

C PROGRAMMING

Student Workbook

C PROGRAMMING

Jeff Howell

Published by ITCourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO 80112

Editor: Rick Sussenbach

Special thanks to: Many C instructors whose ideas and careful review have contributed to the quality of this workbook, and the many students who have offered comments, suggestions, criticisms, and insights.

Copyright © 2011 by ITCourseware, LLC. All rights reserved. No part of this book may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to itcourseware, LLC, 7245 South Havana Street, Suite 100, Centennial, CO, 80112. (303) 302-5280.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

CONTENTS

Chapter 1 - Course Introduction	1
Course Objectives	2
Course Overview	4
Suggested References	6
Chapter 2 - Introduction to C	9
What is C ?	10
Features of C	12
Why Program in C ?	14
History of C	16
Current Status and Future	18
Chapter 3 - An Overview of C	21
The First Program (hello.c)	22
How to Compile and Run a C Program	24
An Arithmetic Program (roof.c)	26
Execution Flow Control (mph.c)	28
The for Loop	30
The for Loop - Diagram	32
Character I/O	34
A File Copier Program (cp2.c)	36
A Character Counter (wc2.c)	38
A Look at Arrays	40
Stock Values (stock1.c)	42
The char Data Type	44
Strings (Character Arrays)	46
A String Copy Program (stringcp.c)	48
A Look at Functions	50
A Functional Program (func1.c)	52
A Review of printf()	54
Labs	56

Chapter 4 - Data Types and Variables	59
Fundamental Data Types	60
Data Type Values and Sizes	62
Data Type Values and Sizes	64
Variable Declarations	66
Variable Names	68
Constants	70
Character Constants	72
String Constants	74
Labs	76
Chapter 5 - Operators and Expressions	79
What are Expressions?	80
Arithmetic Operators	82
Relational Operators	84
Assignment Operator	86
Expressions Have Resulting Values	88
True and False	90
Logical Operators	92
Increment and Decrement Operators (++ and --)	94
Increment and Decrement Operators: Examples	96
'Operate-Assign' Operators (+=, *=, ...)	98
Conditional Expression	100
Operator Precedence	102
Precedence and Order of Evaluation	104
Evaluation of Logical Operators	106
Type Conversions	108
The Cast Operator	110
Bitwise Logical Operators	112
Labs	114

Chapter 6 - Control Flow	117
Statements	118
if - else	120
if() - else if()	122
switch()	124
while()	126
do - while()	128
for()	130
The for Loop - Diagram	132
Example: for() Loop	134
Another Example: for() Loop	136
The break Statement	138
The continue Statement	140
Labs	142
 Chapter 7 - Functions	 145
What is a Function?	146
Example: findbig3()	148
Why Use Functions?	150
Anatomy of a Function	152
Example: find_big_int()	154
Arguments Passed by Value	156
Addresses of Arguments Can Be Passed	158
A Picture of Addresses and Values	160
When to Use the Return Statement	162
Returning Non-Integer Values	164
Functions in Multiple Source Files	166
A Simple make File	168
The Concept of Variable Scope	170
Automatic Variables	172
Global (External) Variables	174
Static Variables	176
External Static Variables	178
Labs	180

Chapter 8 - The C Preprocessor	183
Symbolic Constants	184
Macro Substitution	186
File Inclusion	188
Labs	190
Chapter 9 - Pointers and Arrays	193
What is a Pointer?	194
Pointer Operators	196
Example: Pointers	198
Why Use Pointers?	200
Arrays	202
Arrays (a Picture)	204
The & Operator	206
Pointers and Arrays	208
Pointer Arithmetic	210
Pointer Arithmetic (a Picture)	212
Pointer Arithmetic	214
Arrays and Pointers	216
Arrays and Pointers (cont'd)	218
Array Names are Constant Pointers	220
Passing Arrays to Functions	222
Initializing Arrays	224
Labs	226
Chapter 10 - Advanced Pointers	229
Pointer Initialization	230
Command-Line Arguments	232
Strings and Character Pointers	234
Arrays of Pointers	236
Command-Line Arguments	238
Access Through Pointers	240
Functions and Pointers	242
Example: Functions and Pointers	244
Labs	246

Chapter 11 - Structures	249
Structures	250
Comparison of Structures and Arrays	252
Structure Definitions	254
Structure Declarations	256
Structure Parameter Passing by Reference	258
Pointers to Structures	260
Structure Parameter Passing Again	262
Arrays of Structures	264
The malloc Routine	266
Labs	268
Appendix - File I/O in C	271
File Streams	272
Predefined Streams	274
The fprintf Function	276
The fscanf Function	278
fscanf() Examples	280
The fputs and fgets Functions	282
The fwrite and fread Functions	284
System I/O	286
Labs	288
Solutions	293
Index	345

CHAPTER 1 - COURSE INTRODUCTION

COURSE OBJECTIVES

- * Write C programs using all the major features of the language.

COURSE OVERVIEW

- * **Audience:** This course is for programmers who need to learn the C language. You will design and write many programs in this class.
- * **Prerequisites:** The ability to program in a high-level language such as a Pascal or COBOL is very helpful. Also, basic UNIX user-level skills are important.

SUGGESTED REFERENCES

Kelley, Al, and Ira Pohl. 1997. *A Book on C, Fourth Edition*. Addison-Wesley, Reading, MA. ISBN 0201183994.

Kernighan, Brian, and Dennis Ritchie. 1988. *The C Programming Language, Second Edition*. Prentice Hall, Englewood cliffs, NJ. ISBN 0131103628.

Kumar, Ram, and Rakesh Agrawal. 1999. *Programming in ANSI C*. Brooks/Cole, Pacific Grove, CA. ISBN 0314895639.

CHAPTER 2 - INTRODUCTION TO C

OBJECTIVES

- * Describe the purpose and use of C.
- * Present a brief history of C.
- * Discuss the main features of C.
- * Explain why people program in the C language.
- * Discuss the current status and probable future of C.

WHAT IS C ?

- * C is a general-purpose programming language.
- * Think of BASIC, FORTRAN, Pascal, COBOL, C.
- * You design and write C programs.
- * The computer compiles and runs them.

FEATURES OF C

- * C is small — few keywords.
- * C is structured.
- * C is modular — functions abound.
- * C is high level, like Pascal.
- * C is low level, like assembler.

WHY PROGRAM IN C ?

- * Very portable language.
- * High-level features, low-level machine control.
- * Structure and modularity are C strengths.
- * Supports small or extremely large programs.
- * C can be learned quickly.

HISTORY OF C

- * Designed by Dennis Ritchie of Bell Labs in 1972.
- * Created to write the UNIX operating system.
- * Predecessor to C was a language named B.
- * Many major UNIX applications are written in C.
- * Many serious PC applications are written in C (provided by Microsoft, Borland, and others).
- * Standardization endorsed by ANSI in 1989.

CURRENT STATUS AND FUTURE

- * An ANSI committee resolves questions about the C specification.
- * C and C++ are different languages.
- * C++ codes object-oriented designs.

CHAPTER 4 - DATA TYPES AND VARIABLES

OBJECTIVES

- * Define the basic C data types.
- * Select which types to use for your applications.
- * Write variable declarations for programs.
- * Choose appropriate variable names.
- * Use the different character constant notations.
- * Explain the machine character set.

FUNDAMENTAL DATA TYPES

* Variables and constants are the objects that programs manipulate.

* Each variable and constant is a specific type.

* C uses only three basic data types:

char A single character, one byte in size. An ASCII character fits in a **char** variable.

int An integer value, 16 or 32 bits on most machines.

float A real number, called floating point. Floating types can have fractional parts.

* We will learn more about variables as we use them in programs.

DATA TYPE VALUES AND SIZES

- * Different machines (computers) have different "word" sizes — 16 bit, 32 bit, etc.
- * The size affects the range of values of numeric variables.

Integers:

Range

16 bit (2 byte)

-32768 to 32767

32 bit (4 byte)

-2,147,483,648 to 2,147,483,647

Floating point:

32 bit (4 byte)

About 10^{38} to 10^{-38}
6 or 7 digits of precision

64 bit (8 byte)

About 10^{308} to 10^{-308}
Approximately 15 digits of precision

DATA TYPE VALUES AND SIZES

* Typical sizes of data types on modern machines.

Type	Size	Examples
char	1 byte	char answer; answer = 'Y';
int	4 bytes	int ccount; ccount = 5000;
short int	2 bytes	short little; little = -10;
long int	4 bytes	long great_one; great_one = 1000000;
float	4 bytes	float price; price = 3.98;
double	8 bytes	double distance; distance = 42e102;

VARIABLE DECLARATIONS

- * All variables must be declared to specify their name and type, and to allocate storage.

vardec.c

```
int main(void)
{
    int i=0;          /* Initialize variables */
    float minimum = -999.99;
    int x, y, z;     /* Several int's on a line */

    float bottom;   /* Using one line per allows */
    float top;      /* One comment per variable */
    float sidel1;
    float sidel2;

    float radius,   /* A "clean" way to declare */
           diameter,
           circumference,
           volume;

    char carray[25], c='n'; /* An array and a scalar */

} /* This IS a program */
```


VARIABLE NAMES

- * Variable names (identifiers) are a sequence of letters, digits, and underscores.
- * They must start with a letter or underscore.
- * Identifiers are case dependent.
- * Maximum length is implementation-defined.
- * ANSI C requires that at least the first 31 characters be recognized as significant.

Examples:

```
char    companyname[40], /* All different */
        company_name[40],
        CompanyName[40];

int Port1, port1, port_1;
```


CONSTANTS

- * Integer constants can be specified to be **long** by appending an **L**.

```
12345L
```

- * A leading zero on an **int** constant specifies octal.

```
033 /* equal to decimal 27 */
```

- * A leading **0X** or **0x** on an **int** constant specifies hexadecimal.

```
0x1b /* equal to decimal 27 */
```

- * Floating point constants are converted to type **double**.

```
47.25  
0.99  
.99  
2.65e-8 /* Exponential notation */  
6e23 /* "Six times ten to the 23rd" */
```


CHARACTER CONSTANTS

- * A character constant is a single character within single quotes.

```
char c;  
c = 'A';
```

- * The value of a character constant is the numeric value of the character in the machine's character set.

ASCII: **'A'** is 65
 'B' is 66
 '0' is 48
 '1' is 49

- * Numeric **char** values can be created with `\ddd`, where **ddd** is one to three octal digits.

```
c = '\014';  
last_name[i] = '\0';
```

- * Note that `'\0'` is equal to `0`.

STRING CONSTANTS

- * A string constant is a sequence of characters within double quotes.

```
char s[20];  
strcpy (s, "Check it out");
```

- * The compiler allocates the number of bytes inside the quotes, plus one for the null terminator.
- * **strcpy()** copies byte-by-byte until it has copied the null.

LABS

- ❶ Look up the C operator **sizeof()**. Write a program using **sizeof()** that prints the size in bytes of each of the following data types: **char**, **int**, **short**, **long**, **float**, and **double**.
(Solution: *sizeof.c*)
- ❷ Write a program that prompts for an integer, reads it as an ASCII string from the keyboard with the **gets()** function, converts the ASCII string to an integer with the **atoi()** function, then uses **printf()** to print the integer's value in decimal, octal, and hexadecimal. For decimal, hexadecimal, and octal conversion, **printf()** uses **%d**, **%x**, and **%o** respectively.
(Solution: *dechexoct.c*)
- ❸ Use a **for** loop to print out the capital letters. In your **printf()** statement, use the integer index variable from the loop as an argument to **printf()**, but use the **%c** specifier in the **printf()** format string. Refer to the ASCII character set table. (The decimal ASCII value for A is 65.)
(Solution: *allcaps.c*)

Expand your program to print out all printable ASCII characters.
(Solution: *allprint.c*)

- ❹ Write a program that makes use of **printf()** format specifiers to display the following exactly:

```
hi mom
  1
 1.0
```

Use a string, an integer, and a float. You must use width specifiers in your **printf()**.
(Solution: *himom.c*)

- ❺ Look at the program *signed.c*, then run it. Do you know how "two's complement" arithmetic works?

ascii - map of ASCII character set in decimal.

000 nul	001 soh	002 stx	003 etx	004 eot	005 enq	006 ack	007 bel
008 bs	009 ht	010 nl	011 vt	012 np	013 cr	014 so	015 si
016 dle	017 dc1	018 dc2	019 dc3	020 dc4	021 nak	022 syn	023 etb
024 can	025 em	026 sub	027 esc	028 fs	029 gs	030 rs	031 us
032 sp	033 !	034 "	035 #	036 \$	037 %	038 &	039 '
040 (041)	042 *	043 +	044 ,	045 -	046 .	047 /
048 0	049 1	050 2	051 3	052 4	053 5	054 6	055 7
056 8	057 9	058 :	059 ;	060 <	061 =	062 >	063 ?
064 @	065 A	066 B	067 C	068 D	069 E	070 F	071 G
072 H	073 I	074 J	075 K	076 L	077 M	078 N	079 O
080 P	081 Q	082 R	083 S	084 T	085 U	086 V	087 W
088 X	089 Y	090 Z	091 [092 \	093]	094 ^	095 _
096 `	097 a	098 b	099 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124	125 }	126 ~	127 del

CHAPTER 10 - ADVANCED POINTERS

OBJECTIVES

- * Initialize pointers at definition time.
- * Create and use arrays of pointers.
- * Pass command-line arguments to your program.
- * Use pointers to functions.
- * Decipher complex C declarations.

POINTER INITIALIZATION

- * Uninitialized or unassigned pointers contain garbage, i.e., they don't point to anything.
- * They can be initialized at definition or assigned at runtime.

```
float Prices[50], *PricePtr=Prices;
```

is equivalent to

```
float Prices[50], *PricePtr;  
...  
PricePtr=Prices;
```


COMMAND-LINE ARGUMENTS

- * Arguments can be passed to a C program at runtime.
- * They are stored in an array of pointers to character strings.
- * Let's review character pointers and strings (arrays of characters) first . . .

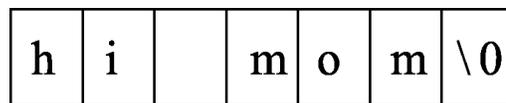
STRINGS AND CHARACTER POINTERS

- * Recall that a *string* is an array of characters, null terminated.

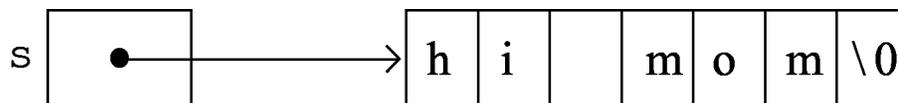
The fragment:

```
char s[7];  
...  
strcpy(s, "hi mom");
```

allocates and writes into memory as follows:



- * Thus, *s* is a pointer to the first of several consecutive characters.

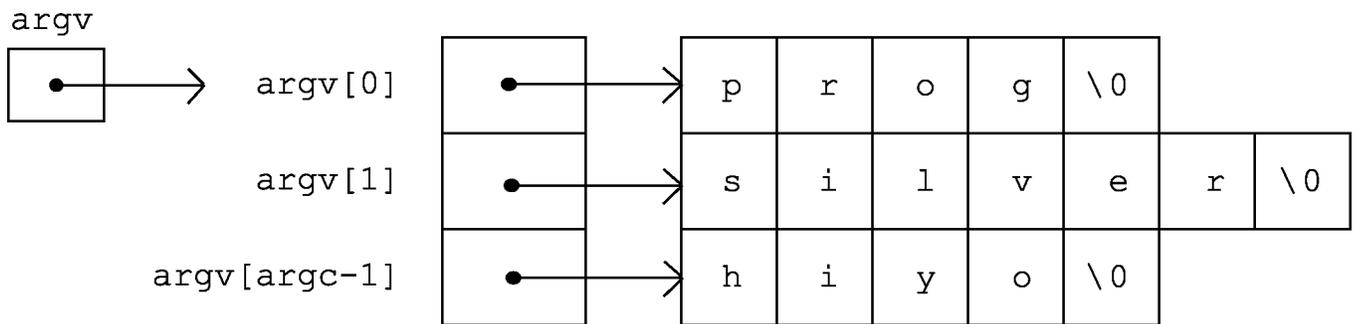


ARRAYS OF POINTERS

- * Since a *string* is an array, it is a *pointer* to the first of several consecutive characters.
- * Arrays can contain *pointers*, just as arrays can contain **ints**, **floats**, or any other type.
- * Thus, an array can contain pointers to characters, i.e., it can contain pointers to the first of several consecutive characters.

Execute a program as:

```
$ prog silver hiyo
```



argc will be set to 3.

COMMAND-LINE ARGUMENTS

cmdargs.c

```
#include <stdio.h>

int main(int argc, char *argv[])
/*  argc:  Count of cmd-line args
 *  argv[]: Array of pointers to argument strings
 */
{

    int i;

    i = 0;
    do {
        printf("Argument %d: %s\n", i, argv[i]);
    } while (++i < argc);
    return 0;
} /* end of main */
```

* **argv** is an *array of pointers to char*.

ACCESS THROUGH POINTERS

cmdargs2.c

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

int main(int argc, char *argv[])
/* int argc:   Count of cmd-line args   */
/* char *argv[]: Array of pointers to argument strings */
/*
{
    char *name;          /* Local pointer - uninitialized */
    int age, subjectsAge = 55;

    if (argc != 3) {
        printf("Usage: %s name age\n", argv[0]);
        exit(1);
    }

    argv++;                /* Get past program name */
    name = *argv++;        /* Get arg1, increment argv */
    age = atoi(*argv);      /* Get arg2, convert to int */

    if (age <= subjectsAge)
        printf("You're lying, you aren't really %d!", age);
    else
        printf("%s, you are %d years young!\n", name, age);

    return 0;
} /* end of main */
```

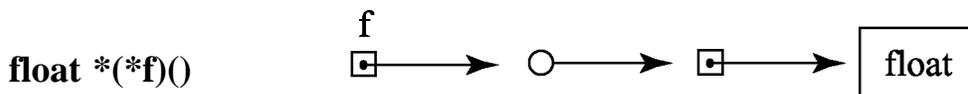
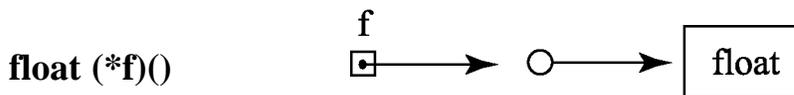
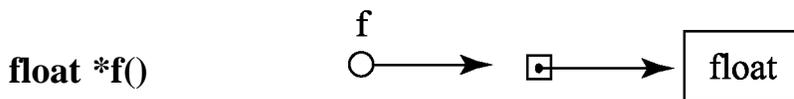
- * The value of ***argv++** is the pointer that **argv** pointed to before **argv** was incremented.

FUNCTIONS AND POINTERS

* Study these:

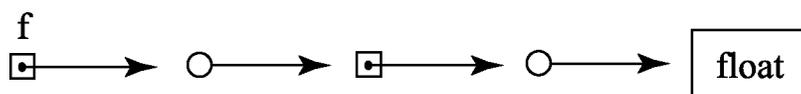
- **float f();** **f** is a function returning **float**.
- **float *f();** **f** is a function returning a pointer to a **float**.
- **float (*f)();** **f** is a pointer to a function returning a **float**.
- **float *(*f)();** **f** is a pointer to a function returning a pointer to a **float**.

Key: ○ → means "function returning"



and

float ((*f)())(); — a pointer to a function, returning a pointer to a function returning a **float**



EXAMPLE: FUNCTIONS AND POINTERS

funcptr.c

```
/* program to demonstrate passing a 'ptr to a function' to a
   function by assigning the ptr to a ptr variable. */
void pcaller(int (*f)(int), int i);
int sub(int j);

main()
{
    int (*pvar)(int); /* ptr variable */

    printf("main: assign sub to pvar\n");
    pvar = sub;
    printf("main: calling pcaller\n");
    /* call pcaller with aux. function pointer */
    pcaller(pvar, -200);
    printf("main: end\n");
}

void pcaller(int (*f)(int), int i)
{
    printf(" pcaller: i= %d\n", i);
    f(i);
    printf(" pcaller: i= %d\n", i);
}

int sub(int j)
{
    printf(" sub: j= %d\n", j);
    j++;
    printf(" sub: j++ = %d\n", j);
}
```

Output:

```
main: assign sub to pvar
main: calling pcaller
 pcaller: i= -200
 sub: j= -200
 sub: j++ = -199
 pcaller: i= -200
main: end
```


LABS

- 1 Write a program to print out its command-line arguments in reverse.
(Solution: *args.c*)
- 2 Write a program that reads up to 100 lines of text from the standard input into an array of character strings (You may use a previous exercise as a starting point). Print the shortest and longest strings and calculate the average string length in the file. You will need to define something like this:

```
char text[100][80];
```

This array of character strings is an array of 100 strings, each of which can be up to 79 (with terminating NULL) bytes.

The line

```
gets(text[i]);
```

will read a line from the standard input into the *i*'th string held in text.

Note: To save typing, you can redirect a file into standard input on the command line:

```
strstats <input.txt
```

(Solution: *strstats.c*)

