C PROGRAMMING GUIDELINES

C programs can be portable to a wide variety of processors, efficient in their use of machine resources, and readable by future maintainers.

Programming projects working in C should use guidelines for style and usage to achieve portability, readability and consistency.

Thomas Plum

C PROGRAMMING GUIDELINES

Thomas Plum

Prentice-Hall, Inc.
Englewood Cliffs, New Jersey 07632

This Prentice-Hall, Inc., edition published 1984.

Library of Congress Catalog Card Number: 84-60386
ISBN: 0-13-109992-2 (Prentice-Hall, Inc., edition)
ISBN: 0-911537-03-1 (Plum Hall Inc edition)

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-109992-2

PRENTICE-HALL INTERNATIONAL, INC., London
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, Sydney
EDITORA PRENTICE-HALL DO BRASIL, LTDA., Rio de Janeiro
PRENTICE-HALL CANADA INC., Toronto
PRENTICE-HALL OF INDIA PRIVATE LIMITED, New Delhi
PRENTICE-HALL OF JAPAN, INC., Tokyo
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., Singapore
WHITEHALL BOOKS LIMITED, Wellington, New Zealand

For Joan

PREFACE

C is a highly portable language which generates efficient code for a wide variety of modern computers. It was originally implemented on the UNIX* operating system for the DEC PDP-11 by Dennis Ritchie. Within Bell Laboratories it is widely used for systems and application programming. In the years since it was made available to universities and commercial organizations, it has proved valuable for systems programming, switching and communications, microprocessors, text processing, process control, test equipment, and the creation of numerous application packages. Its audience has been widened by compilers for many operating systems besides UNIX; these compilers have been produced by numerous software companies, among which the pioneer was Whitesmiths, Ltd. Whitesmiths' Idris* operating system (on which this book was composed) is, like UNIX, almost entirely written in C.

Programming standards can be valuable to any organization producing programs in C. Its compact notation and absence of restrictions can be used in an undisciplined fashion to produce programs unreadable to any but the original author -- if indeed the author can read them after the passage of time! A uniformity of style can make the thankless task of the maintainer much easier. The code can be modified much easier when standards are followed.

In addition to aiding consistency, standards also enhance portability of the source code, since one of the important virtues of the C language is its combination of portability and efficiency.

However, achieving portability requires attention to a small set of problem areas, which are addressed by the portability standards in this book.

---

*Trademarks: Idris of Whitesmiths, Ltd.; UNIX of AT&T Bell Laboratories.

Disagreements over programming style have been a primary obstacle to teams attempting to work closely together. One suggestion for preventing style arguments is for each project to choose its own standard in the early phases of the project. The layout of this book was chosen to facilitate its use in a pick-and-choose fashion. Space has been left for local notes so that the book could be used to keep a record of meetings regarding style agreements. Each section is named (in the style of UNIX manuals) as well as numbered, for ease of later reference. The author asks in accordance with copyright laws that this book not be run through the copying machine; Plum Hall Inc will make available on a reasonable license arrangement both hard-copy and machine-readable originals for projects that wish to incorporate this material in their own standard.

In this edition of *C Programming Guidelines*, the usage of types, function names, and indentation conforms to the format of the UNIX manuals published by Bell Laboratories. The style usage is consistent with *Learning to Program in C*, by Thomas Plum (Plum Hall Inc, 1983).

Previous editions of these guidelines made reference to the (now obsolescent) UNIX Version 6 compiler. Such references are omitted from this edition. Where portability is a concern, do not use the V6 compiler.

This book is also available in the format of the manuals from Whitesmiths, Ltd., in *C Programming Standards and Guidelines: Version W (Whitesmiths)*. The discussions of portability, however, apply to both systems.

During 1983, a committee was formed by the American National Standards Institute (ANSI) to standardize the definition of the C language. Previous editions of this book had the word "standards" in the title. To avoid any possible confusion with the development of the ANSI standard, I have dropped the word "standard" from the title of this book. I thank the other members of the ANSI X3J11 committee for enlightening discussions about the C language. In my opinion, based on information currently available, programs written according to the guidelines in this book should be well prepared for the eventual ANSI standard; however, no one can give any official guarantees as of this date.

For thoughtful comments and suggestions on various drafts of this material, I am indebted to Tom Bishop, Debbie Fullam, David Graham, Joan Hall, Brian Kernighan, Bill Koenig, Mark Krieger, Eli Lamb, Ian MacLeod, Bill Masek, Paul Matthews, Bill Plauger, Ed Rathje, Chaim Schaap, and Steve Schustack.

And for their steadfast support, my heartfelt thanks to Suzanne Battista, Linda Deutsch, Anne Hall, Joan Hall, Karl Karlstrom, and Sonya Whynman.

Thomas Plum

CONTENTS

NAME

0.1standards - standards and guidelines

STANDARD

Criteria labeled as "STANDARD" are mandatory for all code included in a product.

The need for exceptions may occasionally arise, but the exception requires a specific justification, and the justification should be documented with the source code. This is a "permissive" approach to exceptions; this book is not intended to satisfy any legal, auditing, or quality-assurance criteria.

Project-wide exceptions to the standards may be justified and should be documented as an appendix to the standard.

Criteria labeled as "GUIDELINE" are recommended practices. Experience has shown that differing approaches can coexist in these areas. It is expected that, in general, a majority of programmers will follow the guidelines, so that they represent a widely-accepted pattern.

NAME

1.1lexdata - lexical rules for variables

STANDARD

Variable names should be written all in lower case. Many compilers require names to be distinct within 8 characters, but longer names can be useful for readability. (For portability, externals should be distinct within 6 characters.) All names should be explicitly declared. The sequence of declarations should be as follows:

external names, alphabetized by name within type;

other names, similarly alphabetized by name within type.

Initializers should be written using the equal-sign, with only one variable declared per source line:

```
short n = MAX;
short m = MAX;
```

Initializers of structures, unions, and arrays should be formatted with one row per line:

```
static short x[2][5] =
    {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    };
```

Declarations should have only one space between type and variable name, and comments are attached with at least one tab:

```
bool mpflg = NO;	/* preprocessed macros flag */
bool ff;	/* fork flag */
```

JUSTIFICATION

The rules pinpoint the location of declarations, avoid conflicts of upper- and lower-case names, and encourage documentation of the meaning of variable names.

ALTERNATIVES

Variable names should be aligned in a tabbed column, and descriptive comments should be attached at a lined-up tab position:

```
bool    ff;             /* fork flag */
bool    mpflg = NO;     /* preprocessed macros flag */
```

Alternatives such as this one, which require columnar layout of source code, should be adopted only when convenient full-screen editing is available to all programmers. Otherwise, the difficulties of program revision offset any readability advantages.

[LOCAL NOTES]

NAME

1.2names - choosing variable names

GUIDELINE

Names should be chosen to be meaningful; their meaning should be exact and should be preserved throughout the program.

For example, variables which count something should be initialized to the count which is valid at that point; i.e., if the count is initially zero, the variable should be initialized to 0, not to -1 or some other number.

This means that each variable has an invariant (i.e. unchanging) meaning -- a property that is true throughout the program. The readability of the code is enhanced by minimizing the "domains of exceptions", which are the regions of the program in which the invariant property fails. For example, in this short loop, the variable nc has the invariant property of being equal to the number of characters read so far. The only exception to the property is during the time between reading a character and incrementing the counter:

```
short nc;    /* number of characters */

nc = 0;
while (getchar() != EOF)
    ++nc;
```

Abbreviations for meaningful names should be chosen by a uniform scheme. For example, use the leading consonant of each word in a name.

Abbreviations should not form letter combinations that suggest unintended meanings; the name inch is a misleading abbreviation for "input character." Similarly, names should not create misleading phonemes; the name metach (abbreviation for "meta-character") forms the phonemes "me-tach" in English, obscuring the meaning.

Names should not be re-defined in inner blocks.

A special case of meaningful names is the use of standard short names like _c_ for characters, _i_, _j_, _k_ for indexes, _n_ for counters, _p_ or _q_ for pointers, and _s_ for strings.

In separate functions, variables with identical meanings can have the same name. But when the meanings are only similar or coincidental, use different names.

Names over four characters in length should differ by at least two characters:

```
systst, sysstst /* easily confused */
```

JUSTIFICATION

Readability of the code is greatly enhanced by the reader's ability to construct natural assertions about the meaning of names anywhere they appear in the code.

[LOCAL NOTES]

NAME

1.3stdtypes - standard defined-types

STANDARD

Programs should use a project-wide standard set of data-type names.

The set of standard types presented here are a mixture of standard C types (sometimes with usage restrictions) and defined-types defined by the header file <stdtyp.h> (presented later in this section).

There are two purposes for this usage of types: portability to the widest range of machines and compilers, and semantic clarity regarding the usage of the data. As regards portability, some of these types have simple mappings onto a wide range of compilers, and others have a more complex mapping.

First, the simple mappings:

	Numbers (signed)	Numbers (unsigned)	Bit Masks	Text Characters	Boolean (0 or 1)
char	-	-	tbits	char	tbool
short	short	ushort	bits	metachar	-
long	long	-	lbits	-	-
float	float	-	-	-	-
double	double	-	-	-	-

```
tbits     - an 8+ bit integer used for bit manipulation
char      - an 8+ bit item used only for characters
tbool     - an 8+ bit integer, but only tested against zero

short     - a 16+ bit signed integer used for a quantity
ushort    - a 16+ bit unsigned integer used for a quantity
bits      - a 16+ bit integer used for bit manipulation
metachar  - a 16+ bit character (either a char or EOF)

long      - a 32+ bit signed integer used for a quantity
lbits     - a 32+ bit integer used for bit manipulation

float     - single precision floating point number
double    - double precision floating point number
```

[LOCAL NOTES]

Note that there are no int types in the preceding table. The intent is to avoid careless dependence on the int size of the computer. In this standard-type scheme, the only type that always maps to int size is bool, which is provided for functions that return a yes-or-no result.

There are, however, two uses of the int type which are appropriate for portable programs. First of all, a function's returned value may be written as int. This avoids compiler-dependent differences in the handling of returned values. Furthermore, many existing library functions are defined to have int returned values. The second usage of int is for register integer variables. Here again, the purpose is to avoid compiler-dependent differences: for example, some compilers treat "register char" as "register int" while others treat it as "auto char." In both usages (returned values and registers), programs should not assume that int contains any more than 16 bits.

Thus, two more types are added to the standard usages:

```
bool        - int, tested only for zero or non-zero
int         - for function returned values and registers
```

The type void is available as a keyword in some recent compilers (see Chapter 6 for examples). In this scheme, compatibility with older compilers is provided by defining void to be int. (Eliminate this definition from your stdtyp.h if your compiler supports void.)

A new type has been added to stdtyp.h since the previous version of these guidelines: sizetype is the proper type for holding the sizeof any object. It is the proper size for the storage size passed to allocation functions such as calloc. Up until recently, an unsigned int was adequate for this purpose, but in some forthcoming environments an unsigned long may be required.

Functions such as calloc must return a pointer which is adequate to hold a pointer to any C data object; char * will