

# Portability and the C Language

Rex Jaeschke

Portability and the C Language

© 1986–1988, 2009 Rex Jaeschke.

Edition: 2.0

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means whatsoever, except in the case of brief quotations embodied in critical reviews and articles.

The information in this book is subject to change without notice, and should not be construed as a commitment by the author or the publisher. Although every precaution has been taken in the preparation of this book, the author and the publisher assume no responsibility for errors or omissions.

BSD is a trademark of University of California, Berkeley.

COS is a trademark of Cray Research.

DEC, VAX, VAX/VMS, Ultrix, RSTS and RSX are trademarks of Digital Equipment Corporation. *[Digital Equipment Corporation has since been purchased by Compaq, which was then merged with Hewlett-Packard.]*

GCOS is a trademark of Honeywell.

Lattice C is a trademark of Lattice. *[Lattice has since been purchased by SAS Institute.]*

MS-DOS and XENIX are trademarks of Microsoft.

PC-DOS, MVS, and VM/CMS are trademarks of IBM.

POSIX is a trademark of IEEE. *[This mark is now owned by The Open Group.]*

UNIX is a trademark of AT&T. *[This mark is now owned by The Open Group.]*

Please address comments, corrections, and questions to the author:

Rex Jaeschke

2051 Swans Neck Way  
Reston, VA 20191-4023

+1 (703) 860-0091

[www.RexJaeschke.com](http://www.RexJaeschke.com)

[rex@RexJaeschke.com](mailto:rex@RexJaeschke.com)

Preface .....	xiii
Reader Assumptions and Advice .....	xv
<b>1. Introduction.....</b>	<b>1</b>
1.1 Defining Portability .....	1
1.2 Portability is Not New .....	2
1.3 Who Needs Portability?.....	3
1.4 The Economics of Portability .....	3
1.5 Measuring Portability .....	5
1.6 The Porting Toolbox.....	6
1.7 Environmental Issues.....	7
1.8 Programmer Portability .....	8
<b>2. The Environment.....</b>	<b>10</b>
2.1 Conceptual Models .....	10
2.1.1 Translation Environment .....	10
2.1.2 Execution Environments.....	12
2.2 Environmental Considerations .....	18
2.2.1 Character Sets.....	18
2.2.2 Character Display Semantics .....	21
2.2.3 Signals and Interrupts.....	22
2.2.4 Environmental Limits.....	22
<b>3. Lexical Elements.....</b>	<b>26</b>
3.1 Tokens.....	26
3.2 Keywords .....	28
3.3 Identifiers.....	30
3.3.1 Scopes of Identifiers .....	34
3.3.2 Linkages of Identifiers.....	36
3.3.3 Name Spaces of Identifiers .....	39
3.3.4 Storage Durations of Objects .....	40
3.3.5 Identifier Types.....	43
3.4 Constants .....	48
3.4.1 Floating Constants .....	48
3.4.2 Integer Constants .....	49
3.4.3 Enumeration Constants .....	52
3.4.4 Character Constants .....	52
3.5 String Literals .....	54
3.6 Operators.....	55
3.7 Punctuators.....	56
3.8 Header Names .....	56
3.9 Comments.....	56
<b>4. Conversions .....</b>	<b>58</b>
4.1 Arithmetic Operands .....	58
4.1.1 Characters and Integers.....	58
4.1.2 Floating and Integral.....	60
4.2 Other Operands .....	62
4.2.1 Pointers.....	62

4.2.2	Structures and Unions .....	65
<b>5.</b>	<b>Expressions .....</b>	<b>68</b>
5.1	General Information .....	68
5.2	Primary Expressions.....	71
5.3	Postfix Operators .....	71
5.3.1	Array Subscripting .....	71
5.3.2	Function Calls.....	75
5.3.3	Structure and Union Members.....	78
5.3.4	Postfix Increment and Decrement Operators .....	80
5.4	Unary Operators .....	80
5.4.1	Prefix Increment and Decrement Operators.....	80
5.4.2	Address and Indirection Operators .....	80
5.4.3	Unary Arithmetic Operators .....	83
5.4.4	The sizeof Operator .....	84
5.5	Cast Operators .....	86
5.6	Multiplicative Operators.....	89
5.7	Additive Operators .....	89
5.8	Bitwise Shift Operators.....	90
5.9	Relational Operators.....	90
5.10	Equality Operators.....	91
5.11	Bitwise AND Operator.....	91
5.12	Bitwise Exclusive OR Operator.....	91
5.13	Bitwise Inclusive OR Operator .....	91
5.14	Logical AND Operator .....	92
5.15	Logical OR Operator.....	92
5.16	Conditional Operator .....	92
5.17	Assignment Operators .....	93
5.17.1	Simple Assignment .....	93
5.17.2	Compound Assignment .....	93
5.18	Comma Operator .....	94
5.19	Constant Expressions.....	94
<b>6.</b>	<b>Declarations.....</b>	<b>96</b>
6.1	Empty Declarations.....	96
6.2	Storage-Class Specifiers .....	97
6.2.1	Position of Class Keyword.....	97
6.2.2	The register Class.....	98
6.2.3	The auto Class .....	100
6.2.4	The static Class.....	103
6.3	Type Specifiers.....	104
6.4	Structure and Union Specifiers .....	106
6.5	Enumeration Specifiers.....	112
6.6	Tags.....	113
6.7	Type Qualifiers.....	114
6.8	Declarators.....	116
6.8.1	General Information .....	116
6.8.2	Function Declarators .....	117
6.9	Type Definitions and Type Equivalence .....	119
6.10	Initialization .....	121

6.11 External Definitions .....	124
6.11.1 Function Definitions .....	124
6.11.2 External Object Definitions.....	126
<b>7. Statements .....</b>	<b>128</b>
7.1 Labeled Statements .....	128
7.2 Compound Statement, or Block .....	129
7.3 Expression and Null Statements .....	131
7.4 Selection Statements .....	132
7.4.1 The <code>switch</code> Statement .....	133
7.5 Iteration Statements.....	134
7.5.1 The <code>do</code> Statement.....	135
7.5.2 The <code>for</code> Statement.....	135
7.6 Jump Statements .....	135
7.6.1 The <code>goto</code> Statement .....	135
7.6.2 The <code>return</code> Statement .....	136
7.7 Extensions.....	138
<b>8. The Preprocessor .....</b>	<b>139</b>
8.1 General Information .....	139
8.1.1 Preprocessor versus Compiler .....	139
8.1.2 The Directive Name Format .....	139
8.1.3 Start Position of Directives .....	140
8.1.4 White Space Within Directives .....	141
8.1.5 Directive Continuation Lines .....	142
8.1.6 Trailing Tokens.....	142
8.1.7 Comments in Directives.....	143
8.1.8 Phases of Translation.....	143
8.1.9 Inspecting Preprocessor Output.....	145
8.2 Source File Inclusion .....	145
8.2.1 <code>#include</code> Directive Format .....	146
8.2.2 Header Names .....	147
8.2.3 Nested Headers .....	149
8.2.4 <code>#include</code> Path Specifiers.....	150
8.2.5 Modification of Standard Headers .....	150
8.3 Macro Replacement .....	151
8.3.1 Macros with Arguments .....	152
8.3.2 Rescanning Macro Names .....	153
8.3.3 Replacement Within Strings and Character Constants .....	153
8.3.4 Command-Line Macro Definition .....	154
8.3.5 Benign Redefinition .....	155
8.3.6 Predefined Macros .....	156
8.3.7 Macro Definition Symbol Table .....	157
8.3.8 Stacking Macro Definitions.....	157
8.3.9 The <code>#</code> Stringize Operator .....	158
8.3.10 The <code>##</code> Token-Pasting Operator .....	158
8.3.11 Redefining Keywords .....	159
8.4 The <code>#undef</code> Directive.....	159
8.5 Conditional Inclusion .....	160

## Portability and the C Language

8.5.1	#if Arithmetic .....	160
8.5.2	The defined Operator .....	162
8.5.3	The #elif Directive .....	163
8.5.4	Nested Conditional Directives .....	164
8.6	Line Control .....	164
8.7	The Null Directive .....	164
8.8	The #pragma Directive .....	165
8.9	The #error Directive .....	165
8.10	Non-Standard Directives .....	165
<b>9.</b>	<b>Library Introduction .....</b>	<b>166</b>
9.1	Definition of Terms .....	166
9.2	The Standard Headers .....	166
9.3	Use of Library Functions .....	168
9.4	Non-Standard Headers .....	169
<b>10.</b>	<b>assert.h – Diagnostics .....</b>	<b>170</b>
10.1	Program Diagnostics .....	170
10.1.1	The assert Macro .....	170
<b>11.</b>	<b>ctype.h – Character Handling .....</b>	<b>172</b>
11.1	Character Testing Functions .....	173
11.1.1	The isalnum Function .....	174
11.1.2	The isalpha Function .....	174
11.1.3	The isascii Function .....	174
11.1.4	The iscntrl Function .....	175
11.1.5	The iscsym Function .....	175
11.1.6	The iscsymf Function .....	175
11.1.7	The isdigit Function .....	176
11.1.8	The isgraph Function .....	176
11.1.9	The islower Function .....	176
11.1.10	The isodigit Function .....	177
11.1.11	The isprint Function .....	177
11.1.12	The ispunct Function .....	177
11.1.13	The isspace Function .....	177
11.1.14	The isupper Function .....	178
11.1.15	The isxdigit Function .....	178
11.2	Character Case Mapping Functions .....	178
11.2.1	The toascii Function .....	179
11.2.2	The tolower Function .....	179
11.2.3	The _tolower Function .....	179
11.2.4	The toupper Function .....	180
11.2.5	The _toupper Function .....	180
<b>12.</b>	<b>errno.h – Errors .....</b>	<b>181</b>
12.1	The errno Macro .....	181
12.2	The errno Value Macros .....	183
<b>13.</b>	<b>float.h – Numerical Limits .....</b>	<b>185</b>
<b>14.</b>	<b>limits.h – Numerical Limits .....</b>	<b>188</b>

<b>15. locale.h – Localization .....</b>	<b>191</b>
15.1 Locale Control .....	192
15.1.1 The setlocale Function .....	192
15.2 Numeric Formatting Convention Inquiry.....	193
15.2.1 The localeconv Function .....	193
<b>16. math.h – Mathematics .....</b>	<b>194</b>
16.1 Treatment of Error Conditions.....	194
16.2 Trigonometric Functions.....	195
16.2.1 The acos Function .....	195
16.2.2 The asin Function .....	195
16.2.3 The atan Function .....	195
16.2.4 The atan2 Function .....	196
16.2.5 The cos Function .....	196
16.2.6 The sin Function .....	196
16.2.7 The tan Function .....	196
16.3 Hyperbolic Functions .....	197
16.3.1 The cosh Function .....	197
16.3.2 The sinh Function .....	197
16.3.3 The tanh Function .....	197
16.4 Exponential and Logarithmic Functions.....	197
16.4.1 The exp Function .....	197
16.4.2 The frexp Function .....	198
16.4.3 The ldexp Function .....	198
16.4.4 The log Function .....	198
16.4.5 The log10 Function .....	199
16.4.6 The modf Function .....	199
16.5 Power Functions .....	199
16.5.1 The pow Function .....	199
16.5.2 The sqrt Function .....	200
16.6 Nearest Integer, Absolute and Remainder .....	200
16.6.1 The ceil Function .....	200
16.6.2 The fabs Function .....	200
16.6.3 The floor Function .....	201
16.6.4 The fmod Function .....	201
<b>17. setjmp.h – Non-Local Jumps .....</b>	<b>202</b>
17.1 Saving the Calling Environment.....	202
17.1.1 The setjmp Macro.....	202
17.2 Restore Calling Environment .....	203
17.2.1 The longjmp Function.....	203
<b>18. signal.h – Signal Handling .....</b>	<b>206</b>
18.1 Types and Macros.....	206
18.1.1 The Type sig_atomic_t.....	206
18.1.2 Function Pointer Macros SIG_* .....	206
18.1.3 Signal Type Macros SIG* .....	207
18.2 Specify Signal Handling.....	208
18.2.1 The signal Function.....	208
18.3 Send Signal.....	210

18.3.1	The raise Function .....	210
18.4	An Example .....	210
18.5	Non-ANSI Issues.....	211
<b>19.</b>	<b>stdarg.h – Variable Arguments .....</b>	<b>212</b>
19.1	The va_start Macro .....	212
19.2	The va_arg Macro.....	213
19.3	The va_end Macro.....	213
19.4	Examples.....	213
<b>20.</b>	<b>stddef.h – Common definitions.....</b>	<b>220</b>
20.1	The ptrdiff_t Type.....	220
20.2	The size_t Type .....	220
20.3	The wchar_t Type .....	221
20.4	The NULL Macro .....	221
20.5	The offsetof Macro .....	222
<b>21.</b>	<b>stdio.h – Input/Output .....</b>	<b>223</b>
21.1	File Systems .....	223
21.1.1	Defined Types .....	224
21.1.2	Defined Macros .....	224
21.2	Common Extensions .....	225
21.3	Operations on Files .....	225
21.3.1	The remove Function.....	225
21.3.2	The rename Function.....	226
21.3.3	The tmpfile Function.....	227
21.3.4	The tmpnam Function.....	227
21.4	File Access Functions .....	228
21.4.1	The fclose Function.....	228
21.4.2	The fflush Function.....	229
21.4.3	The fopen Function .....	230
21.4.4	The freopen Function.....	231
21.4.5	The setbuf Function.....	231
21.4.6	The setvbuf Function.....	232
21.5	Formatted Input/Output Functions.....	233
21.5.1	The fprintf Function.....	233
21.5.2	The fscanf Function.....	233
21.5.3	The printf Function.....	234
21.5.4	The scanf Function .....	236
21.5.5	The sprintf Function.....	237
21.5.6	The sscanf Function.....	238
21.5.7	The vfprintf Function.....	238
21.5.8	The vprintf Function.....	239
21.5.9	The vsprintf Function.....	239
21.6	Character Input/Output Functions .....	240
21.6.1	The fgetc Function .....	240
21.6.2	The fgets Function .....	240
21.6.3	The fputc Function .....	240
21.6.4	The fputs Function .....	241
21.6.5	The getc Function .....	241
21.6.6	The getchar Function.....	241

21.6.7	The gets Function .....	242
21.6.8	The putc Function .....	242
21.6.9	The putchar Function.....	242
21.6.10	The puts Function .....	243
21.6.11	The ungetc Function.....	243
21.7	Direct Input/Output Functions .....	244
21.7.1	The fread Function .....	244
21.7.2	The fwrite Function.....	244
21.8	File Positioning Functions .....	245
21.8.1	The fgetpos Function.....	245
21.8.2	The fseek Function .....	245
21.8.3	The fsetpos Function.....	246
21.8.4	The ftell Function .....	247
21.8.5	The rewind Function.....	247
21.9	Error-Handling Functions.....	247
21.9.1	The clearerr Function.....	247
21.9.2	The feof Function .....	248
21.9.3	The ferror Function.....	248
21.9.4	The perror Function.....	248
<b>22.</b>	<b>stdlib.h – General Utilities .....</b>	<b>250</b>
22.1	String Conversion Functions .....	250
22.1.1	The atof Function .....	251
22.1.2	The atoi Function .....	251
22.1.3	The atol Function .....	251
22.1.4	The strtod Function.....	252
22.1.5	The strtol Function.....	252
22.1.6	The strtoul Function.....	253
22.2	Pseudo-Random Sequence Generation Functions .....	253
22.2.1	The rand Function .....	253
22.2.2	The srand Function .....	254
22.3	Memory Management Functions .....	254
22.3.1	The calloc Function.....	255
22.3.2	The free Function .....	256
22.3.3	The malloc Function.....	256
22.3.4	The realloc Function.....	256
22.4	Communication with the Environment .....	257
22.4.1	The abort Function .....	257
22.4.2	The atexit Function.....	258
22.4.3	The exit Function .....	258
22.4.4	The getenv Function.....	259
22.4.5	The system Function.....	260
22.5	Searching and Sorting Utilities.....	260
22.5.1	The bsearch Function.....	260
22.5.2	The qsort Function .....	261
22.6	Integer Arithmetic Functions .....	261
22.6.1	The abs Function .....	261
22.6.2	The div Function .....	262
22.6.3	The labs Function .....	262

## Portability and the C Language

22.6.4	The ldiv Function .....	262
22.7	Multibyte Character Functions .....	263
22.7.1	The mblen Function .....	263
22.7.2	The mbtowc Function .....	263
22.7.3	The wctomb Function .....	263
22.8	Multibyte String Functions .....	264
22.8.1	The mbstowcs Function.....	264
22.8.2	The wcstombs Function.....	264
<b>23.</b>	<b>string.h – String Handling .....</b>	<b>265</b>
23.1	Copying Functions.....	265
23.1.1	The memcpy Function .....	265
23.1.2	The memmove Function.....	266
23.1.3	The strcpy Function.....	266
23.1.4	The strncpy Function.....	266
23.2	Concatenation Functions.....	267
23.2.1	The strcat Function.....	267
23.2.2	The strncat Function.....	267
23.3	Comparison Functions .....	267
23.3.1	The memcmp Function.....	268
23.3.2	The strcmp Function.....	269
23.3.3	The strcoll Function.....	269
23.3.4	The strncmp Function.....	270
23.3.5	The strxfrm Function.....	270
23.4	Search Functions.....	271
23.4.1	The memchr Function .....	271
23.4.2	The strchr Function.....	271
23.4.3	The strcspn Function.....	271
23.4.4	The strpbrk Function.....	272
23.4.5	The strrchr Function.....	272
23.4.6	The strspn Function.....	273
23.4.7	The strstr Function.....	274
23.4.8	The strtok Function.....	274
23.5	Miscellaneous Functions .....	275
23.5.1	The memset Function.....	275
23.5.2	The strerror Function.....	275
23.5.3	The strlen Function.....	276
23.6	Non-Standard Functions.....	276
<b>24.</b>	<b>time.h – Date and Time .....</b>	<b>277</b>
24.1	Components of Time .....	277
24.2	Time Manipulation Functions.....	277
24.2.1	The clock Function .....	277
24.2.2	The difftime Function.....	278
24.2.3	The mktime Function.....	278
24.2.4	The time Function .....	279
24.3	Time Conversion Functions.....	279
24.3.1	The asctime Function.....	279
24.3.2	The ctime Function .....	280
24.3.3	The gmtime Function.....	280

24.3.4	The <code>localtime</code> Function .....	280
24.3.5	The <code>strftime</code> Function.....	281
24.4	Miscellaneous Functions .....	281
<b>Annex A. Keywords and Reserved Identifiers.....</b>		<b>282</b>
A.1	C Keywords .....	282
A.2	C++ Keywords .....	282
A.3	The ANSI Standard Headers.....	282
A.4	Identifiers Alphabetically by Header .....	283
A.5	Identifiers in Alphabetical Order .....	286
A.6	Identifiers Reserved for Future Use.....	297
<b>Annex B. Recommended Reading and References .....</b>		<b>298</b>



# Preface

Early in 1986, I was invited to teach a 3-day seminar on portability as it pertained to the C language. The seminar was to be offered in several major cities around the U.S. As it happened, the series was cancelled, but by then I had put together a 70-page manuscript intended for use as class handouts.

Ever since I came to the C fold, I have been fascinated by the apparent contradiction of C being both a low-level systems implementation language yet, somehow, also being a portable one. And every time I heard someone speak, or write, enthusiastically about C's "inherent" portability, I became more uneasy with the fact that either I or a significant part of the C community was missing some major part of the "C picture." As it happens, I don't think it's me although it does seem that a surprising amount of well-written C code can be ported relatively easily.

Driven by the fact that I had a base portability document and an acute interest in the C phenomenon generally, and in the ANSI C Standard and portability in particular, I embarked on a formal and detailed look at C and portability. The fact that I also make a substantial living from consulting in C and teaching introductory and advanced seminars about it added more weight to my decision to develop a serious manuscript for a 3-day portability seminar. Along the way, I decided the end result was worthy of becoming a book, and as you can see, Howard W. Sams agreed.

At first, I expected to produce about 200 book pages. Then it became 300 and 400, and finally settled on 425, but only after I decided to cut quite a few appendices, purely for space reasons. In fact, since the amount and utility of the material left "on the editing room floor" is substantial, I am looking at ways to distribute that as well, perhaps through a future revision, or a companion volume. In any case, this book does not contain *all* my findings.

This book attempts to document C-specific issues you may encounter when porting existing code, or when writing code that is to be ported to numerous target environments. I use the term *attempt* since I don't believe this book provides all the answers, and in many cases, it does not even pretend to do so. For example, if you are porting from one flavor of UNIX to another, this book does not discuss all (or in fact any of) the dark corners of that operating system. Nonetheless, I do believe it to be a credible beginning on which future works can be based. It is, as far as I can tell, the first widely published work of more than 20–30 pages that specifically addresses portability as it pertains to C. Since I do not claim to be well versed in more than 3–4 operating system and hardware environments, I have overlooked some relevant issues. Alternately, I may have overindulged in various esoteric aspects that may occur only in theory.

Whatever your interest in portability is, I hope this book provides some food for thought, even if only to help convince you that portability is not for you. If that is all this book achieves, it will have been wildly successful. If, on the other hand, it helps you define a port strategy, or saves you going down a few wrong roads, then, too, I am happy. Whatever your opinion of this text, let me know since only by getting constructive criticism, outside input, and more personal experience can I improve it in future revisions or in a companion volume.

Anyone who has ever written a lengthy document that is to be read by more than a few people knows that after the first 2–3 reads, you no longer actually read what is written. You simply read what should be there. Therefore,  
© 1986–1988, 2009 Rex Jaeschke.

## Portability and the C Language

you need technically competent reviewers who can provide constructive criticism. In this regard, the following people made significant contributions by proofing all or major parts of the manuscript: Steve Bartels, Don Bixler, Don Courtney, Dennis Deloria, John Hausman, Bryan Higgs, Gary Jeter, Tom MacDonald, and Sue Meloy. While I implemented many of their suggestions, space and time constraints prohibited me from capitalizing fully on their organizational and other suggestions. But as software vendors say, “We have to leave something to put in the next release.”

Others who have had more than a passing influence on my relatively short, but intense, C career are: P.J. Plauger, ANSI C Secretary, ISO C Convener and President of Whitesmiths Ltd, an international vendor of C and Pascal development tools; Tom Plum, ANSI C Vice-Chair, Chairman of Plum Hall, and leading C author; Larry Rosler, formerly the Editor of the Draft ANSI C Document, and AT&T's principal member on the ANSI C Committee (now of Hewlett-Packard); and Jim Brodie, an independent consultant (formerly of Motorola) who convened the ANSI C Standards Committee in mid-1983, and has so ably chaired it to its (hopefully) successful completion in late 1988 or thereabouts. Also, to my colleagues on the ANSI C X3J11 Standards Committee, I say thanks for the opportunity to work with you all—without your papers, presentations, and sometimes volatile (pun intended) discussions both in and out of committee, the quality and quantity of material in this book would have been significantly reduced, perhaps to the point of not being sufficient enough for publication.

*Rex Jaeschke*

### **Notes about the Second Edition:**

This electronic edition is intended to be a faithful reproduction of the original paper version; however, some typographic changes were made because of a change in typesetting systems. Other changes made include the following:

- Minor punctuation and grammar corrections and improvements
- Omission of the index
- Omission of Annex (Appendix) C, “Portability Software Suite”
- Ownership of some trademarks has changed

The paper version was published more than 20 years ago, so keep the following in mind when reading this new edition:

- Committee names have changed. X3J11 became J11, and then INCITS/PL22.11.
- The ANSI C standard was approved in 1989, and became an ISO Standard (ISO/IEC 9899) in 1990. A substantial amendment was published in 1985, and a revised version of the standard was published in 1999.
- The C++ language and library has been standardized.
- At the time of writing, 16-bit systems were the norm, and 64-bit systems were rare and “exotic”.
- The publications mentioned in Annex B, “Recommended Reading and References”, might no longer be available.

# Reader Assumptions and Advice

This book does not attempt to teach introductory, or even advanced, C constructs. (Nor is it a tutorial on ANSI C.) In fact, at times some paragraphs seem as terse as C itself. Yet while I have attempted to soften such passages, I make no apologies for those that remain. Portability is not something a first time or trainee C programmer embarks on—quite the opposite. Portability in C requires more than a passing background in C, and years of experience are not necessarily a good measure of C expertise. Many programmers with 5–7 years of C experience often know far less about the language than those with 1–2 years. A lot depends on your background and attitude.

The text is aimed specifically at those aspects directly related to porting a C language routine. However, it does not provide a “recipe” for successfully porting a system in any given set of target environments—it merely details many of the problems and situations you may encounter or may need to investigate. The book presumes that you are familiar with the basic constructs of the C language, such as all the operators, statements, and preprocessor directives and that you are fluent with data and function pointers and interfacing with the standard run-time library.

The organization of the text merits some comments. Chapter 1 attempts to get you into a portability mindset, addresses some of the misconceptions about C and portability, and helps put portability, as a discipline, in perspective. Chapters 2–8 cover language and preprocessor issues and closely follows the order of sections in the ANSI C Language Standard. In fact, almost all of the chapter, section, and subsection headings are taken directly from that document and are presented in the same order. Chapters 9–24 cover the standard run-time library. While this is one section in the Standard and Rationale, in this book, each header has its own chapter. The Appendices provide some useful information, particularly in Appendix B where a detailed list of related reading material is provided, as is information about obtaining a copy of the ANSI C Standard.

You may notice a small amount of duplication. This is intentional since I decided to organize it that way rather than to have you chase down critical and related information via forward references.

Note also, that I have not always used the “new style” of declaring and defining functions, or other ANSI “goodies.” This is not an oversight, but, rather, a deliberate choice. The reason for this is simple. Since there is not yet an ANSI C Standard, no implementation yet conforms, and many C implementations support few (if any) of the additions and/or changes introduced by that Standard. Also, many people porting code may be doing so from one non-ANSI compiler to another.

Since the Standard, its accompanying Rationale document, and this text have the same basic organization, having a copy of each is advantageous, although not completely necessary, since the Standard can be “heavy going.” However, the Rationale is much more leisurely paced and readable by the non-linguist. Note though that, having participated in the deliberations of the ANSI committee for four years my vocabulary reflects that of the C Standard. Therefore, a copy of that document will prove very useful.

## Portability and the C Language

Just for the record, all references (unless otherwise stated) to K&R refer to the first edition (1978) of Kernighan and Ritchie's book, since, at the time of writing, that was *the* K&R. Also, all references to ANSI C are actually references to the *Draft ANSI C Standard* dated May 13<sup>th</sup> 1988. This is the version of the Standard made available for the third public comment period. It is expected there will be little (if any) difference between this document and the final Standard.

Numerous references to acronyms, abbreviations, and specialized terms are made throughout the book. Most are in common use in the C industry today; however, some are defined here (their definitions taken verbatim from the ANSI C Standard) in case they are new to you:

- *Unspecified behavior* – Behavior, for a correct program construct and correct data, for which the Standard imposes no requirements.
- *Undefined behavior* – Behavior, upon use of a nonportable or erroneous program construct, of erroneous data, or of indeterminately valued objects, for which the Standard imposes no requirements. Permissible undefined behavior ranges from ignoring the situation completely with unpredictable results to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).  
If a “shall” or “shall not” requirement that appears outside of a constraint is violated, the behavior is undefined. Undefined behavior is otherwise indicated in the Standard by the words “undefined behavior” or by the omission of any explicit definition of behavior. There is no difference in emphasis among these three; they all describe “behavior that is undefined.”
- *Implementation-defined behavior* – Behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation and that each implementation shall document.
- *Locale-specific behavior* – Behavior that depends on local conventions of nationality, culture, and language that each implementation shall document.
- *Ivalue* – An expression (with an object type or an incomplete type other than `void`) that designates an object. When an object is said to have a particular type, the type is specified by the lvalue used to designate the object. (The name “lvalue” comes originally from the assignment expression  $E1 = E2$ , in which the left operand  $E1$  must be a (modifiable) lvalue. It is perhaps better considered as representing an object “locator value.” What is sometimes called “rvalue” is in the Standard described as the “value of an expression.” An obvious example of an lvalue is an identifier of an object. As a further example, if  $E$  is a unary expression that is a pointer to an object,  $*E$  is an lvalue that designates the object to which  $E$  points.)
- *Modifiable lvalue* – An lvalue that does not have array type, does not have an incomplete type, does not have a `const`-qualified type, and, if it is a structure or union, does not have any member (including, recursively, any member of all contained structures or unions) with a `const`-qualified type.
- *Multibyte character* – A sequence of one or more bytes representing a single character in the extended character set of either the source or the execution environment. The extended character set is a superset of the basic character set.
- *Compiler* – This term is usually used to mean a C language translator and includes such tools as interpreters and incremental compilers as well. When some aspect is peculiar to one or more of these specific tools, this is so stated.

- *ANSI* – This organization is the American National Standards Institute. It is the focal point for standards for numerous industries including those related to data processing hardware and software. The ANSI X3 Secretariat oversees Data Processing with the X3J11 Committee being designated as the group standardizing the C language. X3J11 is the committee and ANSI C is their product.
- *POSIX* – Another ANSI standards group is IEEE, the Institute for Electrical and Electronic Engineers. One of their committees, P1003, is chartered with forming a standard for a portable operating system definition based on UNIX. This system is known as POSIX.
- *SVID* – The base documents used by X3J11 and P1003 came from AT&T who has developed its own *de facto* Standard. This standard is known as the System V Interface Definition (SVID).
- *X/OPEN* – This is a hardware and software consortium involved in operating system, language, and applications development tool standardization. It is building on existing *de facto* and official standards.

The ANSI C Standard contains a more complete list of definitions and, in particular, discusses the criteria for conformance of programs and implementations. It also introduces the term *obsolescent*, which identifies certain existing practices as “frowned upon” now that a (presumably) better alternative has been made available.

The various standards and documents mentioned above are discussed further in Appendix B.



# 1. Introduction

## 1.1 Defining Portability

How many times have you heard the statement “C is a portable language”? Given the large number of meanings, such a claim might imply the statement is, at best, vacuous. That is, for such a claim to make sense, one must define what they mean by portability.

According to *The Prentice-Hall Standard Glossary of Computer Terminology* by Robert A. Edmunds, portability is defined as follows. “*Portability*: A term related to compatibility. Portability determines the degree to which a program or other software can be moved from one computer system to another.” The key phrase here is “the degree to which a program can be moved.”

We can talk about portability from two points of view: generic and specific. Generically, portability simply means running a program in a host environment that is somehow different from the one for which it was designed. Since the cost of producing and maintaining software far outweighs that for producing hardware, we have a huge incentive to increase the shelf life of our software beyond the current incarnations of hardware. It simply makes economic sense to do so.

Specific portability involves identifying the individual target environments in which a given program must run and clearly stating how those environments differ. Some examples of port scenarios follow:

- Moving from one operating system to another on the same machine (VAX Ultrix or UNIX to VAX/VMS; IBM MVS to VM/CMS; DEC's RSTS to RSX)
- Moving from a version of an operating system on one machine to the same operating system on another machine (UNIX on an AT&T Bell 3B2 to a VAX)
- Moving between variants of the same operating system (BSD4.x UNIX to System III or System V UNIX or to XENIX; PC-DOS to MS-DOS)
- Moving between two entirely different hardware and software environments (Honeywell GCOS to Cray COS)
- Moving between different compilers on the same system (Lattice C to Microsoft C under MS-DOS)
- Moving from one version of a compiler to another version of the same compiler

The last scenario may seem out of place. However, it is common to encounter problems when taking existing code that compiles without error, runs, and does the job adequately, and running it through a new version of the same compiler. The impact of the upgrade can vary from none to traumatic depending on how well the original code was written and the kinds of extra checking, new features, etc., that exist in the new compiler. While it can be argued that such upgrades are not typically traumatic, this situation is changing with the advent of the ANSI C Standard.

## Portability and the C Language

Perhaps the most noticeable aspect of such an upgrade is the ANSI C requirement that you be overt when mixing pointers of one type with another or with pointers and integer types or with pointers to data and pointers to functions.

**Recommendation:** Since ANSI C is a significantly different environment compared to all existing non-ANSI C environments, the task of upgrading to an ANSI C compiler (or one that has just implemented the bulk of ANSI C for the first time) should be considered as a port project, if you ever intend to recompile existing code with the new compiler in ANSI-mode.

There are even more subtle situations that really involve porting yet are not often recognized as such. One such example involves programming on MS-DOS systems that support various memory models. Consider the case where you develop an application in the small memory model, where all pointers are 16 bits. Eventually, adding extra code and/or data forces you to compile using another model. If data pointers now require 32 bits, you can no longer store such a pointer in an `int`, and `register` declarations of such pointers may be ignored since registers are only 16 bits long. Also, twice as much storage space is needed on the heap and the stack for data pointers that are automatically and dynamically allocated. And if you hard-coded the number of bytes to allocate as 20 instead of 10 times `sizeof(char *)`, your program will likely fail in mysterious ways.

Porting just doesn't involve getting a piece of software to work on multiple targets. It also involves doing so with a reasonable (and affordable) amount of resources, in a timely manner, and in such a way that the resulting code will perform adequately. There is little point in porting a system to a target such that when the port is complete, it runs so slowly or uses so many system resources, it is rendered unusable.

Important questions to ask yourself are:

- Am I porting to or from an ANSI C compiler? If so, are all targets ANSI-conforming?
- Am I porting code that was designed and written with portability in mind?
- Do I have the luxury of starting completely from scratch?
- Do I know what all the target environments will be up front and how many of them I will actually have available for testing?
- What are my performance requirements regarding speed, memory, and disk efficiency?

## 1.2 Portability is Not New

C is a relatively new programming language and since it has spawned a whole generation of programmers, many of its advocates likely have spent little, if any, time working with many other languages. Along with the wide availability of good and cheap C compilers and development tools in the early 1980s, the idea of software portability has become popular. So much so, that, to hear some people talk, portability became possible because of C.

The plain and simple facts are that the notion of portability is much older than C and that software was being successfully ported long before C (presumably) became an idea in Ritchie's head. In 1959, a small group defined a standard business language called COBOL, and in 1960, two vendors (Remington Rand and RCA) implemented compilers for that language. In December of that year, they conducted an experiment where they exchanged

COBOL programs, and according to Jean E. Sammet, a member of the COBOL design team, "... with only a minimum number of modifications primarily due to differences in implementation, the programs were run on both machines." Regarding COBOL's development of a description for data that is logically machine independent, Sammet wrote in 1969, "[COBOL] does not simultaneously preserve efficiency and compatibility across machines." It is the author's opinion that this statement holds true today for all languages commonly used as a portability vehicle.

The fact that a program was written in C provides no indication whatsoever as to the effort required to port it. The task may be trivial, difficult, impossible, or uneconomical. Given that a program was written in any language without regard to the possibility of its being ported to some different host environment, the ease with which it may actually be ported to that environment probably depends more on the discipline and idiosyncrasies of its author, than on the language itself. That is, porting an existing C program may inherently be no easier than porting any other language program. In fact, given C's strengths (and its initial purpose) in system programming, the idea of a low-level implementation language such as C being portable is inherently contradictory.

Designing a program to be portable over a range of environments, of which some may not yet be defined, may be difficult, but it is not impossible. It just requires considerable discipline and a good dose of common sense. The main aim in such a project is not to write a program that will run on any system without modification, but to isolate environment-specific functions so they may be rewritten for new systems. The major portability considerations are much the same for any language. Only the specific implementation details are determined by the actual language used.

### 1.3 Who Needs Portability?

There is no doubt that portability is a specialized topic and one that is not often implemented either because it is not deemed necessary or desirable or because the required resources are not available. In order to put the idea of portability in perspective, the author has developed a crude rule of thumb regarding its need and rate of success. The rule is, that perhaps no more than 5% of the programs being written are required to run in more than one environment, and of those 5%, only 5% are likely to be implemented successfully in more than one environment without a significant rewrite along the way. The intersection of these percentages is 0.25%, which indicates a fairly rarefied atmosphere. Even if the success rate were 100% of 5%, the requirement for portability is still very low.

Despite this, it has been this author's experience that perhaps seven (or more) out of ten people choosing C as the language for a given project, or as the strategic language for their company's future, cite portability as a major factor in arriving at that decision. Portability should become a factor only if it is, or is likely to be, required by the project being implemented. Unfortunately, this is not often the case. Not only is portability irrelevant for most C projects, it is also very likely that C itself is not the most appropriate language, but that topic is outside the scope of this book.

### 1.4 The Economics of Portability

Two main requirements for being successful at porting are: having the necessary technical expertise and tools for the job, and having management support as well as approval.

## Portability and the C Language

Clearly, one needs to have, or be able to get and keep, good C programmers. The term *good* does not imply guru status alone or at all since such staff can often have egos that are difficult to manage. And perhaps the most important attribute required in a successful port project is discipline both at the individual and at the group levels. So even if the technicians are not stellar, the project can succeed if the project manager is experienced and well disciplined.

The issue of management support is often more important, yet it is largely ignored both by the developers and by management itself. The following (not atypical) example should demonstrate the problem.

A software company decides to develop a commercial package in C that will run on both MS-DOS and UNIX systems. A main part of the rationale is that these systems represent a significant piece of the retail market permitting a potentially large number of units to be sold.

Now due to the relatively low cost of MS-DOS-class machines, system software, and tools, each programmer has his or her own DOS system. And expanding the team is as easy as getting another PC configuration. No developer impacts any other developer since each has his or her own development resource. Perhaps the systems are even linked using some form of local area network (LAN) so resources can be shared.

Regardless of good intent, an implied order of precedence almost always exists when it comes to selecting the target environments. And if such an order exists, either firmly stated or simply “understood,” this should be made known and accepted. In this example, the company may well be “interested” in the UNIX market because it seems to be large and growing, yet this same company won't spend, for example, the \$30,000 or so up front needed for a MicroVAX and UNIX System V licence. That is, supporting UNIX “would be nice, but we're not quite sure, yet.”

So, the developers implement XENIX (a UNIX derivative) on one of their 80x86 systems so they can test the C sources. Now while this permits the code to be run under XENIX, it by no means is tested against UNIX System V (given that XENIX is not currently System V-conformant). And since XENIX runs on the same processor as their DOS versions, the developers do not run into machine-specific issues. So, even if the project is completed and runs on both DOS and XENIX, this is no guarantee it will run on another derivative of UNIX or, indeed, a version of XENIX running on another processor family.

A second possibility is that adequate hardware and software is provided for all (or a representative subset) of the specified target environments and the development group religiously runs all its code through all targets at least weekly. Often, it submits a test stream in batch every evening.

Six months into the project, management reviews progress and finds that the project is taking more resources than anticipated (doesn't it always?) and decides to narrow the set of targets, at least on a temporary basis. That is, “we have to have something tangible to demonstrate at tradeshow since we have already announced the product” or “the venture capitalists are expecting to see a prototype at the next board meeting.” Whatever the reasons, testing on, and development specifically for, certain targets is suspended, often permanently.

From that point on, the development group must ignore the idiosyncrasies of the dropped machines since they are no longer part of the project and the company can't afford the extra resources to consider them seriously. Of course, management's suggestion is typically, “while we don't want you to go out of your way to support the dropped environments, it would be nice if you don't do anything to make it impossible or inefficient for us to pick

them up again at some later date.” That is, “do what you can, but we’re not going to give you the resources to do the job properly.”

Of course, as the project slips even further, competitors announce and/or ship alternative products, or the company falls on hard economic times, other targets may also be dropped, possibly leaving only one since that is all that development and marketing can support. And each time it drops a target, the development group starts to “cut corners” since it no longer has to worry about “that particular piece of deficient” hardware and/or operating system. Ultimately, this decreases the chances of ever starting up on dropped targets at some later date as all code designed and written since support for those targets was dropped needs to be inspected (assuming, of course, that this code can even be identified) to ascertain the effort required and the impact on resuming that target. You may well find that certain design decisions that were made either prohibit or negatively impact reactivating the abandoned project(s).

The end result often is that the product is initially delivered for one target only and is never made available in any other environment. Another common situation is to deliver for one target, and then go back and salvage “as much as possible” for one or more other targets. In such cases, the project may be no different from one in which you are porting code that was never designed with portability in mind.

A third situation occurs when you try to anticipate what a specific target (or family of targets) will require from a portability viewpoint. This can happen because either you have abandoned that target or it is perceived as being a long-term possibility. (For example, you might try to develop software that will run on any twos-complement, ASCII, byte architectures with word sizes of 16, 32, or 64 bits.) Whatever the reason, this is usually an untenable situation both because you can’t know everything there is to know without actually experimenting, and because support for this “optional” target is not officially sanctioned.

## 1.5 Measuring Portability

How do you know when or if a system has been successfully ported? Is it when the code compiles and links without error? Is it when the same output is produced by all targets?

Certainly, the code must compile and link without error, but because of implementation-defined behavior, it may be quite possible to get different results from different targets. The legitimate results may even be sufficiently different as to render them useless. For example, floating-point range and precision may vary considerably from one target to the next such that results produced by the most limited floating-point environment are not precise enough. Of course, this is a design consideration and should be considered before the system is ported.

A general misconception is that exactly the same source code files must be used on all targets such that the files are full of conditionally compiled lines. This need not be the case at all. Certainly, you may require custom headers for some targets. You may also require system-specific code written in C, and possibly in assembler or other languages. Provided such code is isolated in separate modules and the contents of, and the interfaces to, such modules is well documented, this approach need not be a problem—in fact, it may well be necessary.

If you are using the same data files across multiple targets, you will need to ensure that the data is ported correctly, particularly if it is stored in binary rather than text format. If you do not, you may waste considerable resources looking for code bugs that do not, in fact, exist.

The bottom line is that, unless you have adequately defined what your specific portability scenario is, you cannot tell when you have achieved it. And by definition, if you achieve it, you must be satisfied. If you are not, either your requirements have changed, or your design was flawed. And most importantly, successfully porting a program to some given number of environments is not necessarily a reliable indication of the work involved in porting it to yet another target.

### 1.6 The Porting Toolbox

A number of basic tools are necessary or desirable when porting code. Even though most of them may be obvious, they are listed nonetheless.

- A full-screen text editor and preferably with some knowledge about the C language. For example, when you enter a left brace, it can be programmed to insert a new-line and indent the next line by an extra horizontal tab.
- A code beautifier or reformatter is used to make code produced by different developers, or by the same developer on different occasions, look consistent and be easily recognized during maintenance.
- A source code pagination and listing utility preferably with a cross-reference capability that shows not only where each identifier is used and its type, class, and other declaration attributes but also whether it is used as a modifiable lvalue.
- A `lint`-like static analysis tool that performs code quality checking beyond that provided by most compilers.
- A C compiler that provides more than just fatal error messages. *Quality of implementation* is a term that was discussed often at ANSI C meetings as the reason for not putting something in the C Standard. So whether your compiler provides warning or informational messages is up to the implementer. Various other compile-time options are highly desirable. They include: the ability to extract function prototypes from source input files and the ability to view not only the final output produced from the preprocessor but also intermediate stages of macro expansion when macros are nested. The ability to save preprocessor output as a file suitable for input to the compiler (without modification) is useful as is the ability to have the preprocessor preserve all whitespace (including comments).
- A symbolic debugger that permits debugging at the C source level. This implies that it is case-sensitive and can handle the same length identifiers the compiler produces, as well as variables that happen to be optimized into registers.
- A make-like system build facility to ensure quality assurance when source modules and/or headers are changed.
- A source code control system to maintain sources, headers, and compilation and build procedures, in a project involving a large number of modules and/or programmers.
- A `grep`-like text search tool that can be used to search a file (or set of files) for a given text string or series of adjacent C tokens.
- Communications software (or other media) that permits reliable transmission and/or conversion of source and data from one target to another.
- Some type of list management tool that permits you to maintain an on-line data base of external identifiers currently in use by user-written source, or reserved by your compilers and third-party libraries and tools. Not only should a user be able to find if a particular name is in use or reserved, they should

also be able to enquire about specific spelling patterns such as `wnd*`, for example, where `*` represents an unlimited length text string, possibly of length zero.

## 1.7 Environmental Issues

As pointed out elsewhere in this chapter, many portability issues have little or nothing to do with the implementation language. Rather, such issues are relevant to the hardware and operating system environments on which the program must execute. Many of these issues are hinted at in the main body of this book, but they are summarized here as follows:

- Mixed-language environments. Certain requirements may be placed on C code that is to call, or be called by, some other language processor. For example, calling VAX C from VAX Fortran (or any other VAX language using the `CALLG` calling instruction) on VAX/VMS requires the called C function to treat its formal argument list as read-only. It cannot assume such arguments exist as copies on the stack, since this is not the case. Similarly, calling Microsoft FORTRAN, Pascal, or Quick Basic from Microsoft C requires the use of extended keywords and/or the use of special compile-time switches so the stack setup and cleanup mechanism used across function calls is compatible.
- Command-line processing. Not only do different command-line processors vary widely in their behavior, but the equivalent of a command-line processor may not even exist for some of your targets.
- Data representation. This is, of course, completely implementation-defined and may vary widely. Not only can the size of an `int` differ across your targets, but you are not even guaranteed that all bits allocated to an object are used to represent the value of that object. For example, on a Cray super-computer, `sizeof(short)` is 8 yet only 24 of the 64 bits allocated are used to represent the `short int` value. Another significant problem is the ordering of bytes within words and words within longwords. Such encoding schemes are often referred to as *big-endian* or *little-endian*. Also, this has been referred to as the NUXI problem, indicating the possible ordering of the letters when the word UNIX is stored in a 4-byte quantity. (Refer to Annex B for references on this topic.)
- CPU speed. It is a common practice to “know” that executing an empty loop  $n$  times in a given environment causes a pause of 5 seconds, for example. However, running the same program on an Intel 80386 instead of an 8088 will invalidate this approach. (The same is true when running it on versions of the same processor having different clock speeds.) Or perhaps the timing is slightly different when more (or less) programs are running on the same system. Related issues include the frequency and efficiency of handling timer interrupts, both hardware and software.
- Operating system. The principal issues here are single- versus multi-tasking and fixed- versus virtual-memory organization. Other issues involve the ability to field synchronous and asynchronous interrupts, the existence of reentrant code, and shared memory. Seemingly, simple tasks such as getting the system date and time may be impossible on some systems. Certainly, the granularity of system time measurement varies widely.
- File systems. Whether multiple versions of the same file can coexist or whether the date and time of creation or last modification are stored, is implementation-defined. Likewise, for the character set permitted in file names and whether or not such names are case-sensitive. And as for device and directory-naming conventions, the variations are as broad as their inventor's imaginations. Consequently,

the ANSI C Standard says nothing about file systems except for sequential files being accessed by a single user.

- Development support tools. These necessary tools may have a significant impact on the way you write, or are required to write, code for a given system. They include C translator, linker, object and source librarian, assembler, source-code management system, macro preprocessors, and utility libraries. Examples of restrictions include the casing, significance, and number of external identifiers; perhaps even the size of each object module or the number and size of source modules. Perhaps the overlay linker has significant restrictions on the complexity of the overlay scheme.
- Compiler optimization. Many C programmers “know,” for example, that `i++` is more efficient than `i = i + 1` and that certain unreadable expressions involving numerous comma and conditional operators are “necessary” for efficiency reasons. While this can, in fact, be true, it is often not the case. Optimization technology is sufficiently advanced that what used to be the case on your old compiler 3–5 years ago is almost certainly no longer true. For example, as predicted by Ritchie in 1978, the `register` storage class is ignored by most compilers since they are better equipped than most programmers to decide just which variables are used more often than others.

Not only can different code be generated when you upgrade your compiler (or simply use different optimization command-line switches), it may also result with the same compiler when a few lines of code and/or data are added or removed since, now, the optimizer views the code from a different perspective.

- Cross-compilation. In environments where the target is not the system on which the software is being developed, differences in character sets and arithmetic representations become important.
- Screen and keyboard devices. The protocols used by these vary widely. While many implement some or all of various ANSI standards, just as many do not or contain incompatible extensions. Getting a character from standard input without echoing it, or without needing to press the return or enter key as well, is not universally possible. The same is true for direct cursor addressing and graphics display and input devices such as light pens, track balls, and mice.
- Other peripheral interfaces. Your design may call for interactions with printers, plotters, scanners, and modems, among other pieces of equipment. While some *de facto* standards may exist for each, you may be forced, for one reason or another, to adopt “slightly” incompatible pieces. And beware of compatibility claims. For example, the Victor 9000 was a PC-compatible system except for the “minor” facts that it could neither read nor write PC-DOS-compatible diskettes and that its video interface was different. DEC's Rainbow was likewise “PC-compatible.”

## 1.8 Programmer Portability

In all the discussions on portability, we continually refer to the aspect of moving code from one environment to another. And while this is an important consideration, it is more likely that C programmers will move to a different environment more often than the software they write. For this reason, the author has coined the term *programmer portability*.

Programmer portability can be defined as the ease with which a C programmer can move from one environment to another. This is an issue important to any C project, not just that involving code portability. If you adopt certain programming strategies and styles, you can make it much easier and quicker to integrate new team members into the project. Note though that, while you may have formulated a very powerful approach, if it is too far from the mainstream C practice, it will either be difficult and/or expensive to teach or to convince other

C programmers of its merits. An example of this could be the use of mixed-case identifier names or the use of macro names spelled entirely in lower-case.

## 5. Expressions

Expressions are the fundamental building blocks of a C program, and as such, C programmers must understand them and how they interact.

ANSI C affects expressions in that it defines a new operator (unary plus), it specifically defines and identifies sequence points at which a subexpression must be completely evaluated, and it forces grouping parentheses to be honored in all cases.

### 5.1 General Information

The order in which subexpressions are evaluated is unspecified except for the function call operator `()`, the logical OR operator `||`, the logical AND operator `&&`, the comma operator and the conditional operator `? :`. While the precedence table defines certain operator precedences and associativity, these can be overridden by grouping parentheses. However, according to K&R, the commutative and associative binary operators `*`, `+`, `&`, `|`, `^` may be arbitrarily rearranged EVEN if grouping parentheses are present. (Note that for `&`, `|`, and `^`, the ordering is unimportant since the same result is always obtained.) However, an ANSI-conforming compiler MUST honor grouping parentheses in ALL expressions.

With the K&R rules (not ANSI), even though you may write

```
i = a + (b + c);
```

the expression may be evaluated as

```
i = (a + b) + c;
```

or even

```
i = (a + c) + b;
```

This can cause overflow on intermediate values if the expression is evaluated one way versus another. To force a specific order of evaluation, break up the expression into multiple statements and use temporary intermediate variables as follows:

```
i = (b + c);
i += a;
```

These examples cause a problem only in “boundary” conditions and even then only on some machines. For example, integer arithmetic on a twos-complement machine is usually “well behaved.” (However, some machines raise an interrupt when integer overflow occurs and presumably, this should be avoided.) In the following example (run on an Intel 8088 system), the same result is obtained regardless of the order of evaluation.

```

#include <stdio.h>

main()
{
    int i;
    int a = 32767;
    int b = 10;
    int c = -20;

    i = a + b + c;
    printf("i = %d\n", i);
    i = a + c + b;
    printf("i = %d\n", i);

    i = b + c + a;
    printf("i = %d\n", i);
}

i = 32757
i = 32757
i = 32757

```

(An inspection of the machine code generated for this example showed that the compiler evaluated the expressions in the same order in which they were written. In the above case,  $a + b$  overflows producing a value of  $-32759$ , which when added to  $c$  ( $-20$ ), overflows again giving  $32757$ , the correct answer.)

**Recommendation:** If you are concerned about the order of evaluation of expressions that associate and commute, break them into separate expressions such that you can control the order. Find out the properties of integer arithmetic overflow for your target systems and see if they affect such expressions.

The potential for overflow and loss of precision errors is much higher with floating-point operands where it is impossible to represent accurately certain real numbers in a finite space. Some mathematical laws that do not always hold true using finite representation are:

```

(x + y) + z == x + (y + z)
(x * y) * z == x * (y * z)
(x / y) * y == x          /* for non-zero y */
(x + y) - y == x

```

When  $a$ ,  $b$ , and  $c$  are simple expressions involving constants or variables (as in the earlier example above), the order of evaluation is often irrelevant to the outcome as discussed above. However, if they are expressions that involve side effects, the order may be quite important. For example,

```
test()
{
    int i, f(), g();

    i = f() + g();
}
```

Here, `f()` may be evaluated before or after `g()`. While the value of `i` might be the same in either case, if `f()` and `g()` produce side effects, this may not be true. For example,

```
extern int j;

f()
{
    return (j += 5);
}

extern int j;

g()
{
    return (j -= 5);
}
```

If `f` is called before `g`, a different value will be assigned to `i` than if `g` had been called first. However, `j` will have the same final value either way. To force a specific order of evaluation, use something like

```
i = f();
i += g();
```

The order in which side effects take place is unspecified. For example, the following are unsafe expressions:

```
j = (i + 1)/i++;
dest[i] = source[i++]
dest[i++] = source[i]
i & i++
i++ | i
i * -i
```

Which expression containing `i` is evaluated first in each line above is undefined.

**Recommendation:** Even if you can determine how your compiler evaluates expressions that contain side effects, don't rely on this being true for future releases of the same product. It may even vary for the same compiler given different circumstances. For example, by changing the source code in other, possibly unrelated, ways, you may change the optimizer's view of the world

## 8. The Preprocessor

According to the ANSI C Standard Rationale document, “Perhaps the most undesirable diversity among existing C implementations can be found in preprocessing. Admittedly a distinct and primitive language superimposed upon C, the preprocessing commands accreted over time, with little central direction, and with even less precision in their documentation.”

### 8.1 General Information

#### 8.1.1 Preprocessor versus Compiler

Many C compilers involve multiple passes, the first of which often contains the preprocessor. Using this knowledge, a compiler can often take short cuts by arranging information to be shared between the preprocessor and the various phases of the compiler proper. While this may be a useful feature for a particular implementation, you should keep in mind that other implementations may use completely separate, and noncooperating, programs for the preprocessor and the compiler.

**Recommendation:** Keep the ideas of preprocessing and compilation separate. One possible problem when you fail to do this will be demonstrated when the `sizeof` operator is used as discussed below.

Although C is a free-format language, the preprocessor need not be since, strictly speaking, it is not part of the C language. The language and the preprocessor each have their own grammars, constraints, and semantics.

#### 8.1.2 The Directive Name Format

A preprocessing directive always begins with a `#` character. However, not all preprocessors require the `#` and the directive name to be one token. That is, the `#` prefix may be separated from the directive name by spaces and/or horizontal tabs. For example, the following macro definitions:

```
#define MAX 26
# define MAX 26
#  define MAX 26
#<tab><tab>define MAX 26
```

may all be treated equally (and correctly) by some preprocessors; yet, all but the first will be rejected by other preprocessors.

K&R shows the `#` as part of the directive name, with no intervening white space. No statement is made as to whether such white space is permitted.

ANSI C permits an arbitrary number of horizontal tabs and spaces between the # and the directive name. The # and the directive name (such as `define` and `include`) are considered to be separate tokens.

**Recommendation:** The directive name should immediately follow the # character with no intervening white space unless all of your preprocessors support otherwise.

### 8.1.3 Start Position of Directives

Many preprocessors permit directives to be preceded by white space allowing indenting of nested directives as follows.

```
#if condition-1a
    #if condition-2a
        #if condition-3a
            ...
        #endif
    #else
        ...
    #endif
#else
    #if condition-2b
        #if condition-3b
            ...
        #endif
    #else
        ...
    #endif
#endif
```

Less flexible preprocessors require the # character to be the first character of a source line, thus requiring the above construct to be written instead as

```
#if condition-1a
#if condition-2a
#if condition-3a
...
#endif
#else
...
#endif
#else
#if condition-2b
#if condition-3b
...
#endif
```

```
#else
...
#endif
#endif
```

This latter format completely lacks any implied meaning regarding the scope of the conditional inclusion directives.

Some implementations require directives to begin in the first column; yet, they permit white space between the # and the directive name. In such cases, indenting can still be achieved as follows:

```
#if condition-1a
#   if condition-2a
#       if condition-3a
#           ...
#       endif
#   else
#       ...
#   endif
#else
#   if condition-2b
#       if condition-3b
#           ...
#       endif
#   else
#       ...
#   endif
#endif
```

K&R states that “Lines beginning with # communicate with this preprocessor.” No definition for “beginning with” is given.

ANSI C permits an arbitrary amount of white space before the # character. This white space is not restricted to horizontal tabs and spaces—any white space is allowed.

**Recommendation:** Always begin directives at the first character on a line.

### 8.1.4 White Space Within Directives

ANSI C requires that all white space appearing between the directive name and the directive's terminating new-line be horizontal tabs and/or spaces. Therefore, the following directives contain syntax errors since the white space includes non-space, nontab characters. (The sequences <VT>, <CR>, <FF>, and <BS> represent a vertical tab, carriage return, form feed, and backspace, respectively.)

## 16. `math.h` – Mathematics

The ANSI C headers are supposed to be self-sufficient; however, it may be necessary to include `errno.h` to guarantee that the macros `EDOM` and `ERANGE` are defined and that `errno` is declared properly.

The math function names created by adding a suffix of `f` or `l` are reserved for implementations of `float` and `long double` versions, respectively. However, an ANSI-conforming implementation is required to support only the `double` set. In the case of the `float` set, these functions must be called in the presence of an appropriate prototype; otherwise, `float` arguments will be widened to `double`. (Note though that specifying `float` in a prototype does not necessarily force such widening to be disabled; this aspect of prototypes is implementation-defined. However, it is necessary when supporting the `float` set.)

The functions `ecvt`, `fcvt`, and `gcvt` are not defined by ANSI C since their capability is available via `sprintf`.

`math.h` contains one macro definition, that for `HUGE_VAL`. This expands to a positive `double` expression that is not necessarily representable as a `float`. ANSI C does not require it to be a constant expression. Some implementations, including SVID, call this macro `HUGE` instead.

ANSI C does not declare `abs` in `math.h`, but it does declare it in `stdlib.h`. If `abs` is implemented as a function, and `math.h` is included rather than `stdlib.h`, the correct result should be obtained as `abs` takes and returns `int` values. However, if `abs` is a macro, `abs` will become an unsatisfied external reference at link-time. Likewise, `labs` is declared in `stdlib.h`. However, if this is called without the correct declaration, `int` instead of `long int`, will be assumed for the return value.

ANSI C has invented the `div` and `ldiv` math functions for computing quotients and remainders. These are declared in `stdlib.h`, as are the random number functions `rand` and `srand`.

The following functions are commonly declared in `math.h`; however, they are not defined by ANSI C: `cabs`, `erf`, `erfc`, `gamma`, `hypot`, `j0`, `j1`, `jn`, `poly`, `pow10`, `y0`, `y1`, and `yn`. The same is true for structure tags and types associated with complex arithmetic.

### 16.1 Treatment of Error Conditions

A domain error occurs if an input argument is outside the domain over which the mathematical function is defined. In this case, an implementation-defined value is returned, and `errno` is set to the macro `EDOM`. (Refer to `sqrt(-0)` for an example.)

A range error occurs if the result of the function cannot be represented as a `double`. If the result overflows, the function returns the value of `HUGE_VAL`, with the same sign as the correct value would have. `errno` is set to the macro `ERANGE`. If the result underflows, the function returns 0 and `errno` may or may not be set to `ERANGE`, as the implementation defines.

The `matherr` machinery used by SVID (and others) is not included in ANSI C partly because it requires a user-defined library function. `matherr` requires a structure type `exception` and the macros `DOMAIN`, `OVERFLOW`, `PLOSS`, `SING`, `TLOSS`, and `UNDERFLOW` to be defined.

## 16.2 Trigonometric Functions

### 16.2.1 The `acos` Function

**Calling sequence:**

```
#include <math.h>

double acos(double x);
```

**Description:** `acos` computes the arc cosine of its argument.

**Comments:**

- If the argument is not in the range  $[-1,+1]$ , a domain error occurs.
- The value returned is the arc cosine in the range  $[0,\pi]$  radians.

### 16.2.2 The `asin` Function

**Calling sequence:**

```
#include <math.h>

double asin(double x);
```

**Description:** `asin` computes the arc sine of its argument.

**Comments:**

- If the argument is not in the range  $[-1,+1]$ , a domain error occurs.
- The value returned is the arc sine in the range  $[-\pi/2,+\pi/2]$  radians.

### 16.2.3 The `atan` Function

**Calling sequence:**

```
#include <math.h>

double atan(double x);
```

**Description:** `atan` computes the arc tangent of its argument.

**Comments:**

- The value returned is the arc tangent in the range  $[-\pi/2,+\pi/2]$  radians.

# 21. stdio.h – Input/Output

## 21.1 File Systems

Almost all aspects of file and directory systems are implementation-defined. So much so that ANSI C cannot even make a statement about the most basic thing, a filename. Just what filenames can and must an implementation support? And as for directory and device names, there is nothing close to a common approach.

Some implementations may permit filenames to contain wildcards. That is, the file specifier may refer to a group of files using a convention such as \*.dat to refer to all files with a type of .dat. None of the standard I/O routines is required to support such a notion.

Numerous operating systems can limit the number of open files on a per user basis. VAX/VMS even can limit the number of versions of any given file in a directory. And if you create one too many, the oldest one is transparently deleted. Note, too, that not all systems permit multiple versions of the same filename in the same directory, and this has consequences when you use fopen with "w" mode, for example.

Some file systems also place disk quotas on users such that an I/O operation may fail when a file grows too big—you may not know this until an output operation fails.

Back to the filename issue. After extensive investigation, the ANSI C committee found that the format of a portable filename is up to six alphabetic characters followed by a period and none or one letter. And given that some file systems are case-sensitive, these alphabetic characters should all be the same case. However, rather than restrict yourself to filenames of the lowest common denominator, the following approach is suggested:

```
/* files.h - filename header */

#if HOST == DOS || HOST == XENIX
#define MASTER "\system\data\master.dat"
#elif HOST == VMS
#define MASTER "DBA3:[system.data.]master.dat"
...
#endif

/* your code */
#include "files.h"
...
fp = fopen(MASTER, "r");
i = remove(MASTER);
i = rename(OLDNAME, NEWAME);
```

So by conditionally compiling within the header files.h, your code becomes readable and the file system dependencies are isolated from your code.

## Portability and the C Language

The whole concept of filename redirection at the command-line level is also implementation-defined. If possible, it means that `printf` and `fscanf`, for example, may actually be dealing with devices other than the user's terminal. They could even be dealing with files. Note that `gets` behaves slightly differently to `fgets` from `stdin`, yet `gets` could be reading from a file if `stdin` were redirected.

The details of file buffering, disk sector (or block) sizes, etc., are also implementation-defined. However, ANSI C requires an implementation to be able to handle text files with lines containing at least 254 characters, including the trailing new-line.

On some systems, (such as UNIX, VAX/VMS, and DOS), `stdin`, `stdout`, and `stderr` are special to the operating system and are maintained by it. On other systems, these may be established during program startup. Whether these files go against your maximum open file limit is implementation-defined.

`stdio.h` defines several types and macros and declares numerous functions useful for performing file and terminal I/O.

### 21.1.1 Defined Types

`size_t` is defined in several headers and is discussed in detail under `stddef.h`.

`FILE` is an object type capable of containing the “current context” of an open file. This information includes, buffering details, error and end-of-file flags, file position indicator, and other implementation-defined fields. Typically, `FILE` is a structure, although it may actually be a pointer to a structure, in which case, `FILE *` is really a pointer to a pointer a structure. You should always traffic in `FILE` objects rather, than their underlying equivalents, since they vary considerably from one implementation to another. (Some implementations define `FILE` using `#define` rather than `typedef`.)

The type `fpos_t` is used by `fgetpos` and `fsetpos`. It must map to a type large enough to hold the largest possible file position indicator for the implementation. It may be an array, a structure, a `long int`, etc., as appropriate. You should always traffic in `fpos_t` objects rather than their underlying equivalents since they vary considerably from one implementation to another.

### 21.1.2 Defined Macros

`NULL` is defined in several headers and is discussed in detail under `stddef.h`.

`_IOFBF`, `_IOLBF`, and `_IONBF` are distinct integral constant expressions that can be used with the third argument to `setvbuf`.

`BUFSIZ` is an integral constant expression that is the size of the buffer to be used by `setbuf`. ANSI C requires it to be at least 256.

`EOF` is a negative integral constant expression that is returned by numerous functions to indicate an end-of-file condition. (Some implementations explicitly document `EOF` as having the value `-1`. However, this is not required by ANSI C.)

`FOPEN_MAX` is an integral constant expression that is the minimum number of files that the implementation guarantees you can have open simultaneously. ANSI C requires `FOPEN_MAX` to be at least eight including `stdin`,

`stdout`, and `stderr`. Note that some operating systems (such as DOS and VAX/VMS) can limit the number of open files on a per user basis at the operating system level. Apart from files you explicitly open, those opened by `tmpfile` also are counted against `FOPEN_MAX`. This name was an ANSI C invention and earlier versions of it were called `SYS_MAX` and `OPEN_MAX`.

`FILENAME_MAX` is an integral constant expression that is the maximum length of a filename string that can be used with the implementation. ANSI C specifies no minimum size.

`L_tmpnam` is an integral constant expression that is the size of a character array large enough to contain the temporary filename generated by `tmpnam`.

`SEEK_CUR`, `SEEK_END`, and `SEEK_SET` are distinct integral constant expressions that can be used with the third argument to `fseek`. These are relatively new additions to C, and their values were previously hard-coded in the call. (These values are often, 1, 2, and 0, respectively, however, these values are not required by ANSI C.)

`TMP_MAX` is an integral constant expression that is the maximum number of unique filenames that can be generated by `tmpnam`. ANSI C requires `TMP_MAX` to be at least 25.

`stdin`, `stdout`, and `stderr` are expressions of type `FILE *` that point to file objects corresponding to “standard input,” “standard output,” and “standard error,” respectively. These expressions need not designate modifiable lvalues (and usually don't.)

## 21.2 Common Extensions

A number of other macros are often defined in `stdio.h`, particularly in UNIX implementations. These include: `L_ctermid`, `L_cuserid`, `L_lcltmpnam`, `L_nettmpnam`, `P_tmpdir`, `max`, and `min`. Two very common ones are `TRUE` and `FALSE`. None of these are defined by ANSI C.

The UNIX I/O functions that deal in file descriptors rather than `FILE` pointers have been widely emulated outside UNIX. UNIX implementations have also defined numerous other I/O functions over the years. These functions include: `open`, `close`, `read`, `write`, `fileno`, `creat`, `read`, `write`, `getw`, `putw`, `getch`, `getche`, and `lseek`. On some systems the macros `STDIN`, `STDOUT`, and `STDERR` were defined to deal with special file descriptors. None of these are defined by ANSI C.

DOS C implementations often support the extra file pointers `stdprn` and `stdaux` that point to the default printer and auxiliary device (often COM1:), respectively.

## 21.3 Operations on Files

### 21.3.1 The remove Function

**Calling sequence:**

```
#include <stdio.h>

int remove(const char *filename);
```

# Annex A. Keywords and Reserved Identifiers

Most of the information in this appendix is taken from the proposed ANSI Standard draft, dated May 1988. While most of the headers will likely remain the same in the final Standard, they are subject to change. New headers and/or identifiers may be added (and, therefore, will become reserved), and identifiers may be removed or have their names changed. Note that the identifier lists include macro, function, typedef, and structure tag names.

Referred to the official ANSI C Standard documents for a precise definition of the standard header contents.

## A.1 C Keywords

The following tokens are defined as keywords by ANSI C: auto, break, case, char, const, continue, default, do, double, else, entry, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, and while.

## A.2 C++ Keywords

The following tokens are defined as keywords in the C++ language but not by ANSI C: asm, class, delete, friend, inline, new, operator, overload, public, this, and virtual. Avoid using these identifiers if you intend to move C programs to a C++ environment.

C++ implementations provide a header called `stream.h`, which defines and declares numerous identifiers. These identifiers are included in the alphabetical list at the end of this chapter. Note that at this writing, the published definition of C++ (see Appendix B) is not a strict superset of ANSI C so problems may be encountered when porting ANSI-conforming C code to C++. For example, in C++, structure, union, and enumeration tags share the same name space as ordinary identifiers.

## A.3 The ANSI Standard Headers

The proposed Standard headers and their purposes follow.

Header	Purpose
<code>assert.h</code>	program diagnostic purposes
<code>ctype.h</code>	character testing & conversion
<code>errno.h</code>	various error checking facilities
<code>float.h</code>	floating type characteristics
<code>limits.h</code>	integral type sizes
<code>locale.h</code>	internationalization support
<code>math.h</code>	math functions
<code>setjmp.h</code>	nonlocal jump facility
<code>signal.h</code>	signal handling

Header	Purpose
stdarg.h	variable argument support
stddef.h	miscellaneous
stdio.h	input/output functions
stdlib.h	general utilities
string.h	string functions
time.h	date and time functions

## A.4 Identifiers Alphabetically by Header

The following list contains the identifiers in alphabetical order within header. (Those in `<math.h>` marked with a † are not required to be present in an ANSI-conforming implementation. They have a suffix of `f` or `l` and represent `float` and `long double` versions, respectively, of the `math` library. If an implementation chooses to provide functions with this capability, they must use the names as shown.)

Header	Identifier	Identifier	Identifier
<code>&lt;assert.h&gt;</code>	<code>assert</code>		
<code>&lt;ctype.h&gt;</code>	<code>isalnum</code>	<code>isalpha</code>	<code>iscntrl</code>
	<code>isdigit</code>	<code>isgraph</code>	<code>islower</code>
	<code>isprint</code>	<code>ispunct</code>	<code>isspace</code>
	<code>isupper</code>	<code>isxdigit</code>	<code>tolower</code>
	<code>toupper</code>		
<code>&lt;errno.h&gt;</code>	<code>EDOM</code>	<code>ERANGE</code>	<code>errno</code>
<code>&lt;float.h&gt;</code>	<code>DBL_DIG</code>	<code>DBL_EPSILON</code>	<code>DBL_MANT_DIG</code>
	<code>DBL_MAX</code>	<code>DBL_MAX_10_EXP</code>	<code>DBL_MAX_EXP</code>
	<code>DBL_MIN</code>	<code>DBL_MIN_10_EXP</code>	<code>DBL_MIN_EXP</code>
	<code>FLT_DIG</code>	<code>FLT_EPSILON</code>	<code>FLT_MANT_DIG</code>
	<code>FLT_MAX</code>	<code>FLT_MAX_10_EXP</code>	<code>FLT_MAX_EXP</code>
	<code>FLT_MIN</code>	<code>FLT_MIN_10_EXP</code>	<code>FLT_MIN_EXP</code>
	<code>FLT_RADIX</code>	<code>FLT_ROUNDS</code>	<code>LDBL_DIG</code>
	<code>LDBL_EPSILON</code>	<code>LDBL_MANT_DIG</code>	<code>LDBL_MAX</code>
	<code>LDBL_MAX_10_EXP</code>	<code>LDBL_MAX_EXP</code>	<code>LDBL_MIN</code>
	<code>LDBL_MIN_10_EXP</code>	<code>LDBL_MIN_EXP</code>	
<code>&lt;limits.h&gt;</code>	<code>CHAR_BIT</code>	<code>CHAR_MAX</code>	<code>CHAR_MIN</code>
	<code>INT_MAX</code>	<code>INT_MIN</code>	<code>LONG_MAX</code>
	<code>LONG_MIN</code>	<code>MB_LEN_MAX</code>	<code>SCHAR_MAX</code>
	<code>SCHAR_MIN</code>	<code>SHRT_MAX</code>	<code>SHRT_MIN</code>
	<code>UCHAR_MAX</code>	<code>UINT_MAX</code>	<code>ULONG_MAX</code>

Header	Identifier	Identifier	Identifier
	USHRT_MAX		
<locale.h>	lconv	LC_ALL	LC_COLLATE
	LC_CTYPE	LC_MONETARY	LC_NUMERIC
	LC_TIME	localeconv	NULL
	setlocale		
<math.h>	acos	acosf <sup>†</sup>	acosl <sup>†</sup>
	asin	asinf <sup>†</sup>	asinl <sup>†</sup>
	atan	atanf <sup>†</sup>	atanl <sup>†</sup>
	atan2	atan2f <sup>†</sup>	atan2l <sup>†</sup>
	ceil	ceilf <sup>†</sup>	ceill <sup>†</sup>
	cos	cosf <sup>†</sup>	cosl <sup>†</sup>
	cosh	coshf <sup>†</sup>	coshl <sup>†</sup>
	exp	expf <sup>†</sup>	expl <sup>†</sup>
	fabs	fabsf <sup>†</sup>	fabsl <sup>†</sup>
	floor	floorf <sup>†</sup>	floorl <sup>†</sup>
	fmod	fmodf <sup>†</sup>	fmodl <sup>†</sup>
	frexp	frexpf <sup>†</sup>	frexpl <sup>†</sup>
	HUGE_VAL		
	ldexp	ldexpf <sup>†</sup>	ldexpl <sup>†</sup>
	log	logf <sup>†</sup>	logl <sup>†</sup>
	log10	log10f <sup>†</sup>	log10l <sup>†</sup>
	modf	modff <sup>†</sup>	modfl <sup>†</sup>
	pow	powf <sup>†</sup>	powl <sup>†</sup>
	sin	sinf <sup>†</sup>	sinl <sup>†</sup>
	sinh	sinhf <sup>†</sup>	sinhl <sup>†</sup>
sqrt	sqrtf <sup>†</sup>	sqrtl <sup>†</sup>	
tan	tanf <sup>†</sup>	tanl <sup>†</sup>	
tanh	tanhf <sup>†</sup>	tanhl <sup>†</sup>	
<setjmp.h>	jmp_buf	longjmp	setjmp
<signal.h>	raise	SIGABRT	SIGFPE
	SIGILL	SIGINT	signal
	SIGSEGV	SIGTERM	sig_atomic_t
	SIG_DFL	SIG_ERR	SIG_IGN
<stdarg.h>	va_arg	va_end	va_list
	va_start		