C – Past, Present, and Future – A Perspective

K Bhaskar

K Bhaskar is presently Technical Officer in the Department of Electrical Engineering, Indian Institute of Science. Prior to this he was Senior Systems Programmer at a public sector industry. His interests are in the area of Operating System, Programming Languages and Networking. He has vast experience ranging from programming on mainframes to modern day computers.

This article looks at the development of the C language from its inception to the present day. The salient features of the language, which are of importance both historically as well as from the point of view of the industry, are brought out. Also, the impact of C as a cross-platform developmental language is highlighted.

1. Introduction

Ken Thompson and Dennis Ritchie at The Bell Laboratories originally designed the C Language, while working on the PDP-11 during 1969–1973. Its name is derived from the Language B, which in turn was a stripped-down version of BCPL (Basic Combined Programming Language).

Historically, C was designed to be very *close* to *assembly language*, and still provide constructs, which were considered to be a part of a high-level language. Hence, most of the features that appeared in the first version of C had features, which were simple and easy to implement in assembly languages. C was therefore a procedural systems implementation language, with many features for low-level memory access, with minimal run-time support but still maintained a cross-platform programming environment. C has since then been the choice of many programmers to bolster cross-platform developments.

C has since then undergone several changes and many C language standards have come into existence. We broadly classify the period of development of C into three stages.

1. K&R C: From 1973–1989, when Kernighan and Ritchie primarily provided the standards.
2. ANSI and ISO C: From 1989–1999, a superset of the K&R C, with many improvements and support for international characters and locale.

Keywords
C language, ANSI C, C99, C11, programming language.

3. C99 and C11: Post-ANSI to the present day, primarily incorporating improvements in libraries and support for multithreading and modern CPUs with hyperthreading.

2. K&R C

K&R C (as it started in 1973) is a low-level friendly, and yet a structured programming language primarily aimed at implementing systems programming environments. It provides block definition using { and } as separators. All the definitions, declarations and code inside the block are local to that block by default and appear only in that order. Code is organized into functions with optional parameters and an optional return type. Functions cannot be defined inside another function, unlike in a block structured programming language such as Pascal. In order to hide a function, and prevent other functions from using it, C provides namespace management via scoping rules[1].

Code is organized into functions with optional parameters and an optional return type. Functions cannot be defined inside another function, unlike in a block structured programming language such as Pascal.

[1] Names of variables, functions whose existence and visibility to other parts of the program are managed by scoping rules in C.

C handles all primitive types such as integer (`int`, `short`, and `long`), floating point (`float`, `double`), character, and aggregated types such as arrays and structs. Some of the other salient features of K&R C are listed below.

- Each of the primitive types can be prefixed by `unsigned` or `signed` to alter the behaviour.
- Scope of these variables can be controlled by prefixing `static`, `register`, or `union`.
- Pointers to all types of data allow low level direct memory access.
- Address arithmetic is performed properly to reflect correct addresses, when pointing to different types of data types.
- Homogeneous data structures such as arrays are implemented using pointers.
- Heterogeneous data structures are implemented using structs.
- Type defs allow user defined types of variables.

In K&R C, function declarations do not use any type definitions for arguments. Hence strict type checks were not part of most C compilers (in those days). Compilers would only give warning

In fact, the heart of C is contained in expressions. An expression in C is an entity, which *returns a value.*

messages, if inconsistent usage of functions were noticed. If the functions were spread across files, then other utilities such as lint were used to detect them. As days progressed, mainly due to necessity, some extra features were included, such as, functions returning void, structs, or unions, instead of pointers to these data types.

Data types such as `int, long, short, float, double,` and `char` can be used in expressions. In fact, the heart of C is contained in expressions. An expression in C is an entity, which *returns a value.* For example, 1, 2+3, x+y, z=x*y+b, are all expressions in C. In fact, C has very few keywords (`if`, `else`, `for`, `do`, `while`, `until`, etc.), and a very rich set of operators (+, -, /, &, | , &&, ~, etc.). The idea behind keeping the number of keywords small was to easily and efficiently implement a C compiler on the target machine with minimal resources and maximum ease.

Operators can be broadly classified into several categories such as (list is not exhaustive):

- Arithmetic operators, e.g., +, -, /,*, ++, –, =, etc.
- Logical operators, e.g., <, >, <=, >=, &&, ||, etc.
- Bitwise operators, e.g., &, |, ~, ^, >>, <<, etc.
- Pointer operators, e.g., *, &, [,], ->, etc.

These operators can be used in any expression anywhere, even in a conditional statement. In fact, this is one of the major pitfalls of C but also contributes to the power of C. A statement such as, if (x=y+z) `statement`; first computes y + z, then assigns it to x, and returns the value of x as the value of the expression (x = y+z). If this value is *non-zero*, then it is taken as *true*, else *false*, and accordingly `statement` is executed. This particular statement does not read well, since it misleads a reader to interpret the condition as, *'x is equal to'*, rather than *'x is assigned to'*. C resolves this issue by using "==" as 'equal to' logical operator. There is a hierarchy implemented in the evaluation of these operators, which is the usual well-known hierarchy, but there are nuances and issues (see Section 2.1 on shortfalls of K&R C).

Operators can be used in any expression anywhere, even in a conditional statement. In fact, this is one of the major pitfalls of C but also contributes to the power of C.

Address arithmetic is implemented in C and it is used to implement arrays. There is no keyword called array, and as such there is no array implementation in C. Rather, it is implemented using pointers with proper address arithmetic to reflect the correct position of the elements. A syntax such as, `int a[10];` is more an allocation strategy rather than an array implementation. Accessing the array element `a[i]` is no more than `*(a+i)`, which means: get the content of location (&a[0]+i), with proper scaling done to i (i * `sizeof` element of a).

Address arithmetic is implemented in C and it is used to implement arrays. There is no keyword called array, and as such there is no array implementation in C. Rather, it is implemented using pointers with proper address arithmetic to reflect the correct position of the elements.

Heterogeneous data types such as `struct` allow the user to aggregate data of different types. Structs contribute to the main power of C, which is used in all system implementations. Accessing the member of a struct is through the **'.'** operator. C also has the union operator, to allow sharing of the same space and/or redefine the same area by another data variable, thus providing a mechanism to access the same information in different ways (e.g., int and char). K&R C does not permit assigning one struct or one array to another. It has to be carried out externally by library calls. Even a data structure such as a *string* is implemented as an array of char terminated by a *NULL*. All string manipulations are carried out externally by a host of library calls.

Scoping deals with the birth, death and life of a variable along with its visibility to other parts of the program. Namespace deals with pools of names of variables. In a large program, programmers like to use a name temporarily in a local function, and then reuse the same name elsewhere in a different scope. If this were not done (or permitted), the compiler would end up with a large number of names of variables to handle. This is dealt in the C language by prefixing the definition statement by `auto`, `register`, `extern` or `static`. All variables defined outside a C block are global variables by default. Hence, unless otherwise specified, all functions are global. Any variable defined inside a block is local to that block unless explicitly mentioned. These variables are generally, by default, `auto` in nature, even if not specified. They are allocated on the stack. On certain older platforms, stack was limited in runtime; hence care had to be

All variables defined outside a C block are global variables by default. Hence, unless otherwise specified, all functions are global. Any variable defined inside a block is local to that block unless explicitly mentioned.

If a variable is global and `static` keyword is applied to it, it continues to be global, but only in the file where the code appears. Hence, the variable (or function) will not be visible outside this file.

taken in such cases, especially when one writes a recursive function. The keyword `extern` makes the statement a declaration, and hence no memory allocation is done. The keyword `register` forces the compiler to consider that variable to be kept in the register, if possible. Hence it is not always guaranteed, but an effort will be made to place it in a register.

Now for the keyword `static`, which is at the heart of the scoping mechanism in C. If a variable is global and `static` keyword is applied to it, it continues to be global, but only in the file where the code appears. Hence, the variable (or function) will not be visible outside this file. If a C program spawns across files, then this static variable will be hidden and invisible to other files. If a variable is local and static is applied to it, then that variable would no more be *auto,* and will not be allocated on stack; instead it will be statically allocated in another memory area. Hence, the variable will retain the value on return to the function next time. This is how a 'locally global' variable can be created. Thus, static in one case hides visibility keeping the allocation strategy unchanged, and in the other case, visibility is unaltered, but allocation strategy is changed from automatic to a permanent memory location. By making global variables and functions static, and using local variables inside a C block, the namespace pool can be efficiently managed.

In C, arguments of functions are always '*pass-by-value*'. '*Pass-by-reference*' is simulated to a certain extent, using pointer mechanisms. However, one can never achieve the same effect in toto. In the past, only primitive types of data were allowed to be passed, and for any aggregate data, an address was passed to access the contents. Hence, implementing recursive functions with aggregate data was not trivial. Any higher abstraction, where both code and data need to be handled as objects, can only be implemented using pointers to both data and functions.

By making global variables and functions static, and using local variables inside a C block, the namespace pool can be efficiently managed.

These are the main features of the initial language proposal and implementation of C by K&R. Entire I/O was kept out of the language C (even today). In fact, this is the only language that

does not include I/O as a part of the language. Controversies apart, no program can be conceived to be complete without a proper I/O. C has outsourced that job to a set of libraries, called Standard Libraries. Hence, a proper C compiler cannot, in practice, work without a standard library. Now it is imperative that one implements the standard library in the runtime environment of the platform. Initially this was carried out on PDP-11, and later extended to other platforms and hardware architectures. This allows the core C compiler to be very simple and easy to implement in an environment where memory is small. Hence, possibly, the designers of C were forced to leave out the I/O, and many other desirable features from the language.

C is the only language that does not include I/O as a part of the language.

2.1 *Shortcomings of K&R C*

Initial versions of K&R C did not specify that the types and number of arguments to a function be tested. Also, the declarations of the type of arguments were not included in the definition statement of a function itself. This has been changed in the later versions of K&R C. Earlier, separate utilities such as *lint* were used on Unix platforms to do a strict type checking of all the functions and formal parameters.

C does not guarantee the order of evaluation in the arguments to a function. This is left completely to the implementation of the compiler. For example, a code segment such as, `i=10; printf("value of i = %d : %d",i++,i);` would not always print the same result across platforms (parameters could be evaluated from left to right or vice-versa). And there are controversies regarding the hierarchy of operator precedence. For example, non-intuitive operator precedence such as == binding more tightly than & and | in expressions such as `(x & 1 == 0)`, which would need to be written as `((x & 1) == 0)` to be evaluated properly.

C does not guarantee the order of evaluation in the arguments to a function. This is left completely to the implementation of the compiler.

Also, types such as `int` are either 32 bit or 16 bit, depending on the architecture where it is compiled (64 bit machines had not yet appeared on the horizon in the 70's and 80's). Though for small

No bounds checks are carried out on arrays during run time keeping an eye on the efficiency of code.

values this will not be a problem, it can create problems during run time. It is the programmers' responsibility to declare it as `long` or `short` as the case may be. No bounds checks are carried out on arrays during run time keeping an eye on the efficiency of code. However, this may be excused, as an array itself is merely a pointer manipulation and not a serious data structure (as implemented in C).

Due to all these shortcomings, several improvements happened to K&R C and by 1989, it supported the following:

- Functions could return void.
- Function could return higher data structures, structs or unions.
- Assignment operators for structs.
- Enumerated types.

3. C89/C90 and ANSI C

During these developments, Brian Kernighan and Dennis Ritchie were involved in the ANSI standardization process of C. The American National Standards Institute formed a committee X3J11, to establish a standard specification of C, which was overdue by then. A non-portable portion of Unix C library was handed off to the IEEE working group 1003, which formed the basis of the POSIX standard. In 1989, this C Standard was ratified as ANSI X3.159-1989 "Programming Language C". This version henceforth was termed ANSI C, and is also known as Standard C, or C89.

In 1990, the ANSI C standard (with formatting changes) was adopted by the ISO (International Standardization Organization) as ISO/IEC 9899:1990, which is sometimes called C90. Therefore, the terms 'C89' and 'C90' refer to the same programming language.

Brian Kernighan and Dennis Ritchie were involved in the ANSI standardization process of C which was overdue by 1989.

The goal of these committees was to make sure that the new superset of K&R C was backward compatible. They incorporated many of the useful unofficial features, which had made their way

into the language by this time. They included several additional features such as function prototypes (borrowed from C++), void pointers, support for international character sets and locales, and preprocessor enhancements. Although the syntax for parameter declarations was augmented to include the style used in C++, the K&R interface continued to be permitted. Many compilers currently implement C90, as the code written in Standard C, without any hardware specifics, is guaranteed to work across platforms. A number of changes happened to the standard library of C, which was specific to implementations.

Many compilers currently implement C90, as the code written in Standard C, without any hardware specifics, is guaranteed to work across platforms.

4. C99

During mid and late 1990's there was not much development in ANSI C, leading to stagnancy. There were a few minor changes made to locales, and only towards late 1999, some development was seen leading to the publication of ISO/IEC 9899:1999 for C language and that has been amended thrice till 2011.

C99 has several features. The salient among them are listed below. It supports:

- Inline functions
- Long long int
- A boolean type which takes value true or false
- A complex type to represent complex numbers
- Variable length arrays
- Improved IEEE 754 floating point operation
- Variadic macros (macros with variable arity)
- One line comment of the form //

Towards late 1999, some development was seen leading to the publication of ISO/IEC 9899:1999 for C language and that has been amended thrice till 2011.

Inline functions are functions, whose block of code is expanded and inserted in between the code, rather than generate some function code and insert a call to that function. This has implication in speed, whenever there is a function which needs to be called again and again. This was earlier achieved by macros with parameters, which suffer from lack of type checking, and argument checking, etc. No more does a definition/declaration statement need to be in the beginning of a file, or block of code. Now,

Use of C99 in scientific computing now becomes a little easier with complex data types, and by providing several library calls to support the same.

it is allowed to be intermixed with code. Newer data types such as long long int, which supports higher precision, and boolean data type, which takes exclusively only true or false values, are allowed now. Scientific computing now becomes a little easier with complex data types, and by providing several library calls to support the same.

Variable length arrays are implemented in C99. The following code shows how a variable array is created and initialized in a function, and how at the end, the entire array is returned to the calling function.

```
float  my_function( int n )
{
   float my_array[n];              // defining variable array
   for (register int i = 0; i < n; i++)
     my_array[i] = read_val();    // populating the array members
   return my_function(my_array, n);
}
```

Support for variadic macros is a new feature and is very powerful. Now there can be a variable number of arguments to macros as well, which enhances the reusability of conventional C code.

Consider the following example.

```
#define eprintf(...) fprintf (stderr, __VA_ARGS__)
```

A macro call such as, `eprintf("%s : %d",name, age);` would expand inline to

```
fprintf (stderr,"%s:%d",name,age);
```

There can be a variable number of arguments to macros as well, which enhances the reusability of conventional C code.

Note that "__VA_ARGS__" corresponds to a varying number of formal parameters, to be expanded appropriately, when actual parameters are supplied in a call.

5. C11

The current version is ISO/IEC 9899:2011 and is known as C11, and it replaces C89/C90. The main contributions are in the form of acceptance of features that were found in various compilers across the platforms; predominantly, acceptance of better memory models to support multiple threads of execution. C11 also accepts more features as optional so as to maintain acceptance of the core standards. C11 has added 5 more keywords to the existing 32 keywords of C89/C90. The main differences from C89/C90 that have been put forth in C11 are listed below. Most of these are enhancements to and not fundamental changes to C89/C90.

- Strict type enforcement – no more is the default value considered as int
- Alignment specification
- Type generic expressions
- Multiple threading support
- Unicode support
- Bounds-checking interfaces
- Anonymous structures and unions
- Static assertions
- Macros for the construction of complex values

The changes to C99 were predominantly improvements and retention of backward compatibility to C89/C90. Enhancements were made to the standard library, with stricter type checking, wherever applicable. Parts of the C99 standard have made their way to the current version of the C and C++11.

The changes to C99 were predominantly improvements and retention of backward compatibility to C89/C90. Enhancements were made to the standard library, with stricter type checking, wherever applicable.

6. Conclusions

C has several shortcomings (only a few have been discussed in this article) and at several points of time, standardization committees could not take a decision to change the language features in order to address these. Thorough discussion of the reasons why changes were not unanimously accepted is beyond the scope of this article. For instance, strict type checking at every level, many

higher-level data structures, and features very special to embedded environments could not be made a part of the language. Hence C has branched out to many related languages, such as, C++, C#, Embedded C, etc., each catering to specific requirements. However the main branch of C, which we have considered has still traversed a long distance, catering to a large population of programmers, especially in the area of systems programming. What else is seen on the horizon? A short speculative list is given here.

- Green code with low power consumption is becoming the order of the day. Are language facilities necessary to enable this, or can it be left to the compiler?
- Support for large-scale multicore programming. Are transaction memories going mainstream or some thing else is more promising?
- Support for handling memory security vulnerabilities in low-level programming. Can the pain in programming be reduced, and can ad hoc techniques be eliminated?

Address for Correspondence
K Bhaskar
Department of Electrical Engineering
Indian Institute of Science
Bangalore 560 012, India.
Email:
bhaskar@ee.iisc.ernet.in

Suggested Reading

[1] **B W Kernighan and D M Ritchie, *The C Programming Language*, Prentice Hall, 1978 (Ist edition), 1988 (2nd edition).**
[2] **P J Plaugher, The Standard C Library, Prentice Hall, 1991.**
[3] **http://en.wikipedia.org/wiki/C_(programming_language)**
[4] **http://en.wikipedia.org/wiki/The_C_Programming_Language**
[5] **http://en.wikipedia.org/wiki/C99**
[6] **http://en.wikipedia.org/wiki/C11_(C_standard_revision)**