Learning Correct Programming

C

Shiv S Dayal

Contents

Tables

Figures

Preface

Please let me introduce myself. As you know my name is Shiv Shankar Dayal if you have taken care of reading author's name on the first page which I expect you have unless you were too busy and having your attention anywhere else. I am a programmer by choice and was so by profession as well till recent past when I decided to quit job because of personal problems. So I decided to write this book to pass my time as I am at home full time. There are many books on C, however, for a person like me who has no source of income is forced to rely on open-source books. I agree that there are tons of excellent free (free as in freedom, not free beer) software, but few free books. Again, there is a problem that software can be created and copied and maintained easily but such is not the case with books if you publish it. Therefore, I have decided to keep under GNU FDL and distribute only soft copies. If some publisher publishes it even then soft copy will remian free.

There is another reason for writing a book on C. First, I think I know it well. Just to brag I have been programming for 11 years, however, my knowledge is very limited and inaccurate when it comes to programming. On the contrary I write programs very slow. They are never commented and complex and difficult to maintain. Second, C is kind of greatest common denominator of many major software. Almost all Unix-like systems and also Windows have been developed in C. Programming languages like C++, Java, Perl, Python, Ruby, Lua etc have all there roots somewhere in C. Again, I am not sure facts please cross-check about names. I am just guessing. All the languages have C bindings. Backward compatibility with C is a very important reason for the success of C++. However, that is again a matter of entirely different discussion. I am just saying what I think.

This is a book on C as you already know by its name. However, this book will delve into topics not ventured together in many books. The most authoritative source on this subject is ofcourse K&R's "The C Programming Language". This book is not a replacement of the above book. It is a complement and supplement for the K&R's (henceforth I will refer it as The C Book) book. However, C has undergone revision and latest standard being C99 which is not part of The C Book. Therefore, until a revision is made in the contents it will not reflect the changes in the standard. Several compilers are already supporting this standard. `gcc` already supports most of standard which is used in preparing this book.

I believe in practical programs so sometimes code examples may span several files and pages. Please skip them if you are not interested. Sometimes toy examples are not simple.

My thoughts jump from one topic to another and I capture them immediately so the book may seem rather wayward. I have tried my best to organize the topics but still the ordering may be not so good. Please use index. After all, that is what indices are meant for! :-)

Let us come back to the title. As you have seen it is "Learn Correct Programming". There are various ways to program. However, programming paradigms associated with language does not enforce correctness in the sense of correct way. Two programming paradigms which are generic try to do it. One of them is "Literate Programming" and "Pair Programming". Literate Programming is baby of greatest computer scientist alive in my humble opinion

Donald Erwin Knuth. I am a big fan of Literate Programming, however, I can not enforce documentation or learning TEX on a normal programmer. Pair programming is not always feasible and not particularly so in enterprise environment where the sole aim is to make money. However, if you can please use both of them as they will help you learn programming immensely. Programming requires discipline. Discipline in choosing correct language for the task, correct selection of algorithm, correct implementation and never to mention documentation and testing.
As you may notice there is something called Part I C in this book. There is a reason that once this part will be finished I will put more parts probably C++ or some library uses like pthreads.

Tools

You must choose your tools carefully as it may determine the direction your career is going in. If you want to Windows programmer choose MS Visual C++ Express Edition if that suits you else get cygwin or MSYS and MINGW. However, you can also choose GNU/Linux (free comes in many variations), MacOS X (not free and supposed to run only on Mac hardware but internally is Unix), OpenSolaris but at this moment it is lagging as only 2009 release is there.
As compiler I use `gcc`, `gdb` as debugger, `emacs` along with viper-mode, cedet, Emacs class browser as my editor and ide. Emacs is fantastic because I can compile and debug right there in editor. Also, viewing manuals in Emacs is pretty easy for quick reference. But be warned Emacs has a steep learning curve and may be daunting who has only used Windows. It will feel better if someone entirely new to computers uses it. Also, Emacs has an in build shell, browser, and many other things which I do not even know. Best part is that if you want some functionality then it can be extended using Emacs-Lisp which is another programming language of its own type. I also use `valgrind` whenever I suspect a suble memory problem or thread problem. All these tools are licensed and cost nothing.

Acknowledgements

The time when I am writing this book is very tough on my family and even then they have supported me. I can not show enough gratitude to my wife, parents and siblings. They have stood by me and my decisions to quit job which is a big change in our lives. I can not state how much the life of my parents have effected me.
I thank my teachers Yogendra Yadav, Hriday Narayan Singh, Satyanand Satyarthi, Gopal, Shilesh Kumar, Praveen Kumar, Prod. T K Basu, Prof. S Sen, and Prof. S Banerjee for imparting their knowledge.
At professional level I learnt a lot from two persons. First is Anurag Johri and second is Vibhav Saluja. They taught me several technical things and a lot more about life.

On the programming side there are far too many people I mena whole Free Software Community, however, I will take some names. Donald Erwin Knuth, creator of TEX, which has been used in preparation of this book, who is also author of many books like ``The Art of Computer Programming'' is first among them. Richard M Stallman is second but his contributions are more to life than software, however, that does not mean that he has done any less contribution on software side. He contributed to `gcc`, `gdb`, `emacs`, TEX`Info`, `ELisp` and may more things. His biggest contribution is free software movement. I also want to thank Hans Hagen who helped me with ConTEXt which I used to finally process the contents of this book. I also used `vim` syntax highlighting module from Aditya Mahajan to highlight the source code given in this book. I am most grateful to all these nice people who believe in sharing the knowledge at source code level and not at binary level.
English is not my native language and even though I have run spell check over buffers there are chances that grammatical and factual errrors are there. The reason is that it has not been proof-read and edited. All I can guarantee is that source code given runs on my machine. Please mail me your suggestions, errors which you have found to me at . I will try to give reply to each of your mail.

Shiv Shankar Dayal

Part I

C

1.1 Introduction

All programs in this book were created using Emacs, gcc, gdb, and Valgrind running on Ubuntu 10.04. However, these should run on any machine having a C99 compliant compiler. Not much of history I am going to tell. Dennis Ritchie designed and architected C programming language in late 1960s and early 1970s. It spread in the programming world along with Unix. Unix was developed by Ken Thompson. Both Ken and Dennis were colleagues at AT&T Bell labs. C is a programming language which is simple, small yet powerful. I will not go in the discussion of low, high or middle level language as this is going to be a very informal book. When C became popular it was standardized by ANSI and latest revision is C99. In this book we will follow C99 standard and see what all is there. C is a matured programming language and has weatherd the test of time well. I love programming in C and so would you (probably :-)) because it is very simple compared to C++. Yes, I agree that object-oriented programming features are not there and there are certain problems in writing large scale code with C but again many very large systems have been written using C successfully. Theoretically, C++ is superior featurewise but C++ is complex as well and in order to properly learn C++ you must learn C first. Apart from C++ other languages like Java, Perl, Python, Ruby, Lua and many others have used C at some point of time. Essentially, you can say that C is the common denominator about which other systems have been developed. Linux kernel has been written using C. C is the standard programming language for any application of GNU operating system. I am not a proponent of C or anything I am just trying to make the reader, in this case you, aware of C capabilities.
There are many programming paradigms with their own strengths. Some of them are structured programming, object-oriented programming, functional programming, and generic programming. C++ is a multi-paradigm language and C supports one which is structured programming. Obviously C is so simple that it can not support multiple paradigms. However, you can twist it a bit to accomodate other paradigms but that is not recommended.
If you are new to programming and just picked up this book in some shop because name appearred to be catchy then please read following carefully. First question is “What is a programming language?” The answer is “A Programming language is a language in which you write a program.” Next question could be “What is a program?” Then answer would be A program is an organized sequence of instructions to computer (Read processor. CPU or GPU or anything which can be programmed. However, in our case we will limit our discussins to CPU or central processing unit of computer and nothing else.) By organized sequence I mean that they must at least satisfy the syntatical requirements of programming

language. It does not matter whether the program is doing anything useful. This leads to another question and that is that what is an instruction. This leads us in to some sort of physics so I will touch it briefly in next sequence.
A computer does not understand 0 or 1 or any other number let alone characters. All a computer understands in voltage levels. For the sake of discussion let us take 0V and 1.5V. We as human interpret 0V as 0 and 1.5V as 1 or vice-versa. However, this interpretation must be uniform across all computers of same type. Then we use binary bumber system to do basic arithmetic and convert them to digital and other number systems. We also use hexadecimal number system to represent bytes as hexadecimal bumber is very compact and symbols for this number system are from 0 to 9 and A to F or a to f. Please do not ask me why base-32 was not used. Honestly, I do not know the answer. I guess probably because 32 is 2^5 and 5 is not 2^n where n is an integer. Next number would have been $n = 8$. This would mean having 256 symbols for denoting the number system which would be impractical for humans. Now coming back original question certain operations like addition, subtraction and other basic operations are implemented in hardware itself for certain width of numbers. For example, a typical 32-bit modern computer can add 2 32-bit integers with carry bit. These instructions have equivalent numbers assigned to them. So for example to computer you say "Add 5 and7". This may translate to 13 5 7. 13 being instruction for addition 5 and 7 have their usual meaning. So it will be then converted to binary that is 1101 101 111. So accordingly volatage levels will change. For other characters like of English language or pucntuation characters or computer specific characters or mathematical or Greek symbols we have ASCII, or UTF-8, and UTF-16 tables. ASCII table is made up of 128 symbols. UTF-8 of 256 and UTF-16 of 65,536 symbols respo=ectively. UTF-16 can accomodate all characters of all languages and everything at the moment.
So far so good. Now it is time to learn about C. A programming language is somewhat like a natural language for example, English or Chinese or Hindi or any other language. Now as you know each language has a set of alphabets, a dictionary and grammar. There are certain rules in natural languages. Similalry, C has a set of alphabets, a dictionary of reserved words and a BNF-grammar. Now since I have introduced the word BNF let me tell you about it a bit. BNF stands for Backus-Naur form. Probably it is wrong place to say about it but I do not like to hold information back. BNF grammar is actually a metasyntax which can be used for describing context-free grammars. For example, in C the compiler or the oparating system or computer will not know about the context of the program. Even then it works. The main use of BNF grammar is to describe lexical and syntatic rules of a programming language and a tool like lex and yacc or flex and bison can used this grammar to help you develop compiler for that language for which the grammar was writte. I will stop BNF discussion here as it is a subject of other books.

1.1.1 Alphabets, and Dictionary of C

Given below are the characters you can use in C programming language:

```
a-z A-Z 0-9 ~ ! # % ^ & * ( ) - = [ ] \ ; ' , . / _ + { } | : " < > ?
```

Given below is the dictionary pf C language as specified in C99 standard. These words are also kmnown as reserved words and can not be used other than what they are meant for. This **1.1** consists only of standard keywords so watch out for your compiler specific keywords.

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

Table 1.1 C Keywords

1.1.2 Tools of Trade

You will need a compiler, debugger, and a text editor. I use GCC as compiler, GDB as debugger and Emacs as text editor. All these three are free (free as in freedom not free beer) programs licensed under GNU GPL. However, you can opt for Microsoft Visual Studio on Windows, XCode on Mac OSX which internally uses gcc, Kdevelop, Anjuta, Codelite or any other IDE on GNU/Linux. Some other IDEs are also available like Eclipse and Netbeans. I use emacs with cedet and ecb-bowser which gives me sufficinet power and ease of use. What you choose is your wish however if you use something like Turbo C++ then outputs may not match.
Valgrind is another piece of software which is used to find memory leaks, heap corruption, cache corruption but at the moment it is only for GNU/Linux. DBX is another nice debugger for Solaris. You can also use mingw and msys on Windows or cygwin if you do not want to do a full install of GNU/Linux and still want to use same set of tools.

For compilation there are various ways. I will use famous GNU Makefiles. However, you learn something like autotools, cmake, scons, premake, bjam or some other good build system as per your choice.

1.1.3 First Program

This is the holy grail program of most of the books. "Hello, world" program is given below and all it does is that it prints "Hello, world!" and does nothing else.

```
1  /*
2   * Author: Shiv Shankar Dayal
3   * Date: July 3rd, 2010
4   * Description: It prints the string "Hello, world!"
5   */
6  #include <stdio.h>
7
8  int main()
9  {
10   printf("Hello, world!\n");
11   return 0;
12 }
```

Hellow World Program

To compile this program just type following at your command prompt if you are using one:

```
gcc hello.c
```

gcc is the name of compiler and hello.c is the name of file in which you saved the code shown above as. Assuming everything went fine you will have a file called **a.out** if you are using a Unix-like system or **a.exe** if using cygwin. For IDEs it will typically depend upon the project name which you specified at the time of project creation. Also, gcc must be in the **PATH** environment variable which is typically **/usr/bin** on standard installation. Look up your system/compiler manual. Once you have **a.out** file you can execute or in other words run it. Given below is one such screeshot.

As you can see first part which you notice is a block starting with /* and */. Such a block is called comment block and everything inside is ignored by the compiler. For example, compiler or the program or operating system or processor will have no Author or Date or Description with them. At the compilation time itself it is filtered. However, be careful of such nested comments for example, `/* something here*/ something else here */`. In such cases the program may not behave as expected.

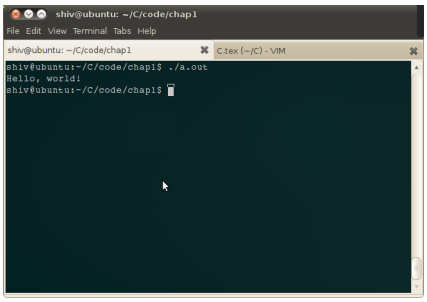

Figure 1.1 Output of hello.c

Next line is `#include <stdio.h>`. This is actually consists of two parts. First part is `#include`. It is a preprocessor directive. By using such directories you can include any kind of file to your source code. There is another syntax associated which we will see later and why headers are required that also we will see. Second part is `<stdio.h>`. Angular brackets are syntatic sugar and the it can be written also like `"stdio.h"`, however, not always. stdio.h has got a single character extension 'h', which means header file. stdio can be interpreted as standard input/output. However, note that stdio.h is not part of C99 standard. This header and associated funciotnality is provided by compiler vendors.
Next line is `int main()`. It declares a special funciton `main`. Every C executable will typically have one such function unless you are willing to program in assembly. main is the entry point of the program. The word `int` preceeding it is the return type of this funciton. `int` means integer. So, probably now you can guess that `main` is supposed to return an integer to its caller. In this case the called is the shell from which `./a.out` is being executed i.e. bash shell on my system. In your cases the system or entity calling `main` may be different. Next you see two curly braces. Every function in C has at least two matching braces. These braces define the function scope. What I mean is whatever code is inside these two braces will be there will be part of logic of that function. At this point I should tell you that in C one function can call another function which is the very basic of structured programming on which C is based. In structure programming you break down the problem in sub-tasks and write one function for each of these tasks. If the sub-tasks are two big then you further break it down until it becomes manageable.
As you see line number 10 is `printf("Hello, world!");`. This printf is also a function which takes a string as argument and can also take a variable number of argumnets as we will soon see. This function's prototype is defined in `stdio.h` and the fumnctionality is in `libc.so` on Unix systems which typically resides in either `/libc` or `/usr/lib`. Please check your system for its location. Prototype defines the way function should appear and definition of that function is entirely different entity. The prototype and function signature at the point of definition must match. Very soon we will see them when we discuss formal and actual parameters.
Line number 11 is `return 0;`. `return` is a keyword of C as you might have noticed the dictionary of C or rather keywords of C. When this keyword is encountered it will return

whatever value is there just by the side of it which happens to be 0 in this case. Whenever a `return` statement is encountered the execution of that function terminates.
You might have noticed `;` at two places. Every function call must terminate with `;` and so as an statement like `return`. There may be more possibilities of using `;` as we encounter them I will tell you about them.
Now we have seen the holy grail so it is time for something holiest. I am talking about attributes of a program. What are the attributes of a program which you as a programmer must look for?

1.1.4 Attributes of a Program

All the programs are meants for different purposes. They may be small, large or mid-sized. However, there are certain criterium which apply to all of them which we are going to discuss here. Given below are some of them. There may be more but depending upon situation but never less.

1. **Correctness.** Every program has some purpose and it must fullfil that purpose without giving wrong solution. This is a mandatory requirement or attribute of a program. Without correctness a program has no meaning.

2. **Efficiency.** By efficiency I mean the program should consume less resources at the same time it gives output in least amount of time. Efficiency is an illusive or rather a trade-off goal. If you want less time then you consume more CPU power or RAM or vice-v versa.

3. **Robustness.** By robustness I mean the ability to handle errors and keep working in a normal way. If possible try to process the same input again and again for a configurable amount of time. The program must not crash or stop working in case of wrong input.

4. **Generic.** The program should be generic. However, this is typically not a requirement as most of the time you know what sort of data program is going to get as input. However, as a library programmer or system programmer your program must be valiod for a wide range rather type of inputs.

5. **Security.** The program should be secure. It does not matter whether the program is running as standalone application or interacting with different processes on the same machine or over a network. Any unauthorized entity must not be allowed to do things like stack overflow, integer overflow etc.

6. **Maintainability.** Typically the period for which a program is maintained far exceeds the period of development. So the program must be maintainable. Also, by maintainable I mean easily maintainable not someone has to take pain to maintain it.

7. **Extensibility.** The program should be extensible. It should be written in such a way that modification to program does not effect it adversely.

8. **Comprehensibility.** The source code of the program should be easy to comprehend. Remember easy-to-read and easy-to-understand are two different things and easy-to-comprehend combines both. This is a precondition for maintainability.

9. **Documentation.** Source code must be documented well. Again this is a precondition for maintainability and comprehensibility.

10. **Portability.** Again, it is an elusive goal and depends on the requirement of the program you are writing. One of the easiest ways to achieve is to stick to the standards of the problem domain and programming language in use.

1.1.5 `printf` and `scanf`

I will tell you about two input and output functions which you will be using a lot in the beginning. Therefore, it is better to know them straightaway. Frist function is and output function which you have already seen. Remember `printf`. Similarly other function is for input having name `scanf`. So first we will take a look at `printf`. By no means this topic is going to cover all the details but only the basics. We will revisit this topic in input/output chapter. It has following prototype in your known header `stdio.h`:

```
1 int printf(const char *format, ...);
```

Figure 1.2 Declaration of `printf`

The string `const char *format` is what we have seen in out program. The more interesting one is triple dot. This implies that function can take variable number of arguments. I will give you one such example `printf("%d %d %f\n", 3, 6, 7.5);`. For the sake of testing just change hello.c with this new one compile and see the put put. You will see 3 6 7.50000 (How 7.5 became 7.50000? I will tell you this when we discuss floating-point numbers) on your screen probably. We can similary send one or more arguments to `printf`. You can also see that `printf` is returning and integer as we discussed for main. The value of integer will reflect that how many characters are printed. `printf` prints everything on STDOUT which is nothing but the display or monitor or screen. However, if `printf` fails it returns a negative integer.

1.1.5.1 Format Argument of `printf`

The format is a character string. It is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the STDOUT; and format specifiers, each

resulting in fetching zero or more subsequent arguments. Each format specifier is introduced by the character %, and ends with a format specifier.
In between there may be zero or more flags, an optional minimum field width, an optional precision and an optional length modifier.
The arguments must correspond properly (after type promotion) with the format specifier. By default, the arguments are used in the order given, where each `*' and each format specifier asks for the next argument (and it is an error if insufficiently many arguments are given).
For mor information just type `man 3 printf` on your unix system and you should see even more information. Do not worry we will cover them as we need them.

1.1.5.2 scanf

`scanf` is similar to `printf` but opposite in nature. Let us see its declaration.

```
1 int scanf(const char *format, ...);
```

Figure 1.3 Declaration of `scanf`

The usage are slightly different as we will see soon. This concludes our discussion for first chapter. In the next chapter we will talk about basic building blocks of a program like data types, operators, expressions and control flows.

1.2 Following Specification

At this point of time I would like to look back and refer to ISO/IEC 9899:TC2 which is the C99 specification with me. Since my thoughts are random in nature I will try to follow the organization of content given in the specification. However, note that I will not include all the contents of specification and will exclude some of very obvious and trivial sections/subsections.

1.2.1 Terms, Definitions, and Symbols

1. `access.` There are two parts of any program. Data and instruction. Programs are stored in file on some non-volatile storage for example, hard disk drive, cd, dvd, tape drive. When they are executed from non-volatile storage they are transferred to some volatile storage typically RAM (Random Access Memory) of the computer. When a program is executed it becomes a living entity capable of doing something and sometimes also referred as process. So when the contents of RAM (henceforth referred as memory) is either read or written (it does not matter whether the value is same or new) to then it is defined as access. Here point to be noted is that the expressions which will not be evaluated do not access objects.

2. `alignment.`Say your program requires x bytes of memory then it will not be always given x bytes but something more. Say an object requires y bytes then it will be always greater or equal to y bytes. This is required so that objects are always located on storage boundaries that are particular multiples of byte address. The reason for this alignment lies in the efficiency of the operating system as a whole. As we know that on 32-bit systems data bus is 32 bits similarly on 64-bit systems it is 64 bits. This means in one fetch cycle (read up on this on some microprocessor or computer architecture book) only 32-bits can be fetched. 32-bits means 4 bytes. Oops! I did not tell you about bits and bytes conversion. Not even nibble. However, since 4 bytes can be fetched in one cycle compiler tries to optimize the data in group of 4 bytes. Given below are some examples.

```
1  /*
2   * Author: Shiv Shankar Dayal
3   * Date: July 4th, 2010
4   * Description: Demonstration of structure padding and memory alignment.
5   */
```

```
6   #include <stdio.h>
7
8   typedef struct
9   {
10    char a;
11    int b;
12  }A;
13
14  typedef struct
15  {
16    char a;
17    int b;
18    char c;
19    char d;
20    char e;
21    int f;
22  }B;
23
24  typedef struct
25  {
26    char x;
27    char y;
28    int z;
29  }C;
30
31  typedef struct
32  {
33    char x;
34    int z;
35    char y;
36  }D;
37
38  int main()
39  {
40    A a;
41    B b;
42    C c;
43    D d;
44
45    printf("Size of structure %c is %d\n",'A',sizeof(a));
46    printf("Size of structure %c is %d\n",'B',sizeof(b));
47    printf("Size of structure %c is %d\n",'C',sizeof(c));
```

```
48      printf("Size of structure %c is %d\n",'D',sizeof(d));
49
50      return 0;
51    }
```

Memory Alignment Program

The output is shown below:

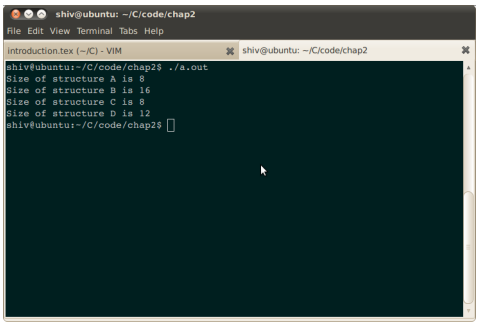

Figure 2.1 Output of align.c

I am always showing screenshots of program's output but now onwards I will not show them as they just consume space.

3. `argument` Actually you have seen this when I discussed the `printf` and `scanf` functions in Chapter 1. Sometimes they are also called actual parameters but as you can see in specification ISO/IEC 9899:TC2 Section 3.3 this term is being deprecated. A function can have zero or more actual arguments and if they are more than one then each of them will be separated by a comma. These also apply to macros that is preprocessor directives when invoked like a function.

4. `implementation-defined-behavior` When specification does not specify how a particular element of language should be implemented then programs use their logic to implement these things and sometimes it depends on hardware as well. Behavior of such elements is called implementation-defined-behavior.

5. `locale-specific-behavior` The implementation has a very generic description for it. In software locale typically confines its meaning to language. However, if you know more please e-mail me about it.

6. `undefined-behavior` Specification is sometimes not sure about behavior of program as it may be out of scope of specification. Specification gives an example of integer overflow

because integer overflow very much depends on word-width of integer which again is dependent on hardware and hence it is very difficult to capture in a specification.

7. `unspecified-behavior` When specification specifies two or more ways to do it and imposes no requirement on choosing one of them then the behavior is unspecified as it is not mentioned in specification.

8. `bit` Now we are encountering the most important entity in programming that is a bit. Specification says it should be large enough to store one of two values. These values are as mentioned in Chapter 1.

9. `byte` This is what is written in specificaiton. ``addressable unit of data storage large enough to hold any member of the basic character set of the execution environment''. Since the very basic chcaracter set of C is `char` which has size of 1 byte. However, `char` is defined as an integral value having 256 distinct numbers. 256 being equal to 2^8 implies that a byte in C is made up of 8-bits.

10. `character` It is specified as a set of elements used for organization, control and representation of data.

    I. `single-byte character` It is specified as bit representation that fits in a byte. This confirms our calculation of number of bits in a byte.

    II. `multi-byte character` It is specified as sequence of one or more bytes, however, when more comes then it can be more than two or three or any integer. But for all practical puposes UTF-16 serves the usage.

    III. `wide character` It is specifies as sequence of bits which can fit in an object of type `wchar_t`. Using `wchar_t` we can represent any symbol of UTF-16 character set. It is a two-byte character.

11. `forward reference` You can look up element 3.11 of specification for it. By forward reference I mean that the complete type is not known or it is an *incomplete type* where used.

12. `3.12 and 3.13` Please look this up in specification at respective places in specification.

13. `object` It is specified as region of data storage in the execution environment (CPU or RAM), the content of which can represent values. These values are always concrete in nature and at any moment are finite. I say so because CPU or RAM will store bits which will always be some finite boolean voltage levels. I will also mean this as an instance of declaration of a data type primitive or non-primitive data type because these instances will have these voltage levels in the execution environment. When we access **21** these

objects then we use the very basic nature of these objects that is type of these objects and interpret the values accordingly.

14. `parameter` These are called formal parameters or formal argument. The usage of word formal argument is deprecated. These are very similar to actual arguments but these acquire value only after function is entered. At the time of declaration or definition these have no values. Same concept of macros which were applicable for **23**.

15. `value` The specification says in section 3.17 that value is ``precise meaning of the contents of an object when interpreted as having a specific type''. However, I would like to replace the word precise with accurate as both the words have different meaning and here accurate is what is more suitable.

    I. implementation-defined value It is a value which is unspecified in specification and implementation documents everything about it like how it is generated.

Index