Programming in C

1. INTRODUCTION

1.1 History of C

A history of the C language begins with a discussion of the UNIX operating system, since both the system and most of the programs that run on it are written in C. However, C is not tied to UNIX or any other operating system or machine. This codevelopment environment has given C a reputation for being a **system programming language** because it is useful for writing compilers and operating systems. It can also write major programs in many different domains.

UNIX was originally developed in 1969, on what would now be considered a small DEC PDP-7 at Bell Laboratories in New Jersey. UNIX was written entirely in PDP-7 Assembly language. By design, this operating system was intended to be "**programmer-friendly**," providing useful development tools, lean commands, and a relatively open environment. Soon after the development of UNIX, Ken Thompson implemented a compiler for a new language called B.

At this point it digress to the origins and history behind Ken Thompson's B language. A true C ancestry would look like this:

Algol 60	Designed by an international committee in early 1960
CPL	(Combined Programming Language) developed at both Cambridge and the University of London in 1963
BCPL	(Basic Combined Programming Language) developed at Cambridge, by Martin Richards, in 1967
B	Developed by Ken Thompson, Bell Labs, in 1970
C	Developed by Dennis Ritchie, Bell Labs, in 1972
ANSI C	The American National Standards Institute committee is formed for the purpose of standardizing the C language, in 1983
C++	C with Classes is refined and renamed to C++ in 1983
ANSI C++	ANSI begins work on the C++ standard in 1990. First draft of the ANSI C++ standard is released in 1995. ANSI approves the final draft of the C++ standard in 1998.

Algol appeared only a few years after FORTRAN was introduced. This new language was more sophisticated and greatly influenced the design of future programming languages.

The inventors of CPL intended to bring Algol's lofty intent down to the realities of an actual computer. But like Algol, CPL was big. This made the language hard to learn and difficult

to implement and explains its eventual downfall. Still clinging to the best of what CPL had to offer, BCPL's creators wanted to boil CPL down to its basic good features.

Bringing the discussion back to the origins of B, when Ken Thompson designed the B language for an early implementation of UNIX, he was trying to further simplify CPL. He succeeded in creating a very sparse language that was well suited for use on the hardware available to him. However, both BCPL and B may have carried their streamlining attempts a bit too far. They limited languages, useful only for certain kinds of problems.

For example, shortly after Ken Thompson implemented the B language, a new machine was introduced, the PDP-11. UNIX and the B compiler were immediately transferred to this machine. While the PDP-11 was larger than its PDP-7 predecessor, it was still quite small by today's standards. It had only 24K of memory, of which the system used 16K, and one 512K fixed disk. Some considered rewriting UNIX and B, but the B language was slow due to its interpretive design. There was another problem: B was word-oriented while the PDP-11 was byte-oriented. For these reasons, work began in 1971 on a successor to B, appropriately named C.

Dennis Ritchie is created with creating C, which restored some of the generality lost in BCPL and B. He accomplished this with his shrewd use of data types, while maintaining the simplicity and computer contact that were the original design goals of CPL.

Many languages that have been developed by a single individual (C, Pascal, LISP, and APL) have a cohesiveness missing from languages developed by large programming teams (Ada, PL/I, and Algol 68). In addition, a language written by one person typically reflects the author's field of expertise. Dennis Ritchie was noted for his work in systems software - computer languages, operating systems, and program generators. With C having a generic link to its creator, one can quickly understand why C is a language of choice for systems software design. C is a relatively **low-level language** that lets you specify every detail in an algorithm's logic to achieve maximum computer efficiency. But C is also a **high-level language** that can hide the details of the computer's architecture, thereby increasing programming efficiency.

1.2 Strengths of C

All computer languages have a particular look. APL has its hieroglyphic appearance, assembly language has its columns of mnemonics, Pascal has its easily read syntax, and then there's C. Many programmers new to C will find its syntax cryptic and perhaps intimidating. C contains few of the familiar and friendly English-like syntax structures found in many other programming languages. Instead, C has unusual-looking operators and a plethora of pointers. You will quickly grow used to C's syntax. New C programmers will soon discover a variety of language characteristics whose roots stem back to its original hardware/software progenitor.

The following sections highlight the strengths of the C language.

Small Size

There are fewer syntax rules in C than in many other languages, and you can write a top-quality C compiler that will operate in only 256K of total memory. There are actually **more operators and combinations of operators in C than there are keywords**.

Language Command Set

As you would therefore expect, C is an extremely small language. In fact, **the original C language contained a mere 27 keywords**. The ANSI C standard has an additional 5 reserved words. Borland C++ added 45 mere keywords. This brings the total keyword count to 77.

C does not include many of the functions commonly defined as part of other programming languages. For example, C does not contain any built-in input and output capabilities, nor does it contain any arithmetic operations (beyond those of basic addition and subtraction) or string-handling functions. Since any language lacking these capabilities is of little use, C provides a rich set of library functions for input/output, arithmetic operations, and string manipulation. This agreed-upon library set is so common that it is practically part of the language. One of the strengths of C, however, is its loose structure, which enables you to recode these functions easily.

Speed

The C code produced by most compilers tends to be very efficient. The combination of a small language, a small run-time system, and a language close to the hardware makes many C programs run at speeds close to their assembly language equivalents.

Not Strongly Typed

Unlike Pascal, which is strongly typed language, C treats data types somewhat more loosely. This is a carryover from B, which was also an untyped language. This flexibility allows you to view data in different ways. For example, at one point in a program the application may need to see a variable as a character and yet for purposes of upcasing (by subtracting 32) may want to see the same memory cell as the ASCII equivalent of the character.

A Structured Language

C includes all of the control structures you would expect of a modern language. This is impressive when considering that C predated formal structured programming. C incorporates ***for*** loops, ***if*** and ***if-else*** constructs, ***case*** (switch) statements, and ***while*** loops. C also enables you to compartmentalize code and data by managing their scope. For example, C provides local variables for this purpose and **call-by-value** for subroutine data privacy.

Support of Modular Programming

C supports the concepts of separate compilation and linking, which allows you to **recompile only the parts of a program** that have been changed during development. This feature can be extremely important when you are developing large programs, or even medium-sized programs on slow systems. Without support for modular programming, the amount of time required to compile a complete program can make the change, compile, test, and modify cycle prohibitively slow.

Easy Interface to Assembly Language Routines

There is a well-defined method of calling assembly language routines from most C compilers. Combined with the separation of compilation and linking, this makes C a strong contender in applications that require a mix of high-level and assembler routines. You can also integrate C routines into assembly language programs on most systems.

Bit Manipulation

In systems programming, you often need to manipulate objects at the bit level. Because C's origins are so closely tied to the UNIX operating system, the language provides a rich set of bit manipulation operators.

Pointer Variables

An operating system must be able to address specific areas of memory. This capability also enhances the execution speed of a program. The C language meets these design requirements by using pointers. While other languages implement pointers, C is noted for its ability to perform pointer arithmetic. For example, if the variable ***index*** points to the first element of an array *student_records*[*index+1*] will be the address of the second element of student_records.

Flexible Structures

In C all arrays are one-dimensional. Multidimensional arrangements are built from combinations of these one-dimensional arrays. You can join arrays and structures (records) in any manner, creating database organizations that are limited only by your ability.

Memory Efficiency

C programs tend to be very memory efficient for many of the reasons that they tend to be fast. The lack of built-in functions saves programs from having to include support for functions that are not needed by a particular application.

Portability

Portability is a measure of how easy it is to convert a program that runs on one computer or operating system to run on another computer or operating system. Programs written in C are currently among the most portable in the computer world. This is especially true for mini and microcomputers.

Special Function Libraries

There are many commercial function libraries available for all popular C compilers. There are libraries for graphics, file handling, database support, screen windowing, data entry, communications, and general support functions. By using these libraries, you can save a great deal of development time.

1.3 Weaknesses of C

There are no perfect programming languages. Different programming problems require different solutions. It is the software engineer's task to choose the best language for a project. On any project, this is one of the first decisions that you need to make, and it is nearly irrevocable once you start coding. The choice of a programming language can also make the difference between a project's success or failure.

Not Strongly Typed

The fact that it is not strongly typed is one of C's strengths but is also one of its weaknesses. Technically, typing is a measure of how closely a language enforces the use of variable types (for example, integer and floating point are two different types of numbers). In some languages, you cannot assign one data type to another without invoking a conversion function. This protects the data from being compromised by unexpected roundoffs.

As mentioned, C will allow an integer to be assigned to a character variable or vice versa. This means that you have to manage your variables properly. For experienced programmers, this task presents no problem. However, novice program developers may want to remind themselves that mismatched data type assignments can be the source of side effects.

A side effect in a language is an unexpected change to a variable or other item. Because C is a weakly typed language, it gives you great flexibility to manipulate data. For example, the assignment operator, =, can appear more than once in the same expression. This feature, which you can use to your advantage, means that you can write expressions that have no definite value. If C had restricted the use of the assignment and similar operators, or had eliminated all side effects and unpredictable results, C would have lost much of its power and appeal as a high-level assembly language.

Lack of Run-Time Checking

C's lack of checking in the run-time system can cause many mysterious and transient problems to go undetected. For example, the run-time system would not warn you if your application **exceeded an array's bounds**. This is one of the costs of streamlining a compiler for the sake of speed and efficiency.

1.4 Why C?

C's tremendous range of features - from bit-manipulation to high-level formatted I/O - and its relative consistency from machine to machine have led to its acceptance in science, engineering, and business applications. It has directly contributed to the wide availability of the UNIX operating system on computers of all types and sizes.

Like any other powerful tool, however, C imposes a heavy responsibility on its users. C programmers quickly adopt various rules and conventions in order to make their programs understandable both to themselves and to others. In C, programming discipline is essential. The good news is that it comes almost automatically with practice.

1.5 The ANSI (American National Standard Institute) Standard

The ANSI committee has developed standards for the C language. This describes some of the significant changes suggested by the committee. A number of these changes are intended to increase the flexibility of the language while others attempt to standardize features previously left to the discretion of the compiler implementor.

Previously, the only standard was **The Programming Langauge** by Brian Kernighan and Dennis Ritchie. This was not specific on some language details, which led to a divergence among compilers. The ANSI standard strives to remove these ambiguities. Although a few of the proposed changes could cause problems for some previously written programs, they should not affect most existing code.

The ANSI C standard provides an even better opportunity to write portable C code. The standard has not corrected all areas of confusion in the language, however, and because C interfaces efficiently with machine hardware, many programs will always require some revision when you move them to a different environment. The ANSI committee adopted as guidelines several phrases that collectively have been called the "spirit of C."

1.6 An Overview of C

In this you will be introduced to the C language so that you can get a feeling as to what programming in C is all about. But what better way to gain an appreciation for this language than by taking a look at an actual program written in C?

To begin with, let's pick a rather simple example - a program that displays the phrase "**Programming is fun**" at the terminal. The complete program is

```
# include <stdio.h>

void main ( )
{
        printf ("Programming is fun" );
}
```

In the C programming language, lowercase and uppercase letters are distinct. In most other programming languages, such as FORTRAN, COBOL, PL/I or BASIC, uppercase letters are used exclusively. Also, unlike many other programming languages, C does not particularly care where on the line you begin typing. Some languages, such as FORTRAN and COBOL, are very fussy about such things, but in C you may begin typing your statement at any position on the line. This fact can be used to advantage in developing programs that are easier to read.

Returning to our first C program, if we were to type this program into the computer, issue the proper commands to the system to have it compiled, and then execute it, we could expect the following results or output to appear at the terminal:

```
Programming is fun
```

Let's take a closer look at our first program. The first line of the program informs the system that the name of the program is **main**. *main* is a special name used by the C system that indicates precisely **where the program is to begin execution**.

The open and closed parenthesis immediately following *main* specify **that no "arguments" or parameters are expected by this routine**. (This concept will be explained in more detail in later.)

Now that we have told the system that the name of our program is *test*, we are ready to specify what function this routine is to perform. This is done by enclosing all program statements of the routine within a pair of braces. (For those readers who are familiar with the Pascal programming language, the braces in C are somewhat analagous to the BEGIN and END block declarations of that language). All program statements included between the braces will be taken as part of the *main* routine by the system.

All program statements in C **must be terminated by a semicolon**. This is the reason for the semicolon that appears immediately following the closed parenthesis of the *printf* call.

Now that we have finished discussing our first program, why don't we modify it to also display the phrase "**And programming in C is even more fun**". This can be done by the simple addition of another call to the *printf* routine, as shown below.

```
# include <stdio.h>

void main ( )
{
        printf ("Programming is fun\n");

        printf ("And programming in C is even more fun\n");
}
```

If we type the above program and then compile and execute it, we can expect the following output at our terminal:

```
Programming is fun
And programming in C is even more fun
```

A string constant in C is a series of characters surrounded by double quotes. This string is an argument to the function *printf* ( ), which controls what is to be printed. The two characters \n at the end of the string **(reads as "backslash n")** represent a single character called newline. It is a nonprinting character. Its effect is to advance the cursor on the screen to the beginning of the next line.

As you will see from the next program example, it is not necessary to make a separate call to the *printf* routine for each line of output.

```
# include <stdio.h>

void main ( )
{
  printf ("Programming is fun\nAnd programming in C is even more fun\n");
}
```

Study the program listed above and try to predict the results before examining the output.

A comment statement is initiated by the two characters / and *****. These two characters must be written without any intervening spaces. To end a comment statement, the characters * and / are used, once again without any embedded spaces. All characters included between the opening **/*** and the closing ***/** are treated as part of the comment statement and are ignored by the C system.

The Intelligent use of comment statements inside a program cannot be overemphasized. Many times a programmer will return to a program that he coded perhaps only six months ago, only to discover to his dismay that he cannot for the life of him remember the purpose of a particular routine or of a particular group of statements. A simple comment statement judiciously inserted at that particular point in the program might have saved a significant amount of time otherwise wasted on rethinking the logic of the particular routine or set of statements.

It is a good idea to get into the habit of inserting comment statements into the program as the program is being written, or typed into the computer. There are three good reasons for this. First, it is far easier to document the program while the particular program logic is still fresh in one's mind than it is to go back and rethink the logic after the program has been completed. Second, by inserting comments into the program at such an early stage of the game, the programmer gets to reap the benefits of the comments during the debug phase, when program logic errors are being isolated and debugged. A comment cannot only help the programmer read through the program, but it can also help to point the way to the source of the logic mistake. Finally, it has yet to be to discover a programmer who actually enjoyed documenting a program. In fact, once you have finished debugging your program, you will probably not relish the idea of going back to the program to insert comments. Inserting comments while developing the program will make this sometimes tedious task a bit easier to swallow.

The C system contains a standard library of functions that can be used in programs. We included the header file **stdio.h** because it provides certain information to the compiler about the function *printf* ( ).

Dissection of the first program

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

We have been using the file inclusion right from the start. A **preprocessor** is build into the C compiler. When the command to compile a program is given, the code is first preprocessed, and then compiled. **Lines that begin with a # communicate with the preprocessor**.

The directive **# include** instructs the preprocessor to replace the current line with the entire contents of the specified file. (In this first program # include line causes the preprocessor to include a copy of the header file **stdio.h** at this point in the code.) This header file is provided by the C system. We have included this file because it contains information about the *printf*( ) function.

If we don't provide a full path name, the preprocessor searches for the file in various locations determined by the **# include** form used. The form

 # include < *filename* >

has the preprocessor look in only those locations specified by the **Options / Directories / Include Libraries** menu entry. It is used to access system-supplied include files.

The alternative form

 # include "*filename*"

is used for include files that we create. It looks in a default directory first, and searches the only if the file is not found locally. This form is used only with local include files. Preprocessing stops if an include file cannot be found. Included files can include other files; most preprocessors allow at least five levels of nesting. Of course, an included file shouldn't include itself or any file that included it. File inclusion allows us to create a file of useful definitions that we can include in all our programs.