

Learning Correct Programming

C

Shiv S Dayal

Contents

Part 1 C	9
1.1 Introduction	11
1.1.1 Alphabets, and Dictionary of C	12
1.1.2 Tools of Trade	12
1.1.3 First Program	13
1.1.4 Attributes of a Program	14
1.1.5 <code>printf</code> and <code>scanf</code>	15
1.1.5.1 Format Argument of <code>printf</code>	16
1.1.5.2 <code>scanf</code>	16
1.2 Following Specification	17
1.2.1 Terms, Definitions, and Symbols	17
1.3 Environment	21
1.3.1 Conceptual Model	21
1.3.1.1 Translation environment	21
1.3.1.1.1 Program Structure	21
1.3.1.1.1.1 Translation Phases	22
1.3.2 Diagnostics	22
1.3.3 Execution Environments	23
1.3.4 Environmental Conditions	23
1.3.4.1 Trigraph Sequences	23
1.3.5 Escape Sequences	24
1.3.6 Signals and Interrupts	25
1.3.6.1 Environmental Limits	25
1.3.6.1.1 Translation Limits	25
1.3.6.1.2 Numerical Limits	27
Index	31

Tables

1.1	C Keywords	12
3.1	Trigraph Sequences	24
3.2	Numerical Limits	27

Figures

1.1	Output of <code>hello.c</code>	14
1.2	Declaration of <code>printf</code>	15
1.3	Declaration of <code>scanf</code>	16
2.1	Output of <code>align.c</code>	19

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 remain 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 their 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.

Prerequisite

As such there are not many prerequisite. This book is well suited for beginners as well as advanced programmers. Since I have tried to follow C99 standard the terminology may be a bit different. I expect the reader to be at least

having high-school level knowledge of Mathematics. If the reader does not have this prerequisite then I strongly recommend him to study it as some of the sections may be difficult for him to comprehend. Programmers from any language will be able to make rapid advances while studying the text and may even skip some sections or chapters entirely.

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 subtle 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`, `TEXinfo`, `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 errors 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 shivshankar.dayal@gmail.com. I will try to give reply to each of your mail.

Shiv Shankar Dayal

Part I

C

1.1 Introduction

1.1.1 Alphabets, and Dictionary of C	12	1.1.3 First Program	13	1.1.5 printf and scanf ..	15
1.1.2 Tools of Trade	12	1.1.4 Attributes of a Program	14		

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 weathered 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 appeared 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 and 7". 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 voltage levels will change. For other characters like of English language or punctuation 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 respectively. UTF-16 can accommodate 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. Similarly, 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 operating 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 syntactic rules of a programming language and a tool like lex and yacc or flex and bison can use this grammar to help you develop compiler for that language for which the grammar was written. 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 of C language as specified in C99 standard. These words are also known 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-browser which gives me sufficient 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

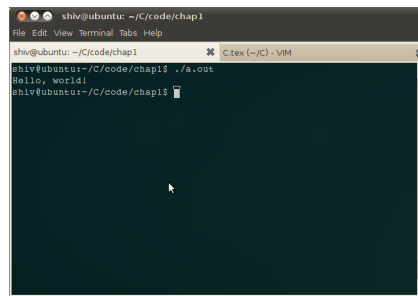
To compile this program just type following at your command prompt if you are using one:

```
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 screenshot.

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.

Next line is #include <stdio.h>. This actually consists of two parts. First part is #include. It is a pre-processor directive. By using such directives you can include any kind of file to your source code. There is



```

shiv@ubuntu: ~/C/code/chap1
File Edit View Terminal Tabs Help
shiv@ubuntu: ~/C/code/chap1
shiv@ubuntu:~/C/code/chap1$ ./a.out
Hello, world!
shiv@ubuntu:~/C/code/chap1$

```

Figure 1.1 Output of hello.c

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 syntactic sugar and 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 functionality is provided by compiler vendors.

Next line is `int main()`. It declares a special function `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` preceding it is the return type of this function. `int` means integer. So, probably now you can guess that `main` is supposed to return an integer to its caller. In this case the caller is the shell from which `./a.out` is being executed i.e. bash shell on my system. In your case 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 structured programming you break down the problem in sub-tasks and write one function for each of these tasks. If the sub-tasks are too 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 arguments as we will soon see. This function's prototype is defined in `stdio.h` and the functionality 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 meant for different purposes. They may be small, large or mid-sized. However, there are certain criteria 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 fulfil 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-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 valid 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. First function is an 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 our 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 output. 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 similarly send one or more arguments to `printf`. You can also see that `printf` is returning an 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 more 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

1.2.1 Terms, Definitions, and Symbols 17

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. The following terms, definitions and symbols have come from specification, however, some are omitted for the sake of conciseness. When I will repeat the specification at times I will do a verbatim copy just for quick reference and then add an explanation to that.

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:

I am always showing screenshots of program's output but now onwards I will not show them as they just consume space. I will also explain the output at a later stage when you will understand `struct`.

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

```

shiv@ubuntu: ~/C/code/chap2
File Edit View Terminal Tabs Help
Introduction.txt (~C) - VIM
shiv@ubuntu:~/C/code/chap2$ ./a.out
Size of structure A is 4
Size of structure B is 16
Size of structure C is 8
Size of structure D is 12
shiv@ubuntu:~/C/code/chap2$ 

```

Figure 2.1 Output of align.c

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 specification. "addressable unit of data storage large enough to hold any member of the basic character set of the execution environment". Since the very basic character 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 purposes UTF-16 serves the usage.
 - III. **wide character** It is specified 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 17 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 18.
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.
 - II. `indeterminate value` Indeterminate values are those which can not be determined at any particular point of time with accuracy. For example, unspecified values and trap representations.
 - III. `unspecified value` Any object can have multiple values one at a point of time. When specification does not require that implementation chooses any one of them particularly then it is called unspecified value.
16. $\lceil x \rceil$ This has usual meaning that is ceiling of x .
17. $\lfloor x \rfloor$ This has usual meaning that is floor of x .

This concludes our discussion on terms, definitions and symbols. We will proceed with more of specification in next chapter.

1.3 Environment

1.3.1 Conceptual Model	21	1.3.4 Environmental	
1.3.2 Diagnostics	22	Conditions	23
1.3.3 Execution		1.3.5 Escape Sequences	24
Environments	23	1.3.6 Signals and Interrupts	25

As last chapter was mapped to chapter 3 of specification similarly this chapter is mapped to chapter 5 of specification. Please refer to these sections simultaneously.

We store source code in two different files with two different extensions. One regular expression for source code is `*.c` and the other is `*.h`. These files are stored in hard disk. When compiled it may produce `*.o` files optionally. Your program may also produce `*.a`, `*.so`, `a.out` (this name may be different for you if you provide `-o` switch to gcc. At least one `*.a` or `*.so` or `a.out` or its equivalent will be produced on unix systems. On Windows systems you may produce `*.lib`, `*.dll` or `*.exe`. `*.lib`, `*.dll` and `*.exe` map directly to their counterparts in unix as `*.a`, `*.so` and `a.out`. These files are called static library, dynamic library, and executables respectively.

As per specification we are first going to consider conceptual model then environmental considerations.

1.3.1 Conceptual Model

1.3.1.1 Translation environment

1.3.1.1.1 Program Structure

A C program typically is never translated together if there are multiple source files (i.e. several `*.c` files). These files are also called preprocessing files in the specification. A source file with all the headers included by the preprocessor directive `#include` is called preprocessing translation unit. After preprocessing, a preprocessing translation unit is called a translation unit. So a preprocessing translation unit can be obtained by running `gcc -E filename.c` and such a translation unit can be obtained by running `gcc -c filename.c`. When we run `gcc -E filename.c` is sent to STDOUT i.e. monitor or screen so if you want to see the output then you need to redirect it to a file. You can give a command like `gcc -E filename.c > filename.extension` so that new file with name `filename.extension` is saved to hard disk and can be viewed later. When you give command `gcc -c filename.c` it automatically generates corresponding `filename.o` unless you specify something like `gcc -c filename.c -o anotherfilename.o`. This however is not a good idea and you should refrain from it as it may lead to filename clashes sooner or later if you try this manual option. So any `*.o` file is a translation unit. It is also called object code. Existing libraries like `libc.so` or `libm.a` is a combination of many such translation units or `*.o` files. When we wrote our first program we got an `a.out` file and no `*.o` file. Just type `ldd a.out` in the directory where `a.out` is and you will see what libraries it depends on. It will certainly point to `libc.so`. The reason is simple. Remember our `printf` function that has got its logic in this `libc.so` or in any other library which `libc.so` depends on. You may see something like `/lib/tls/i686/cmov/libc.so.6` instead of `libc.so`. Therefore we conclude that at runtime either `a.out` will take the definition of `printf` from `libc.so` or at compile time itself the logic of `printf` is embedded in `a.out`. First operation is called dynamic linking and loading and second is called static linking. Just try `gcc -static hello.c` and `ldd a.out` and you will see some very nice output. :-) Do not worry it is not going to crash your operating system or delete any data. So by now you can infer that if we want to do runtime linking then presence of required libraries at runtime is a must and if we do static linking then you do not need these libraries once they are linked with. People in general prefer dynamic linking

because a common function like `printf` should not have multiple copies in memory as memory is costly and it also violates the “Efficiency” requirement of a program. How exactly dynamic and static linking happens will become clear only after ELF (Executable and Linkable Format) file format is described. I will try to cover it if time permits. ELF is a typical standard like C99 on Unix systems. There is another format whose name you know but is slightly older and that is “a.out”. Probably now you can guess why by default gcc puts the executable output with filename a.out. As you will soon see that these *.o files or translation units can be separately translated or compiled and linked at a later stage to produce an executable file or linkable library.

1.3.1.1.1 Translation Phases

Please refer to section 5.1.1.2 of specification. There are eight steps specified for translation phases. I will cover how gcc behaves. gcc is a multi-pass compiler. Several switches control its behavior. These switches are `-E`, `-S`, `-c`. We have already seen first `-E` switch. Let us see the `-S` switch. When you invoke gcc with `-S` switch it generates equivalent assembly code and does not generate object code i.e. it does not assemble or link. Given below is the sample output of command `gcc -S hello.c`.

```

1      .file    "hello.c"
2      .section      .rodata
3  .LC0:
4      .string "Hello, world!"
5      .text
6  .globl main
7      .type     main, @function
8  main:
9      pushl    %ebp
10     movl     %esp, %ebp
11     andl     $-16, %esp
12     subl     $16, %esp
13     movl     $.LC0, (%esp)
14     call     puts
15     movl     $0, %eax
16     leave
17     ret
18     .size    main, .-main
19     .ident    "GCC: (Ubuntu 4.4.3-4ubuntu5) 4.4.3"
20     .section      .note.GNU-stack,"",@progbits

```

Hellow World Program in Assembly

The code shown above is in `as` syntax. `as` stands for Assembler. Once assembly code is generated then assembler assembles it to *.o file. Then `ld` the GNU linker and loader comes and links it to a binary executable file. I think I am going a bit too wayward as far as subject is concerned. Please see documentation for gcc, as, and ld for complete reference. So all in all we have seen at least four stages namely they are preprocessing with `-E` switch, generating assembly code with `-S`, producing object code with `as`, and finally linking with `ld`. All of this may be hidden from you or combined together. Typically on large projects you will combine first three steps.

1.3.2 Diagnostics

Section 5.1.1.3 of specification tells about implementation producing diagnostics messages in case of any prob-

lem so that the problems in the source code can be sorted out. gcc sends all errors and warnings and other diagnostics messages to `STDERR` device which is typically monitor or screen. My discussion is more becoming oriented towards specification and gcc however this is necessary until we hit the stage where we can write some serious program or discuss programming language's features.

1.3.3 Execution Environments

Please go through complete section 5.1.2 of specification as it is too boring to describe everything in a book of C language. The only things which I think must be mentioned is `main` function and the way it is specified. It says the implementation must not provide a prototype for `main` function. It also describes two possible ways in which `main` can be written. These two ways are given below:

```
1 int main(void) { /* ... */ }
2 int main(int argc, char *argv[]) { /* ... */ }
```

main function

The block between `/*` and `*/` is supposed to contain code not only comments. :-) The parameters shown may have different names. I interpret `argc` as argument count of the arguments passed to the program containing this `main` and `argv` as argument value so if that is fine with you you can use them as well. There are some more interesting things there.

1. The value of `argc` will be nonnegative. This is obvious because we can not enter nonnegative number of arguments to the program.
2. `argv[argc]` shall be a `NULL` pointer. This will become clear later when we discuss array and pointers.
3. `argv[0]` shall be program name if the host operating system provides it.

Rest of the points you can study yourself. I am itching to write some code. You know I am not much of an author but a programmer who is trying to be an author.

1.3.4 Environmental Conditions

Out of environmental conditions I have already described the C alphabets or rather character sets (following the specification language :P) in the first chapter itself. Now I am going to write some blah, blah about Trigraph Sequences.

1.3.4.1 Trigraph Sequences

In this world many countries and languages are there. All countries do not speak english and hence their character sets are also different. But a very subtle change happens because of this and that is that some of the keyboards have missing characters from the C character set. Now the system has stabilized a lot but it was not so when C was first specified. So to counter such keyboards Trigraph sequences came into picture. Given below is a table of all trigraph sequences. They are just replaced with equivalent characters if they are encountered anywhere in the source code.

Trigraph	Equivalent	Trigraph	Equivalent	Trigraph	Equivalent
??=	#	??'	^	??!	
??([??)]	??<	{
??/	\	??>	}	??-	~

Table 3.1 Trigraph Sequences

One more important topic in section 5.2.2 is character display semantics which I am going to cover next. I will be treating only item number 2 which talks about escape sequences.

1.3.5 Escape Sequences

These are very important and some of them you will use very often. These escape sequences apply to the display device or your monitor or screen. To learn about these escape sequences it is a must to know what is an active position. The specification specifies that active position is the position where the next character will be printed on the device if `fputc` function is used to write this character. This usually means the position of cursor or point on your terminal device or text editors. On a file basis they are stored as one character not two.

1. `\a` (alert) should produce a visible or audible alert without changing the system. Typically a program like `gnome-terminal` or `konsole` which are both terminal application for GNOME and KDE are capable of producing either depending upon configuration.
2. `\b` (backspace) should move the active position to the previous position on the current line. This is again a requirement of terminal applications. This is required for backward movement of cursor or point and inserting some text over there. Let us write a simple program to test this.

```

1  /*
2   * Author: Shiv Shankar Dayal
3   * Date: July 4th, 2010
4   * Description: Demonstration of backspace escape sequence.
5   */
6
7  #include <stdio.h>
8  int main()
9  {
10     int i=0;
11     printf("hello\b");
12     scanf("%d",&i);
13
14     return 0;
15 }
```

Demonstration of backspace `\b` escape sequence

Just watch the position of cursor on terminal. It should be over 'o' not after it.

3. `\f` (form feed) is specified to move the active position to initial position of next logical page. Let us write another program to test it. I will just replace `\b` with `\f`.

```

1  /*
```

```

2      * Author: Shiv Shankar Dayal
3      * Date: July 4th, 2010
4      * Description: Demonstration of form feed escape sequence.
5      */
6
7      #include <stdio.h>
8      int main()
9      {
10         int i=0;
11         printf("hello\f");
12         scanf("%d",&i);
13
14         return 0;
15     }

```

Demonstration of form feed \f escape sequence.

just notice how the cursor or active position advances exactly one line of terminal. At least that is the behavior on my machine.

4. \n (new line) This is simple and I will not provide a program for this one. The specification says the active position will be moved to initial position of next line. You have already seen this in hello.c, our first program.
5. \r (carriage return) This is similar to new line the only difference being that it moves the active position to initial position of current line instead of next line.
6. \t (horizontal tab) It moves active position by one tab horizontally. Typically it is configurable and of size 2, 4 and 8 characters (monospaced). This is one which you will encounter a lot in text editors than in terminal applications. If the active position is at last tabulation position or past it then behavior is unspecified.
7. \v (vertical tab) Same as horizontal tab just the movement is vertical.

The implementation of alert, new line, carriage return, and both the tabs is left to the reader as exercise.

1.3.6 Signals and Interrupts

At software level Unix systems have concept of signal which one process can send to another and there are interrupts which are of two types software and hardware. Please refer to section 5.2.3 and 5.2.4 for details as I do not want to just copy that as it is very difficult to add an explanation at this moment because I have not touched signals and interrupts at all.

1.3.6.1 Environmental Limits

1.3.6.1.1 Translation Limits

Some of these which are trivial and may need a forward reference are just copy of specification and explaining them would require introduction of many more terms. Specification specifies following translation limits.

1. 127 levels of nesting blocks. As I have told you one block is identified by a pair of matching braces. This means 127 nested (one block inside another) blocks must be possible.
2. 63 levels of nesting inclusion. I have not yet touched the control flow of conditional statements. The keyword `if` and `else` or `switch` come under this category. Nesting inclusion means one `if/else` inside another `if/else` block.
3. 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic structure, union, or incomplete type in declaration. This statement itself is self explanatory. The difference between declaration/definition will soon be clear. I have used the word declarator so I will try to show its meaning right here. Declarator. From specification if you see section 6.7.5 declarator means the concrete object. However, I tested with gcc and it supports more than 12. But if you want to write portable code this must not be violated.
4. 63 nesting levels of parenthesized declarators within a full declarator. Again it is self explanatory. A parenthesized declarator is like data-type (declarator). But if you want to write portable code this must not be violated.
5. 63 nesting levels of parenthesized expressions within a full expression. Please find the given definition of expression. I will quote this directly from specification so that its accurate meaning is not interpreted in different way. "An expression is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof." All the terms which are not clear at the moment will become clear soon. Implementation may support more but if you want to write portable code this must not be violated.
6. 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character). However, certain implementations also allow 127 or 255 significant initial characters. Please refer to your compiler manual. But if you want to write portable code this must not be violated.
7. 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any). Again if you want to write portable code do not rely on implementation but rather stick to standard.
8. 4095 external identifiers in one translation unit. Once again same consideration should be taken for portability. Even if your compiler suite remains same across different platforms (hardware and operating system) the implementations may vary for the same vendor.
9. 511 identifiers with block scope declared in one block.
10. 4095 macro identifiers simultaneously defined in one preprocessing translation unit
11. 127 parameters in one function definition and similarly 127 arguments in one function call. This also means `printf` can take only 126 extra parameters/arguments in call.
12. 127 parameters in one macro definition or 127 arguments in one macro invocation. A macro is a preprocessor directive. Most famous is `#define`.

13. 4095 characters in a logical source line. Though old coding guidelines will restrict you to 80 characters. But with modern graphics displays these can be more. However, Leslie Lamport the L^AT_EX creator says it is difficult to read a line if it contains more than 70/72 characters. It is a kind of rule of typography.

14. 4095 characters in a character string literal or wide string literal (after concatenation)

15. 65535 bytes in an object (in a hosted environment only). This may be a major hinderance in development. However implementation may be different.

16. 15 nesting levels for `#include` files. An include file may include another and this can be nested. However there should not be more than 15 levels if you want portability.

17. 1023 case labels for a `switch` statement (excluding those for any nested `switch` statements). This I will describe in detail when we see control flows.

18. 1023 members in a single structure or union.

19. 1023 enumeration constants in a single enumeration.

20. 1023 enumeration constants in a single enumeration.

You must have noticed that all the numbers are $2^n - 1$ where n is an integer and you also know the reason why. If you are not clear then you must wait for answer.

1.3.6.1.2 Numerical Limits

These limits come from specification as well as compiler implementation. Three header files contain these details. `limits.h`, `float.h` and `stdint.h`. I will prepare a table for all the limits and show you if there is a difference between my headers and specification. I guess there should not be any.

Data Type/Macro	Specified Value	limits.h value	Description
CHAR_BIT	8	8	No. of bits in char type.
SCHAR_MIN	-127 or $-(2^7 - 1)^1$	-128	Minimum value of signed char
SCHAR_MAX	127 or $2^7 - 1$	127	Maximum value of signed char
UCHAR_MAX	255 or $2^8 - 1$	255	Maximum value of unsigned char
CHAR_MIN	See below ²	See below ²	Maximum value of unsigned char
CHAR_MAX	See below ²	See below ²	See below ²
MB_LEN_MAX	1	16	Maximum no. of bytes in multi-byte character, supporting that locale.

Table 3.2 Numerical Limits

¹ I do not agree with 127 being equal to minimum value of a signed `char`. If we see 2's complement form then it should be -128. Please correct me if I am wrong.

² `char`

Index

a

align.c 18

b

backspace.c 24, 25

d

Dennis

Ritchie 11

e

escape sequence

backspace 24, 25

h

hello.c 13

hello.s 22

m

main function 23

Memory Alignment Program 18

p

printf 16

s

scanf 16

stdio.h 13

