

Mc
Graw
Hill Education

大学计算机教育国外著名教材系列



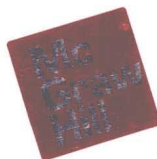
Programming in ANSI C

Fifth Edition

标准C程序设计 (第5版)



E Balagurusamy 著



清华大学出版社

名教材系列（影印版）

Programming in ANSI C

Fifth Edition

标准 C 程序设计

（第 5 版）

E Balagurusamy

清华大学出版社
北 京

E Balagurusamy
Programming in ANSI C, Fifth Edition
EISBN: 0-07-068182-1

Copyright © 2011 by The McGraw-Hill Companies, Inc.

Original language published by The McGraw-Hill Companies, Inc. All Rights reserved. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Authorized English language edition jointly published by McGraw-Hill Education (Asia) Co. and Tsinghua University Press. This edition is authorized for sale only to the educational and training institutions, and within the territory of the People's Republic of China (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书英文影印版由清华大学出版社和美国麦格劳-希尔教育出版(亚洲)公司合作出版。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾地区)针对教育及培训机构之销售。未经许可之出口,视为违反著作权法,将受法律之制裁。未经出版者预先书面许可,不得以任何方式复制或抄袭本书的任何部分。

北京市版权局著作权合同登记号 图字: 01-2011-4034

本书封面贴有 McGraw-Hill 公司防伪标签,无标签者不得销售。
版权所有,侵权必究。侵权举报电话: 010-62782989 13701121933

图书在版编目(CIP)数据

标准 C 程序设计=Programming in ANSI C, Fifth Edition: 第 5 版: 英文 / (印)巴拉古路萨米 (Balagurusamy, E.) 著.--影印本.--北京: 清华大学出版社, 2011.10

(大学计算机教育国外著名教材系列)

ISBN 978-7-302-26419-4

I. ①标… II. ①巴… III. ①C 语言—程序设计—高等学校—教材—英文 IV. ①TP312

中国版本图书馆 CIP 数据核字 (2011) 第 161355 号

责任编辑: 龙啟铭
责任印制: 王秀菊

出版发行: 清华大学出版社
http://www.tup.com.cn
社 总 机: 010-62770175
投 稿 与 读 者 服 务: 010-62795954, jsjic@tup.tsinghua.edu.cn
质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

地 址: 北京清华大学学研大厦 A 座
邮 编: 100084
邮 购: 010-62786544

印 装 者: 三河市李旗庄少明装订厂
发 行 者: 全国新华书店
开 本: 148×210 印张: 17.5
版 次: 2011 年 10 月第 1 版
印 数: 1~3000
定 价: 29.00 元

印 次: 2011 年 10 月第 1 次印刷

产品编号: 042470-01

出版说明

进入 21 世纪，世界各国的经济、科技以及综合国力的竞争将更加激烈。竞争的中心无疑是对人才的竞争。谁拥有大量高素质的人才，谁就能在竞争中取得优势。高等教育，作为培养高素质人才的事业，必然受到高度重视。目前我国高等教育的教材更新较慢，为了加快教材的更新频率，教育部正在大力促进我国高校采用国外原版教材。

清华大学出版社从 1996 年开始，与国外著名出版公司合作，影印出版了“大学计算机教育丛书（影印版）”等一系列引进图书，受到国内读者的欢迎和支持。跨入 21 世纪，我们本着为我国高等教育教材建设服务的初衷，在已有的基础上，进一步扩大选题内容，改变图书开本尺寸，一如既往地请有关专家挑选适用于我国高等本科及研究生计算机教育的国外经典教材或著名教材，组成本套“大学计算机教育国外著名教材系列（影印版）”，以飨读者。深切期盼读者及时将使用本系列教材的效果和意见反馈给我们。更希望国内专家、教授积极向我们推荐国外计算机教育的优秀教材，以利我们把“大学计算机教育国外著名教材系列（影印版）”做得更好，更适合高校师生的需要。

清华大学出版社

Contents

1 Overview of C	1
1.1 History of C	1
1.2 Importance of C	3
1.3 Sample Program 1: Printing a Message	3
1.4 Sample Program 2: Adding Two Numbers	6
1.5 Sample Program 3: Interest Calculation	8
1.6 Sample Program 4: Use of Subroutines	10
1.7 Sample Program 5: Use of Math Functions	11
1.8 Basic Structure of C Programs	12
1.9 Programming Style	14
1.10 Executing a 'C' Program	14
1.11 Unix System	16
1.12 Ms-Dos System	18
<i>Review Questions</i>	19
<i>Programming Exercises</i>	20
2 Constants, Variables, and Data Types	23
2.1 Introduction	23
2.2 Character Set	23
2.3 C Tokens	25
2.4 Keywords and Identifiers	25
2.5 Constants	26
2.6 Variables	30
2.7 Data Types	31
2.8 Declaration of Variables	34
2.9 Declaration of Storage Class	37
2.10 Assigning Values to Variables	38
2.11 Defining Symbolic Constants	44
2.12 Declaring a Variable as Constant	45
2.13 Declaring a Variable as Volatile	45

2.14	Overflow and Underflow of Data	46	
	<i>Review Questions</i>	49	
	<i>Programming Exercises</i>	51	
3	Operators and Expressions		52
3.1	Introduction	52	
3.2	Arithmetic Operators	52	
3.3	Relational Operators	55	
3.4	Logical Operators	57	
3.5	Assignment Operators	57	
3.6	Increment and Decrement Operators	59	
3.7	Conditional Operator	61	
3.8	Bitwise Operators	61	
3.9	Special Operators	61	
3.10	Arithmetic Expressions	63	
3.11	Evaluation of Expressions	64	
3.12	Precedence of Arithmetic Operators	65	
3.13	Some Computational Problems	67	
3.14	Type Conversions in Expressions	68	
3.15	Operator Precedence and Associativity	72	
3.16	Mathematical Functions	74	
	<i>Review Questions</i>	78	
	<i>Programming Exercises</i>	81	
4	Managing Input and Output Operations		84
4.1	Introduction	84	
4.2	Reading a Character	85	
4.3	Writing a Character	88	
4.4	Formatted Input	89	
4.5	Formatted Output	98	
	<i>Review Questions</i>	110	
	<i>Programming Exercises</i>	112	
5	Decision Making and Branching		114
5.1	Introduction	114	
5.2	Decision Making with IF Statement	114	
5.3	Simple IF Statement	115	
5.4	The IF.....ELSE Statement	119	
5.5	Nesting of IF....ELSE Statements	122	
5.6	The ELSE IF Ladder	126	
5.7	The Switch Statement	129	
5.8	The ?: Operator	133	
5.9	The GOTO Statement	136	
	<i>Review Questions</i>	144	
	<i>Programming Exercises</i>	148	

6	Decision Making and Looping	152
6.1	Introduction	152
6.2	The WHILE Statement	154
6.3	The DO Statement	157
6.4	The FOR Statement	159
6.5	Jumps in LOOPS	166
6.6	Concise Test Expressions	174
	<i>Review Questions</i>	182
	<i>Programming Exercises</i>	186
7	Arrays	190
7.1	Introduction	190
7.2	One-dimensional Arrays	192
7.3	Declaration of One-dimensional Arrays	193
7.4	Initialization of One-dimensional Arrays	195
7.5	Two-dimensional Arrays	199
7.6	Initializing Two-dimensional Arrays	204
7.7	Multi-dimensional Arrays	208
7.8	Dynamic Arrays	209
7.9	More about Arrays	209
	<i>Review Questions</i>	223
	<i>Programming Exercises</i>	225
8	Character Arrays and Strings	229
8.1	Introduction	229
8.2	Declaring and Initializing String Variables	230
8.3	Reading Strings from Terminal	231
8.4	Writing Strings to Screen	236
8.5	Arithmetic Operations on Characters	241
8.6	Putting Strings Together	242
8.7	Comparison of Two Strings	244
8.8	String-handling Functions	244
8.9	Table of Strings	250
8.10	Other Features of Strings	252
	<i>Review Questions</i>	257
	<i>Programming Exercises</i>	259
9	User-defined Functions	262
9.1	Introduction	262
9.2	Need for User-defined Functions	262
9.3	A Multi-function Program	263
9.4	Elements of User-defined Functions	266
9.5	Definition of Functions	267
9.6	Return Values and their Types	269
9.7	Function Calls	270
9.8	Function Declaration	272

9.9	Category of Functions	274	
9.10	No Arguments and no Return Values	274	
9.11	Arguments but no Return Values	277	
9.12	Arguments with Return Values	280	
9.13	No Arguments but Returns a Value	284	
9.14	Functions that Return Multiple Values	285	
9.15	Nesting of Functions	286	
9.16	Recursion	288	
9.17	Passing Arrays to Functions	289	
9.18	Passing Strings to Functions	294	
9.19	The Scope, Visibility and Lifetime of Variables	295	
9.20	Multifile Programs	305	
	<i>Review Questions</i>	311	
	<i>Programming Exercises</i>	315	
10	Structures and Unions		317
10.1	Introduction	317	
10.2	Defining a Structure	317	
10.3	Declaring Structure Variables	319	
10.4	Accessing Structure Members	321	
10.5	Structure Initialization	322	
10.6	Copying and Comparing Structure Variables	324	
10.7	Operations on Individual Members	326	
10.8	Arrays of Structures	327	
10.9	Arrays within Structures	329	
10.10	Structures within Structures	331	
10.11	Structures and Functions	333	
10.12	Unions	335	
10.13	Size of Structures	337	
10.14	Bit Fields	337	
	<i>Review Questions</i>	344	
	<i>Programming Exercises</i>	348	
11	Pointers		351
11.1	Introduction	351	
11.2	Understanding Pointers	351	
11.3	Accessing the Address of a Variable	354	
11.4	Declaring Pointer Variables	355	
11.5	Initialization of Pointer Variables	356	
11.6	Accessing a Variable through its Pointer	358	
11.7	Chain of Pointers	360	
11.8	Pointer Expressions	361	
11.9	Pointer Increments and Scale Factor	362	
11.10	Pointers and Arrays	364	
11.11	Pointers and Character Strings	367	
11.12	Array of Pointers	369	

11.13	Pointers as Function Arguments	370	
11.14	Functions Returning Pointers	373	
11.15	Pointers to Functions	373	
11.16	Pointers and Structures	376	
11.17	Troubles with Pointers	379	
	<i>Review Questions</i>	385	
	<i>Programming Exercises</i>	388	
12	File Management in C		389
12.1	Introduction	389	
12.2	Defining and Opening a File	390	
12.3	Closing a File	391	
12.4	Input/Output Operations on Files	392	
12.5	Error Handling During I/O Operations	398	
12.6	Random Access to Files	400	
12.7	Command Line Arguments	405	
	<i>Review Questions</i>	408	
	<i>Programming Exercises</i>	409	
13	Dynamic Memory Allocation and Linked Lists		411
13.1	Introduction	411	
13.2	Dynamic Memory Allocation	411	
13.3	Allocating a Block of Memory: MALLOC	413	
13.4	Allocating Multiple Blocks of Memory: CALLOC	415	
13.5	Releasing the Used Space: Free	415	
13.6	Altering the Size of a Block: REALLOC	416	
13.7	Concepts of Linked Lists	417	
13.8	Advantages of Linked Lists	420	
13.9	Types of Linked Lists	421	
13.10	Pointers Revisited	422	
13.11	Creating a Linked List	424	
13.12	Inserting an Item	428	
13.13	Deleting an Item	431	
13.14	Application of Linked Lists	433	
	<i>Review Questions</i>	440	
	<i>Programming Exercises</i>	442	
14	The Preprocessor		444
14.1	Introduction	444	
14.2	Macro Substitution	445	
14.3	File Inclusion	449	
14.4	Compiler Control Directives	450	
14.5	ANSI Additions	453	
	<i>Review Questions</i>	456	
	<i>Programming Exercises</i>	457	

15	Developing a C Program: Some Guidelines	458
15.1	Introduction	458
15.2	Program Design	458
15.3	Program Coding	460
15.4	Common Programming Errors	462
15.5	Program Testing and Debugging	469
15.6	Program Efficiency	471
	<i>Review Questions</i>	472
	Appendix I: Bit-level Programming	474
	Appendix II: ASCII Values of Characters	480
	Appendix III: ANSI C Library Functions	482
	Appendix IV: Projects	486
	Appendix V: C99 Features	537
	Bibliography	545
	Index	547

Overview of C

1.1 HISTORY OF C

'C' seems a strange name for a programming language. But this strange sounding language is one of the most popular computer languages today because it is a structured, high-level, machine independent language. It allows software developers to develop programs without worrying about the hardware platforms where they will be implemented.

The root of all modern languages is ALGOL, introduced in the early 1960s. ALGOL was the first computer language to use a block structure. Although it never became popular in USA, it was widely used in Europe. ALGOL gave the concept of structured programming to the computer science community. Computer scientists like Corrado Bohm, Guiseppe Jacopini and Edsger Dijkstra popularized this concept during 1960s. Subsequently, several languages were announced.

In 1967, Martin Richards developed a language called BCPL (Basic Combined Programming Language) primarily for writing system software. In 1970, Ken Thompson created a language using many features of BCPL and called it simply B. B was used to create early versions of UNIX operating system at Bell Laboratories. Both BCPL and B were "typeless" system programming languages.

C was evolved from ALGOL, BCPL and B by Dennis Ritchie at the Bell Laboratories in 1972. C uses many concepts from these languages and added the concept of data types and other powerful features. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system, which was also developed at Bell Laboratories, was coded almost entirely in C. UNIX is one of the most popular network operating systems in use today and the heart of the Internet data superhighway.

For many years, C was used mainly in academic environments, but eventually with the release of many C compilers for commercial use and the increasing popularity of UNIX, it began to gain widespread support among computer professionals. Today, C is running under a variety of operating system and hardware platforms.

During 1970s, C had evolved into what is now known as "traditional C". The language became more popular after publication of the book *'The C Programming Language'* by Brian Kerningham and Dennis Ritchie in 1978. The book was so popular that the language came to be known as "K&R C" among the programming community. The rapid growth of C led to the development of different versions of the language that were similar but often incompatible. This posed a serious problem for system developers.

To assure that the C language remains standard, in 1983, American National Standards Institute (ANSI) appointed a technical committee to define a standard for C. The committee approved a version of C in December 1989 which is now known as ANSI C. It was then approved by the International Standards Organization (ISO) in 1990. This version of C is also referred to as C89.

During 1990's, C++, a language entirely based on C, underwent a number of improvements and changes and became an ANSI/ISO approved language in November 1997. C++ added several new features to C to make it not only a true object-oriented language but also a more versatile language. During the same period, Sun Microsystems of USA created a new language **Java** modelled on C and C++.

All popular computer languages are dynamic in nature. They continue to improve their power and scope by incorporating new features and C is no exception. Although C++ and Java were evolved out of C, the standardization committee of C felt that a few features of C++/Java, if added to C, would enhance the usefulness of the language. The result was the 1999 standard for C. This version is usually referred to as C99. The history and development of C is illustrated in Fig. 1.1.

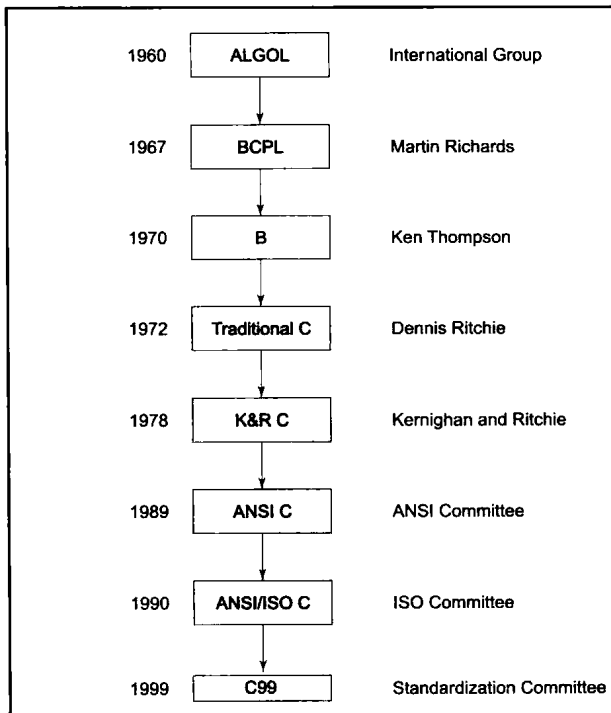


Fig. 1.1 History of ANSI C

Although C99 is an improved version, still many commonly available compilers do not support all of the new features incorporated in C99. We, therefore, discuss all the new features added by C99 in an appendix separately so that the readers who are interested can quickly refer to the new material and use them wherever possible.

1.2 IMPORTANCE OF C

The increasing popularity of C is probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assembly language with the features of a high-level language and therefore it is well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC. For example, a program to increment a variable from 0 to 15000 takes about one second in C while it takes more than 50 seconds in an interpreter BASIC.

There are only 32 keywords in ANSI C and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs.

C is highly portable. This means that C programs written for one computer can be run on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C language is well suited for structured programming, thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own functions to C library. With the availability of a large number of functions, the programming task becomes simple.

Before discussing specific features of C, we shall look at some sample C programs, and analyze and understand how they work.

1.3 SAMPLE PROGRAM 1: PRINTING A MESSAGE

Consider a very simple program given in Fig. 1.2.

```
main( )
{
/* .....printing begins.....*/
    printf("I see, I remember");
/* .....printing ends.....*/
}
```

Fig. 1.2 A program to print one line of text

This program when executed will produce the following output:

```
I see, I remember
```

Let us have a close look at the program. The first line informs the system that the name of the program is **main** and the execution begins at this line. The **main()** is a special function used by the C system to tell the computer where the program starts. Every program must have *exactly one main* function. If we use more than one **main** function, the compiler cannot understand which one marks the beginning of the program.

The empty pair of parentheses immediately following **main** indicates that the function **main** has no *arguments* (or parameters). The concept of arguments will be discussed in detail later when we discuss functions (in Chapter 9).

The opening brace “{” in the second line marks the beginning of the function **main** and the closing brace “}” in the last line indicates the end of the function. In this case, the closing brace also marks the end of the program. All the statements between these two braces form the *function body*. The function body contains a set of instructions to perform the given task.

In this case, the function body contains three statements out of which only the **printf** line is an executable statement. The lines beginning with **/*** and ending with ***/** are known as *comment* lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements and therefore anything between **/*** and ***/** is ignored by the compiler. In general, a comment can be inserted wherever blank spaces can occur—at the beginning, middle or end of a line—“but never in the middle of a word”.

Although comments can appear anywhere, they cannot be nested in C. That means, we cannot have comments inside comments. Once the compiler finds an opening token, it ignores everything until it finds a closing token. The comment line

```
/* = = = /* = = = */ = = = */
```

is not valid and therefore results in an error.

Since comments do not affect the execution speed and the size of a compiled program, we should use them liberally in our programs. They help the programmers and other users in understanding the various functions and operations of a program and serve as an aid to debugging and testing. We shall see the use of comment lines more in the examples that follow.

Let us now look at the **printf()** function, the only executable statement of the program.

```
printf("I see, I remember");
```

printf is a predefined standard C function for printing output. *Predefined* means that it is a function that has already been written and compiled, and linked together with our program at the time of linking. The concepts of compilation and linking are explained later in this chapter. The **printf** function causes everything between the starting and the ending quotation marks to be printed out. In this case, the output will be:

```
I see, I remember
```

Note that the print line ends with a semicolon. *Every statement in C should end with a semicolon (;) mark.*

Suppose we want to print the above quotation in two lines as

```
I see,  
I remember!
```

This can be achieved by adding another **printf** function as shown below:

```
printf("I see, \n");
printf("I remember !");
```

The information contained between the parentheses is called the *argument* of the function. This argument of the first `printf` function is "I see, \n" and the second is "I remember !". These arguments are simply strings of characters to be printed out.

Notice that the argument of the first `printf` contains a combination of two characters \ and n at the end of the string. This combination is collectively called the *newline* character. A newline character instructs the computer to go to the next (new) line. It is similar in concept to the carriage return key on a typewriter. After printing the character comma (,) the presence of the newline character \n causes the string "I remember !" to be printed on the next line. No space is allowed between \ and n.

If we omit the newline character from the first `printf` statement, then the output will again be a single line as shown below.

```
I see, I remember !
```

This is similar to the output of the program in Fig. 1.2. However, note that there is no space between , and I.

It is also possible to produce two or more lines of output by one `printf` statement with the use of newline character at appropriate places. For example, the statement

```
printf("I see,\n I remember !");
```

will output

```
I see,
I remember !
```

while the statement

```
printf("I\n.. see,\n... .. I\n... .. remember !");
```

will print out

```
I
.. see,
... .. I
... .. remember !
```

NOTE: Some authors recommend the inclusion of the statement

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

at the beginning of all programs that use any input/output library functions. However, this is not necessary for the functions `printf` and `scanf` which have been defined as a part of the C language. See Chapter 4 for more on input and output functions.

Before we proceed to discuss further examples, we must note one important point. C does make a distinction between *uppercase* and *lowercase* letters. For example, `printf` and `PRINTF` are not the same. In C, everything is written in lowercase letters. However, uppercase letters are used for symbolic names representing constants. We may also use uppercase letters in output strings like "I SEE" and "I REMEMBER"

The above example that printed `I see, I remember` is one of the simplest programs. Figure 1.3 highlights the general format of such simple programs. All C programs need a `main` function.


```

int number;          /* line-5 */
float amount;       /* line-6 */
                    /* line-7 */
number = 100;       /* line-8 */
                    /* line-9 */
amount = 30.75 + 75.35; /* line-10 */
printf("%d\n", number); /* line-11 */
printf("%.2f", amount); /* line-12 */
                    /* line-13 */
}

```

Fig. 1.4 Program to add two numbers

This program when executed will produce the following output:

```

100
106.10

```

The first two lines of the program are comment lines. It is a good practice to use comment lines in the beginning to give information such as name of the program, author, date, etc. Comment characters are also used in other lines to indicate line numbers.

The words **number** and **amount** are *variable names* that are used to store numeric data. The numeric data may be either in *integer* form or in *real* form. In C, *all variables should be declared* to tell the compiler what the *variable names* are and what *type of data* they hold. The variables must be declared before they are used. In lines 5 and 6, the declarations

```

int number;
float amount;

```

tell the compiler that **number** is an integer (**int**) and **amount** is a floating (**float**) point number. Declaration statements must appear at the beginning of the functions as shown in Fig.1.4. All declaration statements end with a semicolon; C supports many other data types and they are discussed in detail in Chapter 2.

The words such as **int** and **float** are called the *keywords* and cannot be used as *variable* names. A list of keywords is given in Chapter 2.

Data is stored in a variable by *assigning* a data value to it. This is done in lines 8 and 10. In line-8, an integer value 100 is assigned to the integer variable **number** and in line-10, the result of addition of two real numbers 30.75 and 75.35 is assigned to the floating point variable **amount**. The statements

```

number = 100;
amount = 30.75 + 75.35;

```

are called the *assignment* statements. Every assignment statement must have a semicolon at the end.

The next statement is an output statement that prints the value of **number**. The print statement

```

printf("%d\n", number);

```

contains two arguments. The first argument “%d” tells the compiler that the value of the second argument **number** should be printed as a *decimal integer*. Note that these arguments are separated by a comma. The newline character \n causes the next output to appear on a new line.