Expert C Programming

Deep C Secrets

Peter van der Linden

Expert C Programming

Deep C Secrets

Peter van der Linden

Addison-Wesley

Library of Congress Cataloging-in-Publication Data
Van der Linden, Peter
Expert C Programming! / Peter van der Linden.
p. cm.
Includes index.
ISBN 0-13-177429-8
1. C (Computer program language) I. Title.
QA76.73.C15V356 1994
005.13'3--dc20 94-253
CIP

The publisher offers discounts on this book when ordered in bulk quantities. For more information, contact: Corporate Sales Department, PTR Prentice Hall, 113 Sylvan Avenue, Englewood Cliffs, NJ 07632, Phone: 201-592-2863, Fax: 201-592-2249

Editorial/production supervision and interior design: *Camille Trentacoste*; Illustrator: *Gail Cocker-Bogusz*
Manufacturing manager: *Alexis Heydt*
Acquisitions editor: *Michael Meehan*; Editorial assistant: *Nancy Boylan*
Cover designer: *Doug DeLuca*
Cover photo: *Coelacanth/Lloyd Ullberg, Special Collections, California Academy of Sciences*

The cover depicts a coelacanth (pronounced C-la-canth), a butt-ugly fish that ichthyologists thought had been extinct for 70 million years. Then, in 1938, a specimen was caught off the coast of South Africa and taken to the local museum curator for identification. She recognized the significance of the find, and called in the experts—but not in time to prevent the fisherman stuffing and mounting his unique trophy! A second specimen was not caught until 1952. The limb-like fins of the coelacanth make this fish a "missing link" between ocean- and land-dwelling vertebrates. It is one vile-looking piece of seafood though.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-177429-8

Warning

Do not unscrew the cover of this book—there are no user-serviceable parts inside.

Dedication

I hereby dedicate this book to pizza, Dalmatian dogs, Sunday afternoons in a hammock, and comedy. The world would be a lot better off if there were more of these. I plan to become reacquainted with them all now the book is done.

In fact, I think I'll spend next Sunday afternoon swinging in a hammock, and laughing at my Dalmatian dog's attempts to eat pizza.

I would also like to acknowledge the fine products of the Theakston Brewing Company, Yorkshire, England.

Contents

Preface

Browsing in a bookstore recently, I was discouraged to see the dryness of so many C and C++ texts. Few authors conveyed the idea that anyone might enjoy programming. All the wonderment was squeezed out by long boring passages of prose. Useful perhaps, if you can stay awake long enough to read it. But programming isn't like that!

Programming is a marvellous, vital, challenging activity, and books on programming should brim over with enthusiasm for it! This book is educational, but also interesting in a way that puts the *fun* back into *functions*. If this doesn't seem like something you'll enjoy, then please put the book back on the shelf, but in a more prominent position. Thanks!

OK, now that we're among friends, there are already dozens and dozens of books on programming in C—what's different about this one?

Expert C Programming should be every programmer's *second* book on C. Most of the lessons, tips, and techniques here aren't found in any other book. They are usually pencilled in the margin of well-thumbed manuals or on the backs of old printouts, if they are written down at all. The knowledge has been accumulated over years of C programming by the author and colleagues in Sun's Compiler and Operating System groups. There are many interesting C stories and folklore, like the vending machines connected to the Internet, problems with software in outer space, and how a C bug brought down the entire AT&T long-distance phone network. Finally, the last chapter is an easy tutorial on C++, to help you master this increasingly-popular offshoot of C.

The text applies to ANSI standard C as found on PCs and UNIX systems. Unique aspects of C relating to sophisticated hardware typically found on UNIX platforms (virtual memory, etc.) are also covered in detail. The PC memory model and the Intel 8086 family are fully described in terms of their impact on C code. People who have already mastered the

basics of C will find this book full of all the tips, hints and shortcuts that a programmer usually picks up over a period of many years. It covers topics that many C programmers find confusing:

- What does `typedef struct bar {int bar;} bar;` actually mean?
- How can I pass different-sized multidimensional arrays to one function?
- Why, oh why, doesn't `extern char *p;` match `char p[100];` in another file?
- What's a bus error? What's a segmentation violation?
- What's the difference between `char *foo[]` and `char(*foo)[]` ?

If you're not sure about some of these, and you'd like to know how the C experts cope, then read on! If you already know all these things and everything else about C, get the book anyway to reinforce your knowledge. Tell the bookstore clerk that you're "buying it for a friend."

PvdL, Silicon Valley, Calif.

Acknowledgments

This isn't one of those lame little acknowledgment sections that you see in most other books: a string of feeble tributes to everyone the author ever borrowed money from, starting with his grade school buddies, proceeding through all his spouse's relatives, and ending with a grovelling but blatant attempt to curry favor with his thesis advisor ("and lastly to the great and powerful Professor Oz, whose work on whether the toilet paper should hang at the front of the roll or the back has done so much to resolve this crucial question"). No way! *This* is a genuine list of people who really and truly helped while I was writing this book. Everyone listed here has actually earned their acknowledgment. And you can bet that as I spend my leisurely days, and the princely royalty payments, on a beach in Tahiti I'll be thinking of them. Really I will!

I'd like to start with a special acknowledgment of the help given by Phil Gustafson and Brian Scearce, who read the entire manuscript in draft form and suggested many corrections and improvements. The effort was so intense that they have now deeded their bodies to science.

Thanks, too, to the friends and colleagues who read large parts of the work-in-progress:

Lee Bieber,

Keith Bierman (whose business card reads "Rabble-Rouser" for his title, and he is certainly the right man for the job),

Robert Corbett,

Rod Evans,

Doug Landauer,

Joseph McGuckin,

Walter Nielsen,

Charlie Springer (who taught me to count on my fingers in binary—you can count up to 1023 that way!),

Nicholas Sterling,

Panos Tsirigotis,

Richard Tuck,

who read parts of the manuscript and generously shared their candid, not to say blunt, views.

And I'm very grateful to the people who generally helped, often by patiently answering a stream of endless questions:

Chris Aoki,

Arindam Banerji,

Mark Brader,

Brent Callaghan (who hacked the audio feature into snoop),

David Chase,

Joseph T. Chew,

Adrian Cockcroft,

Sam Cramer,

Steve Dever,

Derek Dongray,

Joe Eykholt,

Roger Faulkner,

Mike Federwisch,

Dave Ford,

Burkhard Gerull of Sun Germany,

Rob Gingell,

Cathy Harris (for the plentiful supply of common sense),

Bruce Hildenbrand (and his amazing flying bicycle trick),

Mike Kazar,

Bob Jervis,

Diane Kelly,

Charles Lasner,

Bil Lewis,

Greg Limes,

Tim Marsland,

Marianne Mueller,

Eugene N. Miya,

Chuck Narad,

Bill Petro (for his inspiring and non-stop history lessons),

Trelford Pinkerton,

Alex Ramos,

Fred Sayward,

Bill Shannon,

Mark D. Smith,

Kathy Stark,

Dan Stein,

Steve Summit,

Paul Tomblin,

Wendy van der Linden (who came up with the bob-for-apples inheritance example for C++, and improved the rhythm of the "two 'l' null" verse),

Dock Williams,

Nigel "Gag Me" Witherspoon,

Brian Wong,

Tom Wong.

I'm grateful to editor Karin Ellison who let me mix metaphors, and several times poured midnight oil onto troubled waters on my behalf; to Astrid Julienne, who answered a lot of questions about Framemaker, and to Peter Van Coutren in the Sun Library.

I appreciate the knowledgeable help of the Prentice Hall staff, including Mike Meehan, Camille Trentacoste, Susan Aumack, Eloise Starkweather, and Nancy Boylan.

I'd also like to acknowledge the following people who didn't make a nuisance of themselves while I was working on this book. They generally stayed out of my hair, and they didn't screw anything up too badly. They're OK kinds of people, I guess:

Dirk Wibble-O'Dooley,

P. A. G. Embleton,

snopes.

Some of the material in this book was inspired by conversations, e-mail, net postings, and suggestions of colleagues in the industry. I have credited these sources where known, but if I have overlooked anyone, please accept my apologies.

PvdL, Silicon Valley, Calif.

This page intentionally left blank

Introduction

C code. C code run. Run code run...please!

—Barbara Ling

All C programs do the same thing: look at a character and do nothing with it.

—Peter Weinberger

Have you ever noticed that there are plenty of C books with suggestive names like *C Traps and Pitfalls*, or *The C Puzzle Book*, or *Obfuscated C and Other Mysteries*, but other programming languages don't have books like that? There's a very good reason for this!

C programming is a craft that takes years to perfect. A reasonably sharp person can learn the basics of C quite quickly. But it takes much longer to master the nuances of the language and to write enough programs, and enough different programs, to become an expert. In natural language terms, this is the difference between being able to order a cup of coffee in Paris, and (on the Metro) being able to tell a native Parisienne where to get off. This book is an advanced text on the ANSI C programming language. It is intended for people who are already writing C programs, and who want to quickly pick up some of the insights and techniques of experts.

Expert programmers build up a tool kit of techniques over the years; a grab-bag of idioms, code fragments, and deft skills. These are acquired slowly over time, learned from looking over the shoulders of more experienced colleagues, either directly or while maintaining code written by others. Other lessons in C are self-taught. Almost every beginning C programmer independently rediscovers the mistake of writing:

```
if (i=3)
```

instead of:

```
if (i==3)
```

Once experienced, this painful error (doing an assignment where comparison was intended) is rarely repeated. Some programmers have developed the habit of writing the literal first, like this: `if (3==i)`. Then, if an equal sign is accidentally left out, the compiler will complain about an "attempted assignment to literal." This won't protect you when comparing two variables, but every little bit helps.

The $20 Million Bug

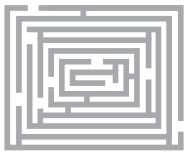
In Spring 1993, in the Operating System development group at SunSoft, we had a "priority one" bug report come in describing a problem in the asynchronous I/O library. The bug was holding up the sale of $20 million worth of hardware to a customer who specifically needed the library functionality, so we were extremely motivated to find it. After some intensive debugging sessions, the problem was finally traced to a statement that read:

```
x==2;
```

It was a typo for what was intended to be an assignment statement. The programmer's finger had bounced on the "equals" key, accidentally pressing it twice instead of once. The statement as written compared x to 2, generated true or false, and discarded the result.

C is enough of an expression language that the compiler did not complain about a statement which evaluated an expression, had no side-effects, and simply threw away the result. We didn't know whether to bless our good fortune at locating the problem, or cry with frustration at such a common typing error causing such an expensive problem. Some versions of the lint program would have detected this problem, but it's all too easy to avoid the automatic use of this essential tool.

This book gathers together many other salutary stories. It records the wisdom of many experienced programmers, to save the reader from having to rediscover everything independently. It acts as a guide for territory that, while broadly familiar, still has some unexplored corners. There are extended discussions of major topics like declarations and arrays/pointers, along with a great many hints and mnemonics. The terminology of ANSI C is used throughout, along with translations into ordinary English where needed.



Programming Challenge

OR

Handy Heuristic

Sample Box

Along the way, we have **Programming Challenges** outlined in boxes like this one.

These are suggestions for programs that you should write.

There are also **Handy Heuristics** in boxes of their own.

These are ideas, rules-of-thumb, or guidelines that work in practice. You can adopt them as your own. Or you can ignore them if you already have your own guidelines that you like better.

Convention

One convention that we have is to use the names of fruits and vegetables for variables (only in small code fragments, not in any real program, of course):

```
char pear[40];
double peach;
int mango = 13;
long melon = 2001;
```

This makes it easy to tell what's a C reserved word, and what's a name the programmer supplied. Some people say that you can't compare apples and oranges, but why not—they are both hand-held round edible things that grow on trees. Once you get used to it,

the fruit loops really seem to help. There is one other convention—sometimes we repeat a key point to emphasize it. In addition, we sometimes repeat a key point to emphasize it.

Like a gourmet recipe book, *Expert C Programming* has a collection of tasty morsels ready for the reader to sample. Each chapter is divided into related but self-contained sections; it's equally easy to read the book serially from start to finish, or to dip into it at random and review an individual topic at length. The technical details are sprinkled with many true stories of how C programming works in practice. Humor is an important technique for mastering new material, so each chapter ends with a "light relief" section containing an amusing C story or piece of software folklore to give the reader a change of pace.

Readers can use this book as a source of ideas, as a collection of C tips and idioms, or simply to learn more about ANSI C, from an experienced compiler writer. In sum, this book has a collection of useful ideas to help you master the fine art of ANSI C. It gathers all the information, hints, and guidelines together in one place and presents them for your enjoyment. So grab the back of an envelope, pull out your lucky coding pencil, settle back at a comfy terminal, and let the fun begin!

Some Light Relief—Tuning File Systems

Some aspects of C and UNIX are occasionally quite lighthearted. There's nothing wrong with well-placed whimsy. The IBM/Motorola/Apple PowerPC architecture has an E.I.E.I.O. instruction[1] that stands for "Enforce In-order Execution of I/O". In a similar spirit, there is a UNIX command, `tunefs`, that sophisticated system administrators use to change the dynamic parameters of a filesystem and improve the block layout on disk.

The on-line manual pages of the original `tunefs`, like all Berkeley commands, ended with a "Bugs" section. In this case, it read:

```
Bugs:
This program should work on mounted and active file systems, but it
doesn't. Because the superblock is not kept in the buffer cache, the
program will only take effect if it is run on dismounted file systems; if
run on the root file system, the system must be rebooted.
You can tune a file system, but you can't tune a fish.
```

Even better, the word-processor source had a comment in it, threatening anyone who removed that last phrase! It said:

```
Take this out and a UNIX Demon will dog your steps from now until the
time_t's wrap around.
```

1. Probably designed by some old farmer named McDonald.

When Sun, along with the rest of the world, changed to SVr4 UNIX, we lost this gem. The SVr4 manpages don't have a "Bugs" section—they renamed it "Notes" (does that fool anyone?). The "tuna fish" phrase disappeared, and the guilty party is probably being dogged by a UNIX demon to this day. Preferably lpd.

Programming Challenge

Computer Dating

When will the `time_t`'s wrap around?

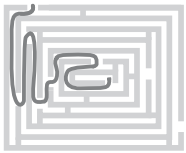
Write a program to find out.

1. Look at the definition of `time_t`. This is in file `/usr/include/time.h`.
2. Code a program to place the highest value into a variable of type `time_t`, then pass it to `ctime()` to convert it into an ASCII string. Print the string. Note that `ctime` has nothing to do with the language C, it just means "convert time."

For how many years into the future does the anonymous technical writer who removed the comment have to worry about being dogged by a UNIX daemon? Amend your program to find out.

1. Obtain the current time by calling `time()`.
2. Call `difftime()` to obtain the number of seconds between now and the highest value of `time_t`.
3. Format that value into years, months, weeks, days, hours, and minutes. Print it.

Is it longer than your expected lifetime?



Programming Solution

Computer Dating

The results of this exercise will vary between PCs and UNIX systems, and will depend on the way time_t is stored. On Sun systems, this is just a typedef for long. Our first attempted solution is

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

int main() {
  time_t biggest = 0x7FFFFFFF;

  printf("biggest = %s \n", ctime(&biggest) );
  return 0;
}
```

This gives a result of:

```
   biggest = Mon Jan 18 19:14:07 2038
```

However, this is not the correct answer! The function `ctime()` converts its argument into *local* time, which will vary from Coordinated Universal Time (also known as Greenwich Mean Time), depending on where you are on the globe. California, where this book was written, is eight hours behind London, and several years ahead.

We should really use the `gmtime()` function to obtain the largest UTC time value. This function doesn't return a printable string, so we call `asctime()`to get this. Putting it all together, our revised program is

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

int main() {
  time_t biggest = 0x7FFFFFFF;
```

Computer Dating (Continued)

```
  printf("biggest = %s \n", asctime(gmtime(&biggest)) );
  return 0;
}
```

This gives a result of:

```
   biggest = Tue Jan 19 03:14:07 2038
```

There! Squeezed another eight hours out of it!

But we're *still* not done. If you use the locale for New Zealand, you can get 13 more hours, assuming they use daylight savings time in the year 2038. They are on DST in January because they are in the southern hemisphere. New Zealand, because of its easternmost position with respect to time zones, holds the unhappy distinction of being the first country to encounter bugs triggered by particular dates.

Even simple-looking things can sometimes have a surprising twist in software. And anyone who thinks programming dates is easy to get right the first time probably hasn't done much of it.

This page intentionally left blank

C Through the Mists of Time 1

C is quirky, flawed, and an enormous success.

—Dennis Ritchie

the prehistory of C...the golden rule for compiler-writers...
early experiences with C...the standard I/O library and C preprocessor...
K&R C...the present day: ANSI C...it's nice, but is it standard?...
the structure of the ANSI C standard...reading the ANSI C standard for
fun, pleasure, and profit...how quiet is a "quiet change"?...some light
relief—the implementation-defined effects of pragmas

The Prehistory of C

The story of C begins, paradoxically, with a failure. In 1969 the great Multics project—a joint venture between General Electric, MIT, and Bell Laboratories to build an operating system—was clearly in trouble. It was not only failing to deliver the promised fast and convenient on-line system, it was failing to deliver anything usable at all. Though the development team eventually got Multics creaking into action, they had fallen into the same tarpit that caught IBM with OS/360. They were trying to create an operating system that was much too big and to do it on hardware that was much too small. Multics is a treasure house of solved engineering problems, but it also paved the way for C to show that small is beautiful.

As the disenchanted Bell Labs staff withdrew from the Multics project, they looked around for other tasks. One researcher, Ken Thompson, was keen to work on another operating system, and made several proposals (all declined) to Bell management. While waiting on official approval, Thompson and co-worker Dennis Ritchie amused themselves porting Thompson's "Space Travel" software to a little-used PDP-7. Space Travel simulated the major bodies of the solar system, and displayed them on a graphics screen along with a space craft that could be piloted and landed on the various planets. At the same time, Thompson worked intensively on providing the PDP-7 with the rudiments of a new operating system, much simpler and lighter-weight than Multics. Everything was

written in assembler language. Brian Kernighan coined the name "UNIX" in 1970, parodying the lessons now learned from Multics on what not to do. Figure 1-1 charts early C, UNIX, and associated hardware.

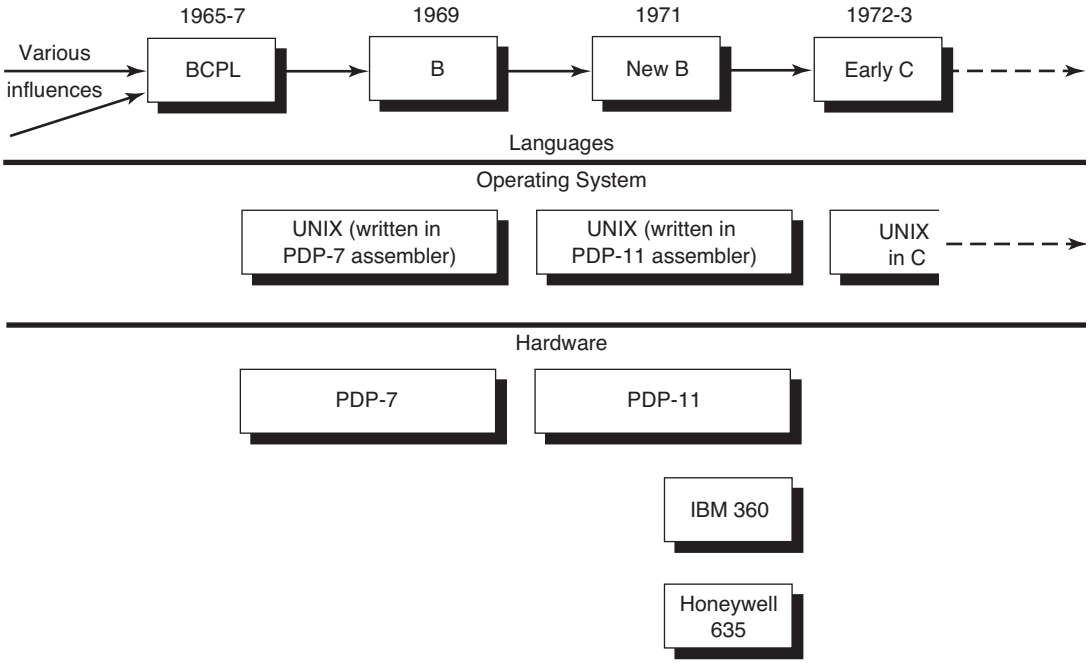


Figure 1-1 Early C, UNIX, and Associated Hardware

In this potential chicken-and-egg situation, UNIX definitely came well before C (and it's also why UNIX system time is measured in seconds since January 1, 1970—that's when time began). However, this is the story not of poultry, but of programming. Writing in assembler proved awkward; it took longer to code data structures, and it was harder to debug and understand. Thompson wanted the advantages of a high-level implementation language, but without the PL/I[1] performance and complexity problems that he had seen on Multics. After a brief and unsuccessful flirtation with Fortran, Thompson created the language B by simplifying the research language BCPL[2] so its interpreter would fit in the PDP-7's 8K word memory. B was never really successful; the hardware memory limits only provided room for an interpreter, not a compiler. The resulting slow performance prevented B from being used for systems programming of UNIX itself.

1. The difficulties involved in learning, using, and implementing PL/I led one programmer to pen this verse:
IBM had a PL/I / Its syntax worse than JOSS / And everywhere this language went / It was a total loss.
JOSS was an earlier language, also not noted for simplicity.

Software Dogma

The Golden Rule of Compiler-Writers:

Performance Is (almost) Everything.

Performance is almost everything in a compiler. There are other concerns: meaningful error messages, good documentation, and product support. These factors pale in comparison with the importance users place on raw speed. Compiler performance has two aspects: runtime performance (how fast the code runs) and compile time performance (how long it takes to generate code). Runtime performance usually dominates, except in development and student environments.

Many compiler optimizations cause longer compilation times but make run times much shorter. Other optimizations (such as dead code elimination, or omitting runtime checks) speed up both compile time and run time, as well as reducing memory use. The downside of aggressive optimization is the risk that invalid results may not be flagged. Optimizers are very careful only to do safe transformations, but programmers can trigger bad results by writing invalid code (e.g., referencing outside an array's bounds because they "know" that the desired variable is adjacent).

This is why performance is *almost* but not quite everything—if you don't get accurate results, then it's immaterial how fast you get them. Compiler-writers usually provide compiler options so each programmer can choose the desired optimizations. B's lack of success, until Dennis Ritchie created a high-performance compiled version called "New B," illustrates the golden rule for compiler-writers.

B simplified BCPL by omitting some features (such as nested procedures and some looping constructs) and carried forward the idea that array references should "decompose" into pointer-plus-offset references. B also retained the typelessness of BCPL; the only operand was a machine word. Thompson conceived the ++ and −− operators and added them to the B compiler on the PDP-7. The popular and captivating belief that they're in C because the PDP-11 featured corresponding auto-increment/decrement addressing modes

2. "BCPL: A Tool for Compiler Writing and System Programming," Martin Richards, Proc. AFIPS Spring Joint Computer Conference, 34 (1969), pp. 557-566. BCPL is not an acronym for the "Before C Programming Language", though the name *is* a happy coincidence. It is the "Basic Combined Programming Language"—"basic" in the sense of "no frills"—and it was developed by a combined effort of researchers at London University and Cambridge University in England. A BCPL implementation was available on Multics.

is wrong! Auto increment and decrement predate the PDP-11 hardware, though it is true that the C statement to copy a character in a string:

```
*p++ = *s++;
```

can be compiled particularly efficiently into the PDP-11 code:

```
movb (r0)+,(r1)+
```

leading some people to wrongly conclude that the former was created especially for the latter.

A typeless language proved to be unworkable when development switched in 1970 to the newly introduced PDP-11. This processor featured hardware support for datatypes of several different sizes, and the B language had no way to express this. Performance was also a problem, leading Thompson to reimplement the OS in PDP-11 assembler rather than B. Dennis Ritchie capitalized on the more powerful PDP-11 to create "New B," which solved both problems, multiple datatypes, and performance. "New B"—the name quickly evolved to "C"—was compiled rather than interpreted, and it introduced a type system, with each variable described in advance of use.

Early Experiences with C

The type system was added primarily to help the compiler-writer distinguish floats, doubles, and characters from words on the new PDP-11 hardware. This contrasts with languages like Pascal, where the purpose of the type system is to protect the programmer by restricting the valid operations on a data item. With its different philosophy, C rejects strong typing and permits the programmer to make assignments between objects of different types if desired. The type system was almost an afterthought, never rigorously evaluated or extensively tested for usability. To this day, many C programmers believe that "strong typing" just means pounding extra hard on the keyboard.

Many other features, besides the type system, were put in C for the C compiler-writer's benefit (and why not, since C compiler-writers were the chief customers for the first few years). Features of C that seem to have evolved with the compiler-writer in mind are:

- **Arrays start at 0 rather than 1.** Most people start counting at 1, rather than zero. Compiler-writers start with zero because we're used to thinking in terms of offsets. This is sometimes tough on non-compiler-writers; although `a[100]` appears in the definition of an array, you'd better not store any data at `a[100]`, since `a[0]` to `a[99]` is the extent of the array.

- **The fundamental C types map directly onto underlying hardware.** There is no built-in complex-number type, as in Fortran, for example. The compiler-writer does not have to invest any effort in supporting semantics that are not directly provided by the hardware. C didn't support floating-point numbers until the underlying hardware provided it.

- **The `auto` keyword is apparently useless.** It is only meaningful to a compiler-writer making an entry in a symbol table—it says *this storage is automatically allocated on entering the block* (as opposed to global static allocation, or dynamic allocation on the heap). Auto is irrelevant to other programmers, since you get it by default.

- **Array names in expressions "decay" into pointers.** It simplifies things to treat arrays as pointers. We don't need a complicated mechanism to treat them as a composite object, or suffer the inefficiency of copying everything when passing them to a function. But don't make the mistake of thinking arrays and pointers are always equivalent; more about this in Chapter 4.

- **Floating-point expressions were expanded to double-length-precision everywhere.** Although this is no longer true in ANSI C, originally real number constants were always doubles, and float variables were always converted to double in all expressions. The reason, though we've never seen it appear in print, had to do with PDP-11 floating-point hardware. First, conversion from float to double on a PDP-11 or a VAX is really cheap: just append an extra word of zeros. To convert back, just ignore the second word. Then understand that some PDP-11 floating-point hardware had a mode bit, so it would do either all single-precision or all double-precision arithmetic, but to switch between the two you had to change modes.

  Since most early UNIX programs weren't floating-point-intensive, it was easier to put the box in double-precision mode and leave it there than for the compiler-writer to try to keep track of it!

- **No nested functions (functions contained inside other functions).** This simplifies the compiler and slightly speeds up the runtime organization of C programs. The exact mechanism is described in Chapter 6, "Poetry in Motion: Runtime Data Structures."

- **The `register` keyword.** This keyword gave the compiler-writer a clue about what variables the programmer thought were "hot" (frequently referenced), and hence could usefully be kept in registers. It turns out to be a mistake. You get better code if the compiler does the work of allocating registers for individual uses of a variable, rather than reserving them for its entire lifetime at declaration. Having a `register` keyword simplifies the compiler by transferring this burden to the programmer.

There were plenty of other C features invented for the convenience of the C compiler-writer, too. Of itself this is not necessarily a bad thing; it greatly simplified the language, and by shunning complicated semantics (e.g., generics or tasking in Ada; string handling in PL/I; templates or multiple inheritance in C++) it made C much easier to learn and to implement, and gave faster performance.

Unlike most other programming languages, C had a lengthy evolution and grew through many intermediate shapes before reaching its present form. It has evolved through years of practical use into a language that is tried and tested. The first C compiler appeared circa 1972, over 20 years ago now. As the underlying UNIX system grew in popularity, so C was carried with it. Its emphasis on low-level operations that were directly supported by the hardware brought speed and portability, in turn helping to spread UNIX in a benign cycle.

The Standard I/O Library and C Preprocessor

The functionality left out of the C compiler had to show up somewhere; in C's case it appears at runtime, either in application code or in the runtime library. In many other languages, the compiler plants code to call runtime support implicitly, so the programmer does not need to worry about it, but almost all the routines in the C library must be explicitly called. In C (when needed) the programmer must, for example, manage dynamic memory use, program variable-size arrays, test array bounds, and carry out range checks for him or herself.

Similarly, I/O was originally not defined within C; instead it was provided by library routines, which in practice have become a standardized facility. The portable I/O library was written by Mike Lesk[3] and first appeared around 1972 on all three existing hardware platforms. Practical experience showed that performance wasn't up to expectations, so the library was tuned and slimmed down to become the standard I/O library.

The C preprocessor, also added about this time at the suggestion of Alan Snyder, fulfilled three main purposes:

- String replacement, of the form "change all *foo* to *baz*", often to provide a symbolic name for a constant.
- Source file inclusion (as pioneered in BCPL). Common declarations could be separated out into a header file, and made available to a range of source files. Though the ".h" convention was adopted for the extension of header files, unhappily no convention arose for relating the header file to the object library that contained the corresponding code.

3. It was Michael who later expressed the hilariously ironic rule of thumb that "designing the system so that the manual will be as short as possible minimizes learning effort." (Datamation, November 1981, p.146). Several comments come to mind, of which "Bwaa ha ha!" is probably the one that minimizes learning effort.

- Expansion of general code templates. Unlike a function, the same macro argument can take different types on successive calls (macro actual arguments are just slotted unchanged into the output). This feature was added later than the first two, and sits a little awkwardly on C. White space makes a big difference to this kind of macro expansion.

```
#define a(y) a_expanded(y)
a(x);
```

expands into:

```
a_expanded(x);
```

However,

```
#define a (y)   a_expanded (y)
a(x);
```

is transformed into:

```
(y)   a_expanded (y)(x);
```

Not even close to being the same thing. The macro processor could conceivably use curly braces like the rest of C to indicate tokens grouped in a block, but it does not.

There's no extensive discussion of the C preprocessor here; this reflects the view that the only appropriate use of the preprocessor is for macros that don't require extensive discussion. C++ takes this a step further, introducing several conventions designed to make the preprocessor completely unnecessary.

Software Dogma

C Is Not Algol

Writing the UNIX Version 7 shell (command interpreter) at Bell Labs in the late 1970's, Steve Bourne decided to use the C preprocessor to make C a little more like Algol-68. Earlier at Cambridge University in England, Steve had written an Algol-68 compiler, and found it easier to debug code that had explicit "end statement" cues, such as `if ... fi` or `case ... esac`. Steve thought it wasn't easy enough to tell by looking at a "`}`" what it matches. Accordingly, he set up many preprocessor definitions:

```
#define STRING char *
#define IF if(
#define THEN ){
#define ELSE } else {
#define FI ;}
#define WHILE while (
#define DO ){
#define OD ;}
#define INT int
#define BEGIN {
#define END }
```

This enabled him to code the shell using code like this:

```
INT  compare(s1, s2)
     STRING s1;
     STRING s2;
BEGIN
     WHILE *s1++ == *s2
     DO IF *s2++ == 0
          THEN return(0);
          FI
     OD
     return(*--s1 - *s2);
END
```

C Is Not Algol (Continued)

Now let's look at that again, in C this time:

```
int compare(s1, s2)
      char * s1, *s2;
{
      while (*s1++ == *s2) {
              if (*s2++ == 0) return (0);
      }
      return (*--s1 - *s2);
}
```

This Algol-68 dialect achieved legendary status as the Bourne shell permeated far beyond Bell Labs, and it vexed some C programmers. They complained that the dialect made it much harder for other people to maintain the code. The BSD 4.3 Bourne shell (kept in /bin/sh) is written in the Algol subset to this day!

I've got a special reason to grouse about the Bourne shell—it's my desk that the bugs reported against it land on! Then I assign them to Sam! And we do see our share of bugs: the shell doesn't use malloc, but rather does its own heap storage management using sbrk. Maintenance on software like this too often introduces a new bug for every two it solves. Steve explained that the custom memory allocator was done for efficiency in string-handling, and that he never expected anyone except himself to see the code.

The Bournegol C dialect actually inspired The International Obfuscated C Code Competition, a whimsical contest in which programmers try to outdo each other in inventing mysterious and confusing programs (more about this competition later).

Macro use is best confined to naming literal constants, and providing shorthand for a few well-chosen constructs. Define the macro name all in capitals so that, in use, it's instantly clear it's not a function call. Shun any use of the C preprocessor that modifies the underlying language so that it's no longer C.

K&R C

By the mid 1970's the language was recognizably the C we know and love today. Further refinements took place, mostly tidying up details (like allowing functions to return structure values) or extending the basic types to match new hardware (like adding the keywords `unsigned` and `long`). In 1978 Steve Johnson wrote *pcc*, the portable C compiler. The source was made available outside Bell Labs, and it was very widely ported, forming a common basis for an entire generation of C compilers. The evolutionary path up to the present day is shown in Figure 1-2.

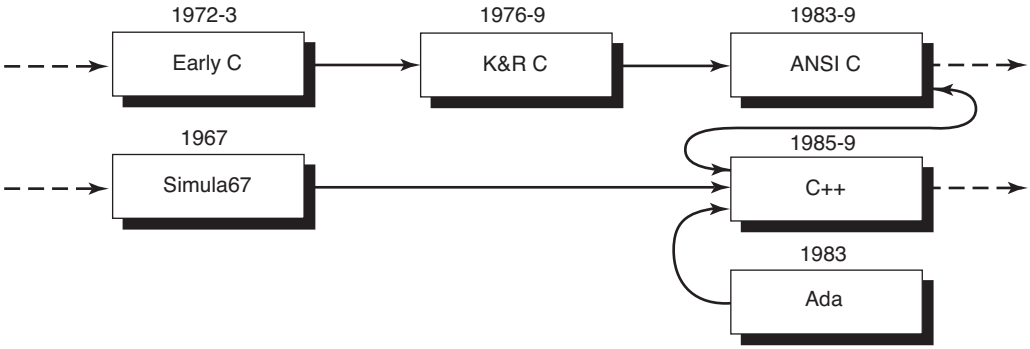


Figure 1-2 Later C

Software Dogma

An Unusual Bug

One feature C inherited from Algol-68 was the assignment operator. This allows a repeated operand to be written once only instead of twice, giving a clue to the code generator that operand addressing can be similarly thrifty. An example of this is writing `b+=3` as an abbreviation for `b=b+3`. Assignment operators were originally written with assignment first, not the operator, like this: `b=+3`. A quirk in B's lexical analyzer made it simpler to implement as *=op* rather than *op=* as it is today. This form was confusing, as it was too easy to mix up

```
b=-3; /* subtract 3 from b */
```

and

```
b= -3; /* assign -3 to b */
```

The feature was therefore changed to its present ordering. As part of the change, the code formatter `indent` was modified to recognize the obsolete form of assignment operator and swap it round to operator assignment. This was very bad judgement indeed; no formatter should ever change anything except the white space in a program. Unhappily, two things happened. The programmer introduced a bug, in that almost anything (that wasn't a variable) that appeared after an assignment was swapped in position.

If you were "lucky" it would be something that would cause a syntax error, like

```
epsilon=.0001;
```

An Unusual Bug (Continued)

being swapped into

```
epsilon.=0001;
```

But a source statement like

```
valve=!open; /* valve is set to logical negation of open */
```

would be silently transmogrified into

```
valve!=open; /* valve is compared for inequality to open */
```

which compiled fine, but did not change the value of `valve`.

The second thing that happened was that the bug lurked undetected. It was easy to work around by inserting a space after the assignment, so as the obsolete form of assignment operator declined in use, people just forgot that indent had been kludged up to "improve" it. The indent bug persisted in some implementations up until the mid-1980's. Highly pernicious!

In 1978 the classic C bible, *The C Programming Language*, was published. By popular acclamation, honoring authors Brian Kernighan and Dennis Ritchie, the name "K&R C" was applied to this version of the language. The publisher estimated that about a thousand copies would be sold; to date (1994) the figure is over one and a half million (see Figure 1-3). C is one of the most successful programming languages of the last two decades, perhaps the most successful. But as the language spread, the temptation to diverge into dialects grew.

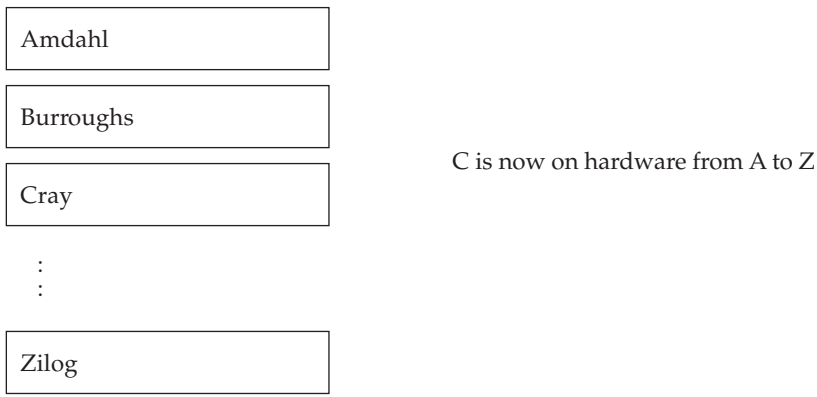


Figure 1-3 Like Elvis, C is Everywhere

The Present Day: ANSI C

By the early 1980's, C had become widely used throughout the industry, but with many different implementations and changes. The discovery by PC implementors of C's advantages over BASIC provided a fresh boost. Microsoft had an implementation for the IBM PC which introduced new keywords (`far`, `near`, etc.) to help pointers to cope with the irregular architecture of the Intel 80x86 chip. As many other non-pcc-based implementations arose, C threatened to go the way of BASIC and evolve into an ever-diverging family of loosely related languages.

It was clear that a formal language standard was needed. Fortunately, there was much precedent in this area—all successful programming languages are eventually standardized. However, the problem with standards manuals is that they only make sense if you already know what they mean. If people write them in English, the more precise they try to be, the longer, duller and more obscure they become. If they write them using mathematical notation to define the language, the manuals become inaccessible to too many people.

Over the years, the manuals that define programming language standards have become longer, but no easier to understand. The Algol-60 Reference Definition was only 18 pages long for a language of comparable complexity to C; Pascal was described in 35 pages. Kernighan and Ritchie took 40 pages for their original report on C; while this left several holes, it was adequate for many implementors. ANSI C is defined in a fat manual over 200 pages long. This book is, in part, a description of practical use that lightens and expands on the occasionally opaque text in the ANSI Standard document.

In 1983 a C working group formed under the auspices of the American National Standards Institute. Most of the process revolved around identifying common features, but there were also changes and significant new features introduced. The `far` and `near` keywords were argued over at great length, but ultimately did not make it into the mildly UNIX-centric ANSI standard. Even though there are more than 50 million PC's out there, and it is by far the most widely used platform for C implementors, it was (rightly in our view) felt undesirable to mutate the language to cope with the limitations of one specific architecture.

Handy Heuristic

Which Version of C to Use?
At this point, anyone learning or using C should be working with ANSI C, not K&R C.

The language standard draft was finally adopted by ANSI in December 1989. The international standards organization ISO then adopted the ANSI C standard (unhappily removing the very useful "Rationale" section and making trivial—but very annoying—formatting and paragraph numbering changes). ISO, as an international body, is technically the senior organization, so early in 1990 ANSI readopted ISO C (again excluding the Rationale) back in place of its own version. In principle, therefore, we should say that the C standard adopted by ANSI is ISO C, and we should refer to the language as ISO C. The Rationale is a useful text that greatly helps in understanding the standard, and it's published as a separate document.[4]

4. The ANSI C Rationale (only) is available for free by anonymous ftp from the site ftp.uu.net, in directory /doc/standards/ansi/X3.159-1989/.
(If you're not familiar with anonymous ftp, run, don't walk, to your nearest bookstore and buy a book on Internet, before you become <insert lame driving metaphor of choice> on the Information Highway.)
The Rationale has also been published as a book, *ANSI C Rationale*, New Jersey, Silicon Press, 1990. The ANSI C standard itself is not available by ftp anywhere because ANSI derives an important part of its revenue from the sale of printed standards.