Appendix V

C99/C11 Features

C99 Features

INTRODUCTION

C, as developed and standardized by ANSI and ISO, is a powerful, flexible, portable, and elegant language. Due to its suitability for both systems and applications programming, it has become an industry-standard, general-purpose language to-day.

The standardization committee working on C language has been trying to examine each element of the language critically and see any change or enhancement is necessary in order to continue to maintain its lead over other competing languages. The committee also interacted with many user groups and elicited suggestions on improvements that are required from the point-of-view of users. The result was the new version of C, called C99.

The C99 standard incorporates enhancements and new features that are desirable for any modern computer language. Although it has borrowed some features from C++ (a progeny of C) and modified a few constructs, it retains almost all the features of ANSI C and thus continues to be a true C language.

In this appendix, we will highlight the important changes and new features added to C by the 1999 standard.

KEYWORDS

ANSI C has defined 32 keywords. C99 has added five more keywords. They are as follows:

_Bool
_Complex
_Imaginary
inline
restrict

Addition of these keywords is perhaps the most significant feature of C99. The use of these keywords are highlighted later in this appendix.

COMMENT

C99 adds what is known as the single-line comment, a feature borrowed from C++. Single-line comments begin with // (two back slashes) and end at the end of the line. Examples:

```
                                        // A comment line
if (x > y)                              // Testing
        printf(.....);                  // Printing
int m;                                  // Declaration
```

Single-line comments are useful when brief, line-by-line comments are needed.

DATA TYPES

C defines five basic data types, namely, **char, int, float, double,** and **void.** C99 adds three new built-in data types. They are as follows:

_Bool
_Complex
_Imaginary

C99 also allows **long** to modify **long** thus creating two more modified data types, namely, **long long int** and **unsigned long long int.**

_Bool Type

_Bool is an integer type which can held the values 1 and 0. Example:

```
_Bool x, y;
x = 1;
y = 0;
```

We know that relational and logical expressions return 0 for false and 1 for true. These values can be stored in _Bool type variables. For example,

```
_Bool b = m > n;
```

The variable b is assigned 1 if m is greater than n, otherwise 0.

_Complex and _Imaginary Types

C99 adds two keywords **_Complex** and **_Imaginary** to provide support for complex arithmetic that is necessary for numerical programming. The following complex types are supported:

float_Complex	float_Imaginary
double_Complex	double_Imaginary
long double_Complex	long double_Imaginary

The long long Types

The **long long int** has range of at least $-(2^{63}-1)$ to $2^{63}-1$. Similarly, the **unsigned long long int** has a range of 0 to $2^{64}-1$.

DECLARATION OF VARIABLES

In C, we know that all the variables must be declared at the beginning of a block or function before any executable statements. However, C99 allow us to declare a variable at any point, just before its use. For example, the following code is legal in C99.

```
main()
{
        int m;
        m = 100;
        . . . . . .
        . . . . . .
        int n;                    /* Legal in C99*/
        n = 200;
        . . . .
}
```

C99 extends this concept to the declaration of control variables in **for** loops. That is, C99 permits declaration of one or more variables within the initialization part of the loop. For example, the following code is legal.

```
main()
{
        . . . . . .
        . . . . . .
        for (int i = 0; i<5; i + +)
        {
                . . . . . .
                . . . . . .
        }
        . . . . . .
        . . . . . .
}
```

A variable declared inside a **for** loop is local to that loop only. The value of the variable is lost, once the loop ends. (This concept is again borrowed from C++.)

I/O FORMATS

In order to handle the new data types with **long long** specification, C99 adds a new format modifier **ll** to both **scanf( )** and **printf( )** format specifications. Examples: **%lld, %llu, %lli, %llo, and %llx.**

Similarly, C99 adds **hh** modifier to d, i, o, u, and x specifications when handling **char** type values.

HANDLING OF ARRAYS

C99 introduces some features that enhance the implementation of arrays.

Variable-Length Arrays

In ANSI C, we must declare array dimensions using integer constants and therefore the size of an array is fixed at compile time. C99 permits declaration of array dimensions using integer variables or any valid integer expressions. The values of these variables can be specified just before they are used. Such arrays are called *variable-length arrays.*

Example:

```
main()
{
        int m, n;
        scanf("%d %d", &m, &n);
        float matrix [ m ] [ n ];      /* variable-length array */
        . . . . . .
        . . . . . .
}
```

We can specify the values of m and n at run time interactively thus creating the matrix with different size each time the program is run.

Type Specification in Array Declaration

When we pass arrays as function arguments, we can qualify the dimension parameters with the keyword **static**. For example:

```
void array (int x [ static 20 ])
{
        . . . . . .
        . . . . . .
}
```

The qualifier **static** guarantees that the array **x** contains at least the specified number of elements.

Flexible Arrays in Structures

When designing structures, C99 permits declaration of an array without specifying any size as the last member. This is referred to as a **flexible array member**. Example:

```
struct find
{
        float x;
        int number;
        float list [  ];          /* flexible array */
};
```

FUNCTIONS IMPLEMENTATION

C99 has introduced some changes in the implementation of functions. They include the following:

- Removal of "default to **int**" rule
- Removal of "implicit function declaration"
- Restrictions on **return** statement
- Making functions **inline**

Default to int Rule

In ANSI C, when the return type of a function is not specified, the return type is assumed to be **int**. For example:

```
prod(int a, int b)          /* return type is int by default */
{
        return (a*b);
}
```

is a valid definition. The return type is assumed to be **int** by default. The implicit **int** rule is not valid in C99. It requires an explicit mention of return type, even if the function returns an integer value. The above definition must be written as:

```
int prod(int a, int b)      /* explicit type specification */
{
        return (a*b);
}
```

Another place where we use implicit **int** rule is when we declare function parameters using qualifiers. For example, function definitions such as

```
fun1( const a)              /* a is int by default */
{
        . . . . . .
}
```

and

```
fun2 (register x, register y)       /* x and y are int */
{
        . . . . . .
}
```

are not acceptable in C99. The parameters a, x and y must be explicitly declared as **int**, like:

```
const int a
const register x
```

Explicit Function Decalaration

Although prior explicit declaration of function is not technically required in ANSI C, it is required in C99 (like in C++).

Restrictions on return Statement

In ANSI C, a non-void type function can include a **return** statement without including a value. For example, the following code is valid in ANSI C.

```
float value (float x, float y)
{
        . . . . . .
        . . . . . .
  return;                   /* no value included */
}
```


But, in C99, if a function is specified as returning a value, its return statement must have a return value specified with it. Therefore, the above definition is not valid in C99. The **return** statement for the above function may take one of the following forms:

```
return(p);      /* p contains float value */
return(p);
return o.o;     /* when no value to be returned*/
```

Making Functions inline

The new keyword **inline** is used to optimize the function calls when a program is executed. The **inline** specifier is used in function definition as follows:

```
inline mul (int x, int y)
{
        return (x*y);
}
```

Such functions are called *inline functions*. When an inline function is invoked, the function's code is expanded inline, rather than called. This eliminates a significant amount of overhead that is required by the calling and returning mechanisms thus reducing the execution time considerably. However, the expansion "inline" may increase the size of the object code of the program when the function is invoked many times. Due to this, only small functions are made inline.

RESTRICTED POINTERS

The new keyword **restrict** has been introduced by C99 as a type qualifier that is applied only to pointers. A pointer qualified with **restrict** is referred to as a *restricted pointer*. Restricted pointers are declared as follows.

```
int *restrict p1;
void *restrict p2;
```

A pointer declared "restricted' is the only means of accessing the object it points to. (However, another pointer derived from the restricted pointer can also be used to access the object.)

Pointers with **restrict** specifier are mainly used as function parameters. They are also used as pointers that point to memory created by **malloc ( )** function.

C99 has added this feature to the prototype of many library functions, both existing and new. For details, you must refer to the functions defined in the C standard library.

COMPILER LIMITATIONS

All language compilers have limitations in terms of handling some features such as the length of significant characters, number of arguments in functions, etc. C99 has enhanced many of such limitations. They are listed below:

- Significant characters in identifiers: increased from 6 to 31
- Levels of nesting of blocks : Increased from 15 to 127
- Arguments in a function : Increased from 31 to 127
- Members in a structure : Increased from 127 to 1023

C11 Features

INTRODUCTION

C11 is the current C standard that replaces the previous standard i.e. C99. C11 mainly aims at standardizing features that are common to most of the present-day compilers. To overcome one of the limitations of C99 where there was a delay in compilers conforming to the C99 standard, C11 makes several of its features optional. This makes it easier for the compilers to conform to the C11 standard.

LANGUAGE AND LIBRARY SPECIFICATION CHANGES

Let's now take a look at some of the key language and library specification changes that C11 brings to the C99 standard:

_Noreturn Function Specifier

It is used in the function declaration statement to indicate that the function does not return either by a return statement or by reaching the end of the function body. The specifier is provided in the stdnoreturn.h header file.

Some of the standard library functions that are declared using noreturn specifier include abort(), exit(), longjmp(), etc.

_Generic Keyword

It provides a method for choosing one of the given expressions at the time of compilation.

Syntax: `_Generic (controlling-expression, association-list)`

Here, controlling-expression is any expression, except the comma operator, and association-list is a comma-separated list of associations.

Thread local Specifier

The variables declared with _Thread_ local specifier are given thread-specific storage duration. That is, the variables are allocated when the thread begins and deallocated when the thread ends. Further, conforming to the thread-specific behavior, each thread has its own instance of the variable. The new header file <threads.h> contains _Thread_local specifier.

Enhanced Unicode Support

C11 supports certain enhancements related to Unicode support. For instance,

- New header file uchar.h comprising of UTF-16 and UTF-32 related character utilities
- u and U string literal prefixes
- u8 prefix for UTF-8 encoded literals

Anonymous Structures and Unions

C11 supports implementation of anonymous structures and unions, which are specifically useful in situations where unions and structures are nested. The following example code illustrates this:

```
struct S
{
 char c;
 union
 {
  int a; float b;
 };
};
```

Note that in the above code union is anonymous, as it has not been named.

ADDITION OF quick_exit FUNCTION

It is a new function for quick program termination. It terminates the program without cleaning the system resources.

REPLACEMENT OF gets WITH gets_s

The gets() function has been completely removed with C11 as it does not perform bounds checking resulting in buffer overflow situations. The new gets_s() function overcomes this limitation by reading at most characters from stdin. It keeps reading until a new line or end-of-file is reached and returns only a part of the read characters to the input buffer.

ALIGNMENT SPECIFICATION

C11 provides four new macros for alignment of objects. These are:

_alignas
_alignof
_alignas_is_defined
_alignof_is_defined.

These macros are provided in the new stdalign.h header.

Note *For a complete list of C11 standard features, you may refer the ISO/IEC 9899:2011 specification.*