, identifiers and escape sequences.

---

## 1 Origins

The C programming language was developed at AT&T's Bell Laboratories in the early 1970's. It was written by a systems software engineer, Dennis Ritchie, in an AT&T UNIX environment.

C is the native language of the Unix operating system. C is particularly well suited to systems software.

Why is it called C? It was derived from an earlier language called B. C gradually made its way into commercial use and is now widely used.

When the IBM PC first appeared, the undisputed leader in the C compiler market was a CP/M based compiler known as BDS-C. BDS-C (which stands for *Brain Damaged Software*) featured a number of utilities, clean documentation and a modest runtime library with full source code. Most of all, C's major attribute was compilation speed.

Today the demand for compilation speed has been extended to include professional tools, optimized code and integrated environment. C runs in a wide range of operating systems including UNIX, DOS, OS/2, and OS/390.

---

## **2 Use of C**

C is an ideal programming language. The language has a block structure similar to that of earlier languages such as PL/1 and Pascal. Yet programs written in C are more concise and more efficient than those other languages.

C is a low level language that performs most high level language functions. It allows you to specify the details in the program's logic to achieve maximum computer efficiency.

It's also a relatively high-level language in that it hides the details of the computer's architecture to promote program efficiency.

This is the beauty of C; it is close to machine-oriented languages and yet not as abstract as problem-oriented languages such as FORTRAN and COBOL.

It is fast, flexible, and portable.

---

### **3 C Applications**

C is a multi-purpose language that can be used for many applications:

- C Numerical programs
- C Text-processing programs
- C Data base programs
- C Systems programs
- C Communications

Most of the popular software packages written today are in C. This includes MS DOS, OS/2, Microsoft Windows, Microsoft Word, Microsoft Excel, dBase IV, Aldus Pagemaker, Paradox, and Quattro Pro to name a few.

C is available for microcomputers, minicomputers and mainframe computers.

---

## **4 C Features**

C's major strengths include the speed and efficiency of executable programs, as well as modularity and portability of programs.

Modularity allows you to break a large program into small, manageable pieces that can be reused in other programs.

C is also quite portable; it allows you to build programs from small functions and library modules that have already been compiled. Such library modules can contain C functions specific to a particular machine.

This ability to build and compile independent function libraries, allows you to write programs in a modular fashion. Large applications can be broken into smaller units. These individual modules are programmed and modified separately providing for greater ease in testing and transporting the code to different machines.

Modularity makes your programs easier to read and understand. Details of a program can be hidden in separate files, which only need be read by those who require detailed information into what the program is doing.

C is portable. It is easy to get programs to run on other computers or in other operating environments. Portability is assisted by the preprocessor, which allows you to substitute information specific to a particular machine when compiling or running a program.

C is a smaller language than Pascal, Modula-2 or Basic. It has only about 30 keywords, or reserved words, although it has many operators or symbols that cause specific actions.

There are actually more operators than keywords. This combination of many operators and a small vocabulary allows you to use C as a type of construction set.

Because many of C's actions are very close to assembly-level activity, using C can help increase the speed of your program, making it easy to achieve concise, optimized code. C lets you get very close to the memory cells and registers in the computer, and is ideal for complex projects and for programs that need to take advantage of the system on which they are running.

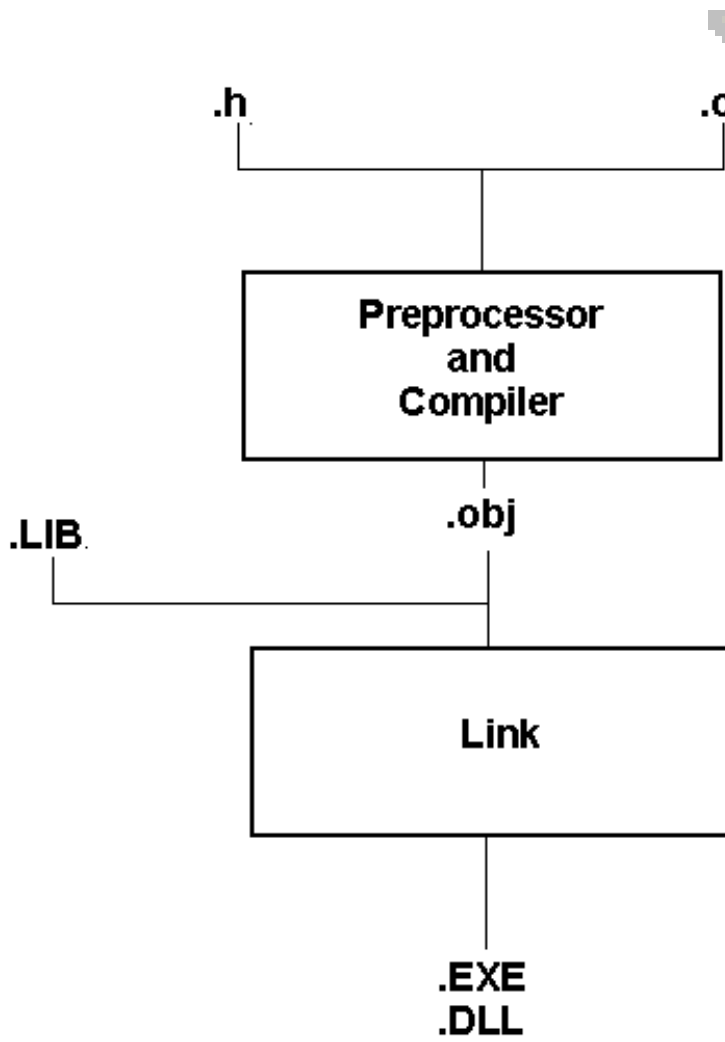
---

**5 Program Stages**

Program Stage	Description
editor	Input from terminal.  Output is a source code file containing C source code and possibly preprocessor commands.
preprocessor	Input is source code file.  Output is source code file with macros expanded and other files included as indicated by preprocessor commands.
compiler	Input is source code file with macros expanded.  Output is object language code.
assembler	Input is assembly language source code file.  Output is relocatable object code.
linker	Input is relocatable object code modules from your programs and from C library.  Output is executable code.

In most systems the preprocessor, assembler, and linker are invoked automatically by a single command.

5.1 Compile Process



---

## 5.2 Preprocessor

The C preprocessor provides a means of modifying a C source file in various ways before the actual compilation process takes place.

Preprocessor directives form what can almost be considered a language within the language of C.

Essentially, the preprocessor makes substitutions in your source file, or preprocesses the file. This capability lets you extend the C language to make it easier or more useful for particular purposes, while indirectly making your programs more portable.

The preprocessor is conceptually distinct from the compiler, but it isn't necessarily a separate program. Preprocessor substitutions may be made before or during the compilation process. The important concept is that the compiler treats your source as if you had actually typed the material the preprocessor substitutes for the appropriate identifiers.

Normal program statements are instructions to the microprocessor; preprocessor directives are instructions to the compiler.

One of the preprocessor's major functions is handling simple substitutions, where the replacement is just a single value, such as a number or a string constant. Other functions are file inclusion and conditional compilation.

The preprocessor allows us to include external text files in our source file just prior to compilation. The preprocessor allows us to use macros and macro processor utilities. The preprocessor improves the portability of the code.

The C preprocessor expands certain shorthand forms in the source code. Its output, the expanded source code, is sent to the C compiler proper. What comes out of the compiler is the original program translated into the computer's assembly language.

### 5.3 Compiling

To run on a computer, the program must be translated into binary numbers understandable to the computer's Central Processing Unit (CPU).

These binary numbers are called machine language, since the computer machine is able to understand them.

It is the job of a compiler program to translate the human-readable, source code version of the program into a machine-readable, object code version.



---

## 5.4 Linking

Linking is necessary because you may not want to compile all of your program at the same time. For instance, you may be writing a large program, with parts of it working and debugged, while other parts are under development. Since it may take a fairly long time to compile a large program, (from seconds to many minutes, depending upon the program size), you may want to recompile each time only those parts under development.

Many compiled languages, with C as the primary example, come with library routines that can be added to your program. It is necessary to have a way to link together previously compiled library and program files with recently compiled files to produce a complete program. It is for this reason that the compiler generates an intermediate type of file called an OBJECT FILE.

The linker combines all the necessary object files, both library and user-written, to produce a final executable program.

### Linker

Combines program code with functions found in the C compiler's libraries. Some C vendors supply linkers; others use operating systems linkers, such as DOSLINK.EXE.

---

## 5.5 Subdirectories

### **BIN Subdirectory**

The programs that actually do the work of compiling your program, linking it, and carrying out other tasks are stored in a Subdirectory named BIN.

### **INCLUDE Subdirectory**

The INCLUDE Subdirectory contains the header files, also called include files. These files have the .h file extension. In C, many programs include the header file `STDIO.H`, which contains the definitions for standard input/output functions such as `printf()`.

### **LIB Subdirectory**

This contains the library files. C compilers have standard libraries consisting of functions used to perform commonly required tasks and I/O operations. Library files are groups of routines for performing a variety of specific tasks, such as I/O, math, or data conversion. Library functions are precompiled routines added to your program by the linker.

### **TMP Subdirectory**

The TMP Subdirectory contains temporary files that the compiler generates while it is working. When the compilation process is complete these files are erased.

---

## 6 C Program Structure

```
/*Sample C program to illustrate three components of a program:
   program, functions, and statements.

*/
#include <stdio.h>
void main(void)          /* main program */
{
    display_message();  /* call function display_message
                        call function display_nr, with
                        argument 7 */
    display_nr (7);
}

/* a simple function writes a message to the screen. */
void display_message()
{
    printf("hello world:");
}

/* a simple function; writes a number to screen. */
void display_nr(int value)
{
    printf("%d", value);
}
```

The above program produces the following output:

```
hello world: 7
```

The above program contains three functions:

```
display_message()
display_nr()
main()
```

---

## 6.1 C Program Description

- C C programs consist of functions, which are similar to subroutines in other languages. Functions consist of statements.
- C Statements serve to define or express something, or to carry out some action.
- C Consists of one or more functions. Must contain one main() function.
- C The actual program starts with function main().
- C Each statement is a function call, a or a command to execute a function.
- C Each statement ends in a semi-colon (;).

---

## 7 C Functions

An independent subroutine function allows modular program creation. The main program will contain calls to functions, which then perform operations, return values, call other functions and generally act as simple stand-alone programs.

Extensive use of functions increases portability.

A C function is a statement written to perform a specific action; it can be compiled by the 'C' compiler.

A C function is a specialist that you call when the need arises.

```
function-name (argument declaration, if any)
{
    statements comprising
    body of function
}
```

---

**7.1 C Function Description**

Command	Explanation
function-name	Indicates the beginning of function. Must be coded. First 6 characters must be unique.
parentheses ( )	Includes argument list passed to function, if any. Must be coded after function name even if the function doesn't accept any argument(s).
argument	Declares variables being passed to declarations function. Must be coded, if there are arguments passed.
open braces {	Indicates beginning of function statement. Must be coded after function argument list.
body of	Statements which describe action to be performed by the function.  Two types of statements: C     Declarations-defining variables to be used within the function. C     Commands to execute the action needed.
closing braces }	Indicates end of function statements.  Must be coded.

---

## 7.2 The main() Function

C programs consist of functions. The function called main() is the one to which control is passed when the program is executed.

No matter how many functions there are in a C program, the main() function is the one to which control is passed from the operating system when the program is run; it's the first function executed.

Main() function starts things off, it is almost always performed first. It controls the program by giving the actual order in which the processes occur.

```
/* A program that does nothing */  
main()  
{  
}
```

```
/******  
/*  
* hello.c  
*  
* This is a C language programz that prints  
* a message to standard output  
* (stdout is usually your terminal screen).  
*  
*/  
  
#include      <stdio.h>  
  
void main(void)  
{  
    printf("Hello, world\n");  
}  
  
/* end of program */
```

---

### 7.3 Function Definition

Every function definition has two parts: a function header and a body. The header goes first and defines the function's identifier and arguments; the body defines what the function does.

#### Delimiters

Following the function definition are braces that signal the beginning and ending of the body of the function. Braces { } are used to delimit other blocks of code as well as functions.

Constants and variable values are manipulated by operators. The braces in the source code help the compiler determine the order in which it will execute the statements.

---

## 8 Program Statements

Program statements are an instruction to the compiler to create machine-language code in order to perform a certain action. C contains a variety of statements that regulate the flow of a program. It is important to notice the semicolon at the end of the statement. Every complete statement in C must be terminated with a semicolon.

### Code Block

A series of statements, enclosed within braces, or {}, that execute together as a unit.

Since you can put as many whitespace characters as you want in your program, it is a universal practice to use these characters to make the program easier to read. The whitespace characters (space, tab, newline) are invisible to the compiler.

Stretching the code out vertically makes the results more comprehensible.

Aligning matching braces is important in ensuring that all opening braces will be followed by closing braces.

Indentation of blocks of code enclosed in braces {} is an important aspect of making your C program readable.

---

## 9 Output Functions

### **printf()**

Causes the phrase contained in the parentheses which are within quotes to be printed to the screen.

### **puts()**

Writes a string to the screen followed by a newline character (\n).

### **putchar()**

Writes a single character to the screen and does not add a \n.

Since C is case sensitive, it is important to write all function statements in lowercase letters. PRINTF(), and Printf(), are not the same as printf(). It is a common practice to keep everything in lowercase letters, for ease of typing.

---

## 10 Identifiers

All the names used in a C program are referred to as identifiers. The names assigned to constants, data types, variables, and functions are known as identifiers.

The identifier is used for constants, variables or function names. All identifiers must follow the same rules; the identifiers must be made up of letters and digits only, and the first character must be a letter. The underscore `_` can also serve as a letter. It helps separate the parts of long descriptive identifiers, since blanks inside identifiers are not allowed. Letters in upper case are not equivalent to the same letters in lower case.

Valid Characters for C Identifiers:

- C Uppercase and lowercase letters ('A'-'Z', 'a'-'z')
- C The underscore character ('\_')
- C Digits ('0'-'9')
- C Identifiers are case sensitive
- C The first 31 characters of an identifier are significant.

You should not use an underscore as a first letter of an identifier. Identifiers beginning with an underscore can cause conflicts with the names of system routines or variables and produce errors. Programs containing names beginning with leading underscores may not be portable. Digits are not allowed as the first character of an identifier.

C is a case sensitive language. NUMBER, Number, and number are all different identifiers. You are free to use any case when naming your constants and variables. However, it is generally not a good practice to have variables that differ only in the case of having a few letters in the upper or lowercase in their identifier.

Many C programmers use only lowercase letters and underscores for their identifiers with one major exception. By convention, identifiers containing only uppercase characters indicate that the identifier is associated with a constant value. This is not a requirement, only a convention.

When naming the program, most compilers need all C source code files to end with the suffix `.c`. You must avoid the use of reserved words, as these have a special meaning to the compiler.

---

**11 Reserved Words in C**

auto	else	long	switch
break	entry	register	typedef
case	enum	return	union
char	extern	short	unsigned
const	float	signed	void
continue	for	sizeof	volatile
default	goto	static	while
do	if	struct	
double	int		

---

## 12 Escape Sequences

The backslash \ signals to the compiler that the letter following has special significance.

Escape Sequence	Description
\n	newline
\t	tab
\b	backspace
\r	carriage return
\f	formfeed
\'	single quote
\"	double quote
\\	backslash
\xdd	ASCII code in hexadecimal notation (each d represents a digit).
\ddd	ASCII code in octal notation (each d represents a digit).

---

## 13 Comment Lines

It is helpful to put comments into the source code file which improve the readability of programs. These comments are invisible to the computer.

A comment begins with a two-character symbol slash-asterisk (/\*) and ends with an asterisk-slash (\*/).

Anything between a /\* and a matching \*/ is ignored by the compiler, and allows us to write comments and documentation in the body of the program. Comments can cover one or many lines.

```
/* This is a comment. It is ignored by the compiler */
```

Since C ignores whitespace characters, it is possible for comments to flow over two or more lines.

```
/* This is a  
multiline comment.  
*/
```

Asterisks may be placed at the beginning of each line in a multiline comment. This is used for aesthetic reasons, the asterisks are not comment symbols and are ignored by the computer.

```
/* This is a  
* multiline  
* comment.  
*/
```

C does not allow comments to be nested.

The following is not allowed:

```
/* Outer comment /* inner comment */ more outer comment */
```

It is easy to make errors with comment symbols, and the results of these errors can be particularly problematic to debug.

For instance if you omit the close-comment symbol `*/` at the end of a comment, the compiler will assume the whole program is a comment. It will not compile and will not give you any error message.